A Standard Definition of Elm?

(Draft version 0.4)

Rupert Smith

2026

Contents

Preface **vii**

1 Introduction **1**

2 Syntax of the Core **4**
2.1 Reserved Words 4
2.2 Special Constants 4
2.3 Comments 6
2.4 Identifiers 6
2.5 Lexical Analysis 8
2.6 Layout 9
2.7 Infixed Operators 10
2.8 Derived Forms 11
2.9 Grammar 11
2.10 Syntactic Restrictions 14

3 Syntax of Modules **16**
3.1 Reserved Words 16
3.2 Identifiers 16
3.3 Infixed Operators 17
3.4 Layout 17
3.5 Grammar for Modules 18
3.6 Default Imports 20
3.7 Syntactic Restrictions 20

4 Static Semantics for the Core **23**
4.1 Simple Objects 23
4.2 Compound Objects 24
4.3 Projection, Injection and Modification 26
4.4 Classes and Admissible Types 27
4.5 Types and Type Functions 28
4.6 Type Schemes 28
4.7 Scope of Explicit Type Variables 29
4.8 Closure 30

4.9 Inference Rules 30
4.10 Principal Derivations 38
4.11 Further Restrictions 39

5 Static Semantics for Modules 42
5.1 Semantic Objects 42
5.2 Interfaces 43
5.3 Provider Maps 44
5.4 Port Types 45
5.5 Inference Rules 46
5.6 Principal Derivations 54
5.7 Further Restrictions 54

6 Dynamic Semantics for the Core 55
6.1 Reduced Syntax 55
6.2 Number Specialisation 55
6.3 Simple Objects 57
6.4 Compound Objects 58
6.5 Basic Values 59
6.6 Function Closures 60
6.7 Inference Rules 61

7 Dynamic Semantics for Modules 69
7.1 Reduced Syntax 69
7.2 Compound Objects 70
7.3 Merging Environments 71
7.4 Record Constructors 71
7.5 Top-level Value Groups 72
7.6 Inference Rules 72

8 Tasks, Commands, Subscriptions and Ports 78
8.1 Simple Objects 78
8.2 Compound Objects 79
8.3 Inference Rules 81
8.4 Tasks 81
8.5 Commands and Subscriptions 84
8.6 The Effect Values of the Basis 85
8.7 Ports 86
8.8 Messages 92

9 Programs 93
9.1 Syntax of Programs 93
9.2 Elaboration of Programs 94
9.3 Evaluation of Programs 97

9.4 Execution . . . 98
9.4.1 Configurations and Events . . . 98
9.4.2 Dispatch and Start . . . 100
9.4.3 Transitions . . . 101
9.5 Observable Behaviour . . . 105
9.6 Views and Other Programs . . . 105

A Derived Forms **107**
A.1 Grouping the Definitions of a Let . . . 110
A.2 Grouping the Declarations of a Module . . . 111

B Full Grammar **113**

C The Initial Static Basis **118**
C.1 Type Names . . . 118
C.2 The Module Environment . . . 120
C.3 Basics . . . 122
C.4 Further Data Types . . . 124
C.5 Debugging, Programs and Effects . . . 124

D The Initial Dynamic Basis **127**
D.1 Special Values . . . 128
D.1.1 Int Values . . . 128
D.1.2 Floats . . . 128
D.1.3 Characters and Strings . . . 129
D.2 The Value Environments . . . 130
D.3 Values Written in Elm . . . 133
D.4 The Function APPLY . . . 135
D.4.1 Int Arithmetic . . . 136
D.4.2 Floating-Point Arithmetic . . . 137
D.4.3 Powers and Elementary Functions . . . 138
D.4.4 Conversions . . . 140
D.4.5 Equality . . . 140
D.4.6 Order . . . 141
D.4.7 Booleans, Appending and Functions . . . 145
D.4.8 Debugging . . . 145
D.4.9 Arrays . . . 145
D.4.10 Bytes . . . 146
D.4.11 Effect Values . . . 146

E Implementation-Dependent Details **147**
E.1 The Int Range and Normalisation . . . 148
E.1.1 The Range and Normalisation . . . 148
E.1.2 Integer Division and Remainders . . . 149

E.1.3 Powers with a Negative Exponent . . . 149
E.1.4 Conversion from Float . . . 149
E.1.5 Negative Zero . . . 150
E.2 Characters and Strings . . . 150
E.2.1 What a Character Is . . . 151
E.2.2 The Order of Characters and Strings . . . 151
E.2.3 The Unicode Version of Identifiers . . . 152
E.2.4 Case Mapping . . . 152
E.3 Floating-Point Operations . . . 152
E.3.1 `compare` with NaN . . . 153
E.3.2 The Order Operators . . . 153
E.3.3 `min`, `max` and `clamp` . . . 154
E.3.4 Special Values and Accuracy . . . 154
E.4 Equality . . . 154
E.4.1 Identity . . . 155
E.4.2 Functions . . . 155
E.5 Bitwise Operations and String Conversion . . . 155
E.5.1 Bitwise Operations . . . 156
E.5.2 Conversion between Numbers and Strings . . . 156
E.6 Evaluation Order and Debug . . . 156
E.6.1 The Order of Evaluation within a Phrase . . . 157
E.6.2 Top-level Values . . . 157
E.6.3 Number Specialisation . . . 158
E.6.4 Debug . . . 159
E.7 Resource Limits . . . 159
E.7.1 The Size of Phrases . . . 160
E.7.2 Stack and Memory . . . 160
E.8 Platform and Ports . . . 160
E.8.1 The Order of Commands . . . 161
E.8.2 Incoming Port Values . . . 162
E.8.3 Outgoing Port Values . . . 162
E.8.4 Values at the Boundary . . . 163
E.8.5 Bytes at the Boundary . . . 163
E.8.6 Flags . . . 164
E.8.7 Processes . . . 164
E.8.8 Managers and Programs on a Platform . . . 165
E.9 Extensions of eco . . . 165
E.10 Known Divergences . . . 165
E.10.1 Int Arithmetic . . . 166
E.10.2 Float, Order and Equality . . . 166
E.10.3 Characters and Strings . . . 167
E.10.4 Evaluation . . . 167
E.10.5 Programs and Ports . . . 168
E.10.6 Elaboration . . . 169

F Elm and Standard ML **170**
F.1 What Is the Same 170
F.2 Differences by Construct 170

Preface

Until now the meaning of an Elm program has been settled in practice by what one implementation does with it: the Elm compiler for JavaScript, together with the JavaScript kernel of its core library. The language is described in prose, in a guide and in the documentation of its libraries, and that description serves a programmer learning the language well. It serves less well three other readers. An implementer must decide, for every construction, what the program means, and cannot always tell from the existing implementation whether a behaviour is the language's or the accident of a host. A programmer who wants to show that two programs are equivalent, or that a function never fails, needs a meaning precise enough to argue from. And two parties who must agree on what a given program does need a text that both accept as the authority.

This book is a formal definition of both the grammar and the meaning of Elm, version 0.19.1. The occasion for it is a second implementation, eco, which compiles Elm to native code and has its own runtime. Two implementations on different hosts give different answers wherever an answer was a consequence of the host rather than of the language: the range of integers, what a character is, what happens when an integer is divided by zero. Once there are two, each such question must be answered once, independently of both, and precisely enough that either implementation can be checked against the answer. We call the language so defined *Standard Elm*.

Standard Elm

Elm is a functional language for writing interactive programs that run under the control of a host, first of all a web browser. It is pure: a phrase denotes a value, and evaluating it changes nothing, since the language has no assignable variables, no mutable data and no input or output. It is strict: the argument of a function is evaluated before the function is applied. It is statically typed, with types inferred and polymorphism introduced at definitions. In its semantics it is extremely close to ML: values, closures, custom types with constructors, pattern matching and polymorphic type inference behave as they do there, and where the two languages agree this definition gives Elm

the meaning ML has.

Five features distinguish Elm from ML and shape the definition. First, Elm has no exceptions and pursues the aim that a program which has been accepted does not fail when it runs. No phrase raises or handles an exception, and the definition gives every phrase that has elaborated either a value or none. A phrase has no value only when its evaluation does not terminate, when it needs a top-level value while that value is itself being evaluated, or when it reaches `Debug.todo`, the one escape the language provides. Arithmetic is accordingly total: where an operation has no mathematically correct result, the definition gives a defined answer in its place, so that `3 // 0` is `0`. The basic arithmetic and comparison operations are defined in this book rather than left to a library. Second, records are row-polymorphic: a function may demand some fields of a record and accept any record that has them, and a record type may end in a row variable standing for the fields not named. Third, a type variable may be constrained to a class (`number`, `comparable`, `appendable` or `compappend`) and then stands only for the few predefined types on which arithmetic, ordering or appending is defined. The classes are fixed: a program cannot add types to them. They take the place of ML's overloaded identifiers and equality types, and equality itself is defined at every type. Fourth, the module system is deliberately simple. A module is a sequence of top-level declarations and makes some of them available to other modules by name; there are no signatures, functors or sharing. Fifth, effects are managed. A program never performs an effect. It computes values that describe effects (*commands*, *subscriptions* and *tasks*) and hands them to the platform that runs it; the platform performs them and returns their outcomes to the program as messages, one at a time. Commands, subscriptions and tasks are defined as types of the basis rather than as syntax, but they are integral to the language and have a chapter of their own. A program communicates with its host through *ports*, which it declares; `port` is a reserved word, port declarations belong to the language, and the definition states exactly which values can and cannot be sent over a port.

Standard Elm leaves out what Elm reserves to its own core packages: effect modules, declarations of new infix operators and references to kernel code. It also leaves out embedded shader blocks, which belong to a graphics extension rather than to the language.

Language Definition

The words of a programming language (`type`, `case`, `port`, `module`) name things that are not phrases: types, values, functions, interfaces, commands. These things are the *semantic objects* of the language, and they, not its syntax, carry its character; when programmers think about an Elm program they think in terms of them. For Standard Elm the semantic objects include types

with rows and classes, type schemes, environments and module interfaces on the one hand, and values, closures, records, tasks, commands and the boundary values (JSON values, with byte sequences) that cross a port on the other. A definition of the language must therefore be a definition in terms of these objects. They must be described in a notation independent of Elm, since to describe them in another programming language would only move the question to that language; ordinary mathematics serves. And they must be given enough theory (substitution, instantiation, closure, the properties on which the rules rely) for the rules to make sense, as a mathematician develops the theory of the objects a subject is about.

With the objects in hand, the definition must say what meaning each phrase of the language has. Here the keystone of our method is that a phrase never has a meaning in isolation. It always has a meaning against a background, which is itself a semantic object: a distillation of what earlier phrases have contributed, such as the types and values of the names in scope and the interfaces of the modules imported. The meaning of a phrase depends on its background as much as on the phrase. The basic assertion of the definition is therefore a sentence of the form

$$B \vdash P \Rightarrow M,$$

read "against the background B, the phrase P has the meaning M". The formal purpose of this definition is exactly to decree which sentences of this form hold, and no more. It does so by rules, each of which allows a sentence about a compound phrase to be inferred from sentences about its constituent phrases, so that the meaning of a phrase is determined by the meanings of its parts. The rules are numbered through the whole book, and each group of them is followed by Comments on the points that are not obvious.

The meaning of an Elm phrase has two parts: what can be known of it before it is evaluated, namely whether it is well formed and what its type is, and the value it evaluates to. We give the two separately. The *static semantics* infers types and interfaces, and the *dynamic semantics* infers values; each is an instance of the keystone, with its own backgrounds and its own meanings. The factoring is genuine. A value never has to be presented together with its type, and the dynamic semantics consults no type except where a port must know the type of what it sends or receives. A phrase that does not elaborate has no meaning at all, and the dynamic semantics need not consider it.

This Definition

This book follows *The Definition of Standard ML (Revised)* closely, in its structure, its notation and its style: a grid of phases against levels, a bare language with derived forms, semantic objects defined in figures, numbered inference rules with Comments, and an initial basis in appendices. Where

Elm agrees with ML we have followed the treatment of that book, so that a reader who knows it will find most of the static semantics of the Core, and much of its dynamic semantics, in familiar form. The additions are the row variables and classes of the static semantics, the much smaller static semantics of modules, a chapter on tasks, commands, subscriptions and ports, and a chapter on programs which defines how the platform runs a program as a labelled transition system whose observable behaviour is its traffic on ports. Appendix F compares the two definitions construct by construct.

Two conventions of the Definition of Standard ML are absent here. There, every sentence of the dynamic semantics carries a state, for references, and the state is threaded through the premises of every rule, so that the order of the premises is the order of evaluation; and for every rule there are further implicit rules by which an exception packet raised in a premise becomes the result of the conclusion. Standard Elm has neither state nor exceptions, so neither convention is needed. Its dynamic sentences carry no state, the order of premises carries no meaning, and no rule produces a packet.

The body of the definition does not settle everything. Where implementations legitimately differ (the range of integers, what a character is, the treatment of floating-point special values, the order in which the commands of a batch are performed), the body states only what every implementation must satisfy, through abstract parameters, and Appendix E records the choices of Elm on JavaScript and of eco, with their consequences for programs. Elsewhere the two implementations disagreed, or agreed on a behaviour that followed from their hosts rather than from the language, and the definition had to decide. Each such decision is a ruling, taken in the spirit of the language: no run-time errors, total arithmetic, and a meaning that does not depend on how a program is optimised. Some rulings declare existing behaviour non-conforming. On Elm on JavaScript, `modBy 0 x` stops the program and comparing two functions with `==` stops it too, where the definition gives `0` and a Boolean; on eco, a negative power of `1` or `-1` at Int is `0` where the definition gives `1`, or `1` or `-1` by the parity of the exponent; on both, a value containing a NaN may be equal to itself, where the definition says that it is not. Appendix E lists every such point with the ruling that settles it.

How this Definition was Produced

This definition was written with the assistance of an AI system, Claude (model Claude Opus 5.5, made by Anthropic), working within a staged, documented workflow. Under the author's direction it surveyed the sources and test suites of the two implementations, Elm 0.19.1 and eco, recording each fact with a citation to the code or test that establishes it; it proposed a ruling, with its reasons, wherever the implementations disagree or are silent; it drafted

the chapters and appendices; it audited the drafts, in independent passes, for consistency between chapters; and it encoded the dynamic semantics of the Core, of modules, of effects and of programs in the proof assistant Lean 4, namely every rule of Chapters 6, 7 and 8, the dynamic rules and the transitions of programs of Chapter 9, and the arithmetic, equality, order, Boolean, conversion, elementary-function, append and character part of APPLY, with an evaluator proved sound and complete, and a scheduler proved sound, with respect to those rules, and checked against the expected results of 702 of the test programs that accompany this book, which a program translates into the encoding from their source, each as the program of modules it is; it also encoded the static semantics, every rule of Chapters 4 and 5 and the static rules of Chapter 9, with the derived forms of Appendix A, the syntactic restrictions that can be stated on a syntax tree, and Appendix C, with a checker of derivations proved sound and a type inference proved sound and principal, by which 702 of the test programs are shown legal and 287 of the 315 that must be refused are shown not legal (the parser the encoding starts from rejects the others, or reads them in a form that has lost the error); and for each of the 702 translated programs, Lean computes from its source, through that inference, number specialisation and type erasure, the program the translator writes, and shows that this specialisation is the only one. The author set the directions the definition follows, reviewed every ruling, and is responsible for the whole; the AI system is not an author. The survey, the record of rulings, the audit reports and the mechanisation are kept with the sources of this book.

Acknowledgements

Elm was designed by Evan Czaplicki, and the language defined here is his. The form of this book is that of *The Definition of Standard ML* by Robin Milner, Mads Tofte, Robert Harper and David MacQueen; we are indebted to them for a method which has served a second language with little change.

Rupert Smith

November 2026

Chapter 1

Introduction

This book formally defines Standard Elm, the language Elm 0.19.1 as delimited in the Preface.

The method of definition is easiest to understand by considering how an implementation of Elm is naturally organised. A *program* consists of modules: a main module, and the modules it imports, directly or indirectly, with no cycle among the imports. The implementation treats the modules one at a time, each after the modules it imports. Each module is checked for the consistency of its types and for other conditions of form, and what it makes available to other modules is recorded in the top-level environment, which we call a *basis*; the modules that import it are then checked against that basis. When every module has been checked, the top-level values of the modules are computed, again each module after those it imports, and recorded in the basis. Finally the platform runs the program: it starts it, performs the commands it issues, maintains its subscriptions, and delivers to it, one at a time, the messages that result.

Two words name the parties to this last step. The *platform* is the part of an implementation that runs a program, performs its effects and exchanges values with the host. The *host* is the environment outside the program: a JavaScript page, an embedding application, or an operating system. The host sends values to the incoming ports of a program and receives those of its outgoing ports, through the platform. The later chapters use both words with these meanings only.

There are thus three phases: *parsing*, *elaboration* and *evaluation*. Parsing determines the grammatical form of a module. Elaboration, the *static* phase, determines whether its types are consistent and whether it meets the other conditions of form; a module that does both *elaborates*. Elaboration records type and interface information in the basis. Evaluation, the *dynamic* phase, determines the values of its declarations and records them in the basis. Corresponding to the phases, the definition divides into grammatical rules, elaboration rules and evaluation rules, and the basis divides into the *static*

basis and the *dynamic basis*: a top-level variable, for example, is associated with a type scheme in the static basis and with a value in the dynamic basis. An implementation need not divide the basis so; for the definition the division makes each phase simpler to present. Every module of a program is elaborated before any is evaluated, and a program that does not elaborate is not evaluated at all.

A further factoring follows the structure of the language. Standard Elm has a lower level, the *Core*, of expressions, patterns, types and `let` definitions: everything the right-hand side of a single definition can contain. Above it is *Modules*: module headers, exposing lists, imports, and the top-level declarations of a module, namely value definitions, custom types, type aliases and ports. The top level, *Programs*, defines what a program is, the type its `main` must have, and how the platform runs it; all three phases of Programs are treated in one chapter. Commands, subscriptions and tasks add no syntax, being types of the basis, but their values and the behaviour of ports need a semantics of their own. We give it in a chapter on *Effects*, which is purely dynamic; the static question of which types may cross a port belongs to the static semantics of Modules. The scheme of the chapters is therefore as follows:

	Core	*Modules*	*Effects*	*Programs*
Syntax	Chapter 2	Chapter 3	none	Chapter 9
Static Semantics	Chapter 4	Chapter 5	Section 5.4	
Dynamic Semantics	Chapter 6	Chapter 7	Chapter 8	

The Core and Modules provide many forms of phrase for the programmer's convenience. Many of them are *derived forms*, whose meaning is given by translation into the remaining phrases, which we call the *bare* language: infixed operators, conditionals, `&&` and `||`, negation, list expressions and patterns, pattern rows written as a label alone, field selection and field accessors, functions of several arguments, and the grouping of the definitions of a `let` or of a module body into dependency groups. Each semantics chapter treats only the bare language; the derived forms are given in Appendix A. The full grammar of the language, bare and derived together, is collected in Appendix B.

Appendices C and D give the *initial basis*, divided into its static and dynamic parts: the predefined types, constructors and values that the rules and the default imports need, with the basic arithmetic and comparison operations defined there precisely. These operations are total, so that no evaluation ends in a run-time error. Where implementations legitimately differ, the body of the definition states only what each must satisfy, and Appendix E records the choices of two implementations and every known point on which they do not conform. Appendix F compares this definition with the Definition of Standard ML.

The semantics is presented as a set of rules allowing *sentences* of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

to be inferred, where A is usually a basis or a context (static) or an environment (dynamic), and A' is a semantic object, usually a type in the static semantics and a value in the dynamic semantics. Such a sentence is read "against the background A, *phrase* elaborates (or evaluates) to A'". The rules are formal; the semantic objects are the subject of a small mathematical theory, given in the chapters that introduce them, and the properties of the objects on which the rules rely are stated there without proof.

Some notation is defined once and used throughout. Finite subsets, finite maps and the modification $f + g$ of one finite map by another are defined in Section 4.2, together with the notation $x^{(k)}$ for a sequence of length k; projection, injection and modification of tuples of objects in Section 4.3; and the first and second options $\langle\ \rangle$ and $\langle\langle\ \rangle\rangle$ of the rules in Section 4.9. The conventions of the grammar figures are stated with the grammar of the Core (Chapter 2). Throughout, a metavariable decorated with primes or subscripts ranges over the same class as the undecorated one.

Chapter 2

Syntax of the Core

2.1 Reserved Words

The following are the *reserved words* of Standard Elm. They may not be used as identifiers or as operators.

```
as    case    else  exposing  if    import  in
let   module  of    port      then  type    where
( )   [ ]     { }   , .       : =   | ->    \ _
```

The words `exposing`, `import`, `module`, `port` and `type` are used by Modules (Chapter 3). The word `where` is used by no phrase of Standard Elm: it belongs to the headers of effect modules, which are kernel-only syntax (Section 2.7). The words `alias`, `infix`, `left`, `right`, `non`, `effect`, `command` and `subscription` are not reserved; `alias` serves as a terminal only directly after `type` (Chapter 3), and elsewhere each of them is an identifier.

2.2 Special Constants

Source text is a sequence of Unicode characters. A *decimal digit* is one of `0`, ..., `9`; a *hexadecimal digit* is a decimal digit or one of `a`, ..., `f`, `A`, ..., `F`.

An *integer constant (in decimal notation)* is a non-empty sequence of decimal digits that does not begin with `0` unless it is `0` itself. An *integer constant (in hexadecimal notation)* is `0x` followed by a non-empty sequence of hexadecimal digits. No integer constant carries a sign: a negated integer is a derived form (Section 2.5). An integer constant denotes a non-negative integer; its value as an `Int` is that integer normalised into the implementation's Int range by norm, the normalisation of Int arithmetic, so that every integer constant has a value in IntRange (Appendix E). Examples: `0  42  0x1F  0xffffffff`. Non-examples: `007  0X1F  1_000  0b101`.

A *float constant* is an integer constant in decimal notation followed by a fraction, by an exponent, or by a fraction and then an exponent. A

`\n`	line feed (U+000A)
`\r`	carriage return (U+000D)
`\t`	horizontal tab (U+0009)
`\"`	`"`
`\'`	`'`
`\\`	`\`
`\u{`*h*`}`	the character with code point *h*, read in hexadecimal

Figure 1: Escape Sequences

fraction is `.` followed by a non-empty sequence of decimal digits; an *exponent* is `e` or `E`, then optionally `+` or `-`, then a non-empty sequence of decimal digits. The constant `0` may be followed by a fraction but not directly by an exponent. A float constant denotes the rational number that it writes in decimal notation. Examples: `0.5` `3.14` `6.02e23` `1E-3` `0.0e1`. Non-examples: `.5` `1.` `1.e5` `0e5`.

A *character constant* is a single quote `'`, then either one character other than `'`, `\` and line feed, or one escape sequence, then a single quote. It denotes the character it contains, or the character its escape sequence denotes. Thus `''` and `'ab'` are not constants.

A *string constant (single-line)* is a double quote `"`, then a sequence of zero or more characters other than `"`, `\` and line feed, and escape sequences, then a double quote. Three double quotes `"""` always begin a multi-line string constant, so `""` is the empty string only when no third `"` follows it. A *string constant (multi-line)* is `"""`, then a sequence of characters other than `\`, and escape sequences, that does not contain `"""`, then `"""`. The first `"""` after the opening one closes the constant; any further `"` immediately after it begins a new item. Line feeds and carriage returns inside a multi-line string constant are characters of the string, and so is a line feed directly after the opening `"""`. A string constant denotes the sequence of characters it contains, each escape sequence standing for the character it denotes.

Each escape sequence begins with `\` and stands for one character. The escape sequences are shown in Figure 1, subject to this restriction:

- In `\u{`*h*`}`, *h* is a sequence of four, five or six hexadecimal digits whose value is a Unicode scalar value, that is, at most `10FFFF` and outside the surrogate range `D800` to `DFFF`. No other sequence beginning with `\` is an escape sequence.

A tab character may occur, unescaped, in a character or string constant. How a character is represented as a value is an implementation parameter (Appendix E).

We denote by SCon the class of *special constants*, that is, the integer, float, character and string constants; we use *scon* to range over SCon.

2.3 Comments

A *line comment* is `--` and the characters after it up to the end of the line. A *block comment* is any character sequence within comment brackets `{-` `-}` in which comment brackets are properly nested. No space is allowed between the two characters of a comment bracket. A program in which a `{-` is unmatched is not a phrase of the language. A tab character may occur in a line comment but not in a block comment.

A comment bracket `{-` directly followed by `|` begins a *doc comment*, which extends, with comment brackets nested as in a block comment, to its matching `-}`. A doc comment is not a comment: it is an item of the grammar of Modules, which admits it only after a module header and before a top-level declaration (Chapter 3). No doc comment occurs in a Core phrase.

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 2. We use *var*, *tyvar* and so on to range over Var, TyVar, etc. For each class marked "long" there is a class of *long identifiers*: LongVar, LongCon and LongTyCon, over which *longvar*, *longcon* and *longtycon* range. Their syntax is

longvar	::=	*var*	identifier
		modid . *var*	qualified identifier
longcon	::=	*con*	identifier
		modid . *con*	qualified identifier
longtycon	::=	*tycon*	identifier
		modid . *tycon*	qualified identifier

with no space or comment on either side of each `.`. Throughout this document the term "identifier", occurring without an adjective, refers to an identifier that is not qualified.

A *letter* is a character of one of the Unicode general categories Lu, Ll, Lt, Lm and Lo; the version of the Unicode standard that assigns these categories is an implementation parameter (Appendix E). The identifiers are formed as this restriction states:

- A *lower identifier* is a letter of category Ll (which includes `a`, ..., `z`) followed by a sequence of zero or more *inner characters*: letters, decimal digits and underbars `_`. An *upper identifier* is a letter of category Lu or Lt (which include `A`, ..., `Z`) followed by a sequence of zero or more inner characters. A letter of category Lm or Lo may continue an identifier but not begin one, and no identifier begins with `_`. In either case reserved words are excluded, and no component of a long identifier may be a reserved word: thus `List.if` is not a long identifier.

An *operator* is a non-empty sequence of the *symbols*

var	Var	value variables	long
con	Con	value constructors	long
tyvar	TyVar	type variables	
tycon	TyCon	type constructors	long
lab	Lab	record labels	
modid	ModId	module identifiers	
op	Op	operators	

Figure 2: Identifiers

```
+  -  *  /  =  .  <  >  :  &  |  ^  ?  %  !
```

that is not a reserved word. This means that `=`, `:`, `|` and `->` are not operators, but `==`, `::`, `||` and `|>` are. Operators are never qualified.

A variable *var*, a type variable *tyvar* and a record label *lab* are lower identifiers. A constructor *con* and a type constructor *tycon* are upper identifiers. A module identifier *modid* is a non-empty sequence of upper identifiers separated by `.`, with no space or comment around each `.`. A type variable whose name begins with `number`, `comparable`, `appendable` or `compappend` is constrained to a class of types (Section 4.4).

The classes Var, TyVar and Lab share the lower identifiers, and Con, TyCon and ModId the upper identifiers. The syntax class of an occurrence of an identifier in a Core phrase is determined thus:

1. An upper identifier immediately followed by `.` is a component of a module identifier. The following rules assume that all occurrences of module identifiers have been removed.

2. Within a type expression, a lower identifier is a type variable and an upper identifier is a type constructor.

3. At the start of a component of a record expression, record update, record type or record pattern, and immediately after the `.` of a field selection or field accessor (Appendix A), a lower identifier is a record label. A record label that forms a whole component of a record pattern, the derived pattern row of Appendix A, is also a variable, bound by that pattern.

4. Elsewhere, a lower identifier is a variable and an upper identifier is a constructor.

For the remainder of this document we shall therefore assume that the classes are all disjoint.

2.5 Lexical Analysis

Each item of lexical analysis is either a reserved word, a special constant, a long identifier, an operator, or a doc comment. Comments and the *whitespace* characters (space, line feed and carriage return) separate items and are otherwise ignored, except that the layout rule depends on the column at which each item begins (Section 2.6) and that the constructs listed below depend on whether two items are separated. A tab character may occur only within a line comment or within a character or string constant. Where the remaining text begins with `--` or `{-`, a comment or doc comment is taken. Otherwise, at each stage the longest next item is taken. Thus `types` is an identifier, not `type` followed by `s`; within a run of symbols `--` is part of an operator and begins no comment, so `a*-b` contains the single operator `*-`, and `2^-1` the operator `^-`. Items are also subject to these restrictions:

- A special constant, and the reserved word `_`, may not be followed immediately by an inner character, and the digits of an integer constant may be followed immediately by `.` only as part of a fraction; so `12abc`, `1.x` and `_x` are not sequences of items.

- The following constructs require two items to be *adjacent*, that is, to have no whitespace or comment between them:

  - the parentheses of the unit `()`, and the operator and parentheses of an operator value `(`*op*`)`;
  - the `.` of a long identifier and the items on either side of it;
  - the `.` of a field selection *atexp* `.` *lab* and the items on either side of it, and the `.` of a field accessor `.`*lab* and the label after it. A `.` adjacent to a following label is the `.` of a field selection only if it is adjacent to the item before it and that item ends an atomic expression that may precede a field selection (Section 2.10); otherwise it begins a field accessor (Appendix A). In particular a `.` directly after a field accessor begins a field accessor;
  - the negation symbol and the atomic expression it negates.

An item `-` is a *negation symbol* when it is adjacent to a following atomic expression and either it begins an operand of an expression, or it follows an operand from which it is separated by whitespace or a comment. A negation symbol negates the atomic expression after it, and no more; in the second case the negated expression is an argument of an application. Every other `-` is the operator `-`. Thus `f -x` applies `f` to the negation of `x`, while `a - b` and `a-b` are subtractions; `-f x` applies the negation of `f` to `x`; and `(-)` is the operator value of subtraction. Negation is a derived form (Appendix A).

2.6 Layout

The phrase structure of Elm depends on indentation. Source text is divided into lines by line feeds. The *column* of an item is one more than the number of characters that precede it on its line, where a carriage return counts as no character. Each item is read under an *indentation*, a natural number i. An item whose column is c is

- *indented* if $c > i$ and $c > 1$;
- *aligned* if $c = i$;
- *at the margin* if $c = 1$.

These tests concern only the column at which an item begins: where the item before it ended, for instance on a later line than it began, is irrelevant. The *layout rule* is this restriction:

- Every item of a phrase is indented, except the items that the constructs below require to be aligned or at the margin.

Only these constructs change the indentation; elsewhere an item is read under the indentation of the phrase that contains it.

1. *Top-level declarations* (Chapter 3). The first item of each top-level declaration is at the margin, and its remaining items are read under the indentation 0; so each of them lies in column 2 or beyond.
2. *Let expressions.* Let k be the column of the reserved word `let`. The item after `let` has a column d with $d > k$ and $d > 1$, and d is the indentation under which the definitions of the `let` are read: the first item of each later definition, and of each annotation, is aligned, and every other item of the definitions is indented. The reserved word `in` and the body after it are read under the indentation in force at `let`.
3. *Case expressions.* The items of `case` *exp* `of` and the first item of the first match rule are read under the indentation in force at `case`. The column p of that first item is the indentation under which the match is read: the first item of each later match rule is aligned, and every other item of the match is indented.

An annotation *var* `:` *ty* is followed directly by a definition of the same *var*, whose first item has the column of the annotation's first item: at the margin at top level, aligned in a `let`. Each construct extends as far as possible: it takes in every following item that passes the test that applies to it and continues the phrase, and it ends before the first item that does not. Thus two definitions of one `let`, or two match rules of one `case`, never begin on the same line, while `let x = 1 in x` is a phrase on one line.

Precedence	Left	Right	Non
9	`<<`	`>>`	
8		`^`	
7	`* / //`		
6	`+ -`		
5		`++ ::`	
4			`== /= < > <= >=`
3		`&&`	
2		`||`	
0	`|>`	`<|`	

Figure 3: Fixities of the Operators of the Initial Basis

2.7 Infixed Operators

An operator *op* has a *fixity*: a *precedence*, which is one of the digits `0`, ..., `9`, and an *associativity*, which is *left*, *right* or *non*. Operators are declared, with their fixities, only by kernel-only syntax. Kernel-only syntax, namely operator declarations, effect module headers and references to `Elm.Kernel` modules, lies outside Standard Elm. No program can therefore declare an operator, and the operators are those of the initial basis, the same in every program. Their fixities are shown in Figure 3; the operator table of the initial basis gives the meaning of each, its type scheme in Appendix C and its value in Appendix D. Operators of library modules beyond the initial basis are outside this definition.

An expression of the form $exp_0\ op_1\ exp_1\ \cdots\ op_n\ exp_n$ $(n \geq 1)$, in which each op_j is an operator and each operand exp_j is an application or an atomic expression, possibly negated, except that the last operand may be any expression, is an *operator chain*. Application binds more tightly than any operator. A chain is grouped so that an operator of higher precedence binds more tightly than one of lower precedence, subject to this restriction:

- If op_j and op_l $(j < l)$ of a chain have equal precedence and no operator of lower precedence stands between them, then either both associate to the left and the chain groups to the left, or both associate to the right and it groups to the right; otherwise the chain is not a phrase of the language.

Precisely, a chain is the infixed application $exp'\ op_j\ exp''$ in which op_j has the least precedence in the chain and is, among the operators of that precedence, the rightmost if they associate to the left and the leftmost if they associate to the right; exp' and exp'' are the chains, or single operands, to the left and right of op_j, grouped by the same rule. For example,

`a |> f |> g`	parses as	`(a |> f) |> g`
`f <| g <| a`	parses as	`f <| (g <| a)`
`x :: ys ++ zs`	parses as	`x :: (ys ++ zs)`
`a < b == c`	is illegal	
`f << g >> h`	is illegal	

An operator value (*op*) is an atomic expression that denotes the operator *op* as a value; it is the only way to use an operator other than in a chain. There are no sections, and no operator is qualified. The infixed application exp_1 *op* exp_2 is a derived form (Appendix A); so are the infixed forms of `&&` and `||`, which evaluate their second operand only when needed, whereas the operator values `(&&)` and `(||)` are ordinary functions. In the semantic rules every chain has been grouped and every infixed application replaced by its equivalent form, since fixity affects only parsing.

2.8 Derived Forms

Many syntactic forms of Elm have a meaning that is given by translation into a smaller set of forms, called the *bare language*. Among them are infixed application, negation, conditional expressions, list expressions with elements, list patterns with elements, pattern rows written as a label alone, field selection and field accessors, lambda expressions and function bindings with parameters beyond the first, and `let` expressions with several definitions. These derived forms, and their equivalent forms in the bare language, are given in Appendix A. Appendix B collects the bare and derived syntax into one grammar.

2.9 Grammar

The phrase classes for the Core are shown in Figure 4. We use the variable *atexp* to range over AtExp, etc. The grammatical rules for the Core are shown in Figures 5, 6 and 7.

The following conventions are adopted in presenting the grammatical rules, and in their interpretation:

- The brackets ⟨ ⟩ enclose optional phrases.
- Alternative forms for each phrase class are in order of decreasing precedence. L (resp. R) means left (resp. right) association.
- The phrases `\` *atpat* `->` *exp*, `case` *exp* `of` *match* and `let` *dec* `in` *exp* extend as far right as the layout rule allows (Section 2.6). Such a phrase is written in parentheses when it is the argument of an application, or an operand of an operator chain other than the last.

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
MRule	match rules
Dec	declarations
ValBind	value bindings
AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns
Ty	type expressions
TyRow	type-expression rows

Figure 4: Core Phrase Classes

- In a type construction *longtycon* $ty_1 \cdots ty_n$ with $n \geq 1$, each argument ty_j is a type variable, a type construction without arguments, a record type expression, a unit or tuple type expression, or a parenthesised type expression; thus `Maybe List a` gives `Maybe` two arguments. Type construction binds more tightly than `->`.

- In patterns, a constructed pattern binds more tightly than `::`, and `::` more tightly than `as`; so `x :: xs as l` is a layered pattern whose pattern is `x :: xs`. The `::` of a pattern always denotes the list constructor of the initial basis. There is no negated constant in a pattern.

- The separator `;` between match rules and the connective `and` between value bindings are abstract syntax: neither is an item of source text, and `and` is not a reserved word. In source text the match rules of a match, and the definitions of a `let`, are separated by layout (Section 2.6). Likewise the annotated value binding *var* `:` *ty* `=` *exp* is written in source text as the annotation *var* `:` *ty* followed by the definition *var* `=` *exp*.

- In the bare language the declaration of a `let` is a single value binding whose members form one group; a `let` with several definitions is a derived form. A value binding *atpat* `=` *exp* in which *atpat* is a variable is read by the first alternative for *valbind*.

- Tuples have exactly two or three components.

atexp	::=	*scon*	special constant
		longvar	value variable
		longcon	value constructor
		`(` *op* `)`	operator value
		`()`	unit
		`(` exp_1 `,` exp_2 ⟨`,` exp_3⟩ `)`	tuple
		`[ ]`	empty list
		`{` ⟨*exprow*⟩ `}`	record
		`{` *var* `|` *exprow* `}`	record update
		`let` *dec* `in` *exp*	local declaration
		`(` *exp* `)`	parentheses
exprow	::=	*lab* `=` *exp* ⟨`,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		`\` *atpat* `->` *exp*	function
		`case` *exp* `of` *match*	case analysis
match	::=	*mrule* ⟨`;` *match*⟩	match
mrule	::=	*pat* `->` *exp*	match rule
dec	::=	*valbind*	value declaration
valbind	::=	*var* ⟨`:` *ty*⟩ `=` *exp* ⟨`and` *valbind*⟩	variable binding
		atpat `=` *exp*	destructuring binding

Figure 5: Grammar: Expressions, Matches, Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		var	variable
		longcon	constructor
		`()`	unit
		`(` pat_1 `,` pat_2 ⟨`,` pat_3⟩ `)`	tuple
		`[ ]`	empty list
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	parentheses
patrow	::=	*lab* `=` *var* ⟨`,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
		longcon $atpat_1 \cdots atpat_n$	constructed pattern ($n \geq 1$)
		pat_1 `::` pat_2	list construction (R)
		pat `as` *var*	layered

Figure 6: Grammar: Patterns

ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type expression
		`{` *tyvar* `|` *tyrow* `}`	extensible record type expression
		`()`	unit type expression
		`(` ty_1 `,` ty_2 ⟨`,` ty_3⟩ `)`	tuple type expression
		longtycon $ty_1 \cdots ty_n$	type construction ($n \geq 0$)
		ty `->` *ty′*	function type expression (R)
		`(` *ty* `)`	parentheses
tyrow	::=	*lab* `:` *ty* ⟨`,` *tyrow*⟩	type-expression row

Figure 7: Grammar: Type Expressions

2.10 Syntactic Restrictions

The restrictions below are conditions on a program as it is written, before any derived form is replaced by its equivalent form; where a restriction names a derived form, the form is that of Appendix A. A program that breaks one of them is not a phrase of the language.

Records.

- No expression row, pattern row or type-expression row may contain the same *lab* twice. This applies to the expression row of a record update.
- No pattern row of a program has the form *lab* `=` *var*: a program writes each component of a record pattern as a label alone, the derived form of Appendix A, whose equivalent form *lab* `=` *var* binds the variable spelt as the label. The bare form exists so that number specialisation (Chapter 6) can give the variable of a row a new name while keeping its label.
- In a record update `{` *var* `|` *exprow* `}` the item before `|` is a variable that is not qualified, and at least one expression row follows `|`; the grammar gives this shape, and an extensible record type expression likewise has at least one row.
- In a field selection *atexp* `.` *lab* (Appendix A), *atexp* is a *longvar*, a record, a record update, a field selection, or a phrase that begins with `(`: a parenthesised expression, a tuple, the unit or an operator value. A field accessor is never followed by a field selection: in `.`lab_1`.`lab_2 the second `.` begins a second field accessor, so the phrase is the application of `.`lab_1 to `.`lab_2. Likewise `"s".x` is the application of a string constant to the field accessor `.x`.

Patterns and bound variables. The *variable of a pattern row* is the *var* of a row *lab* `=` *var* and, for a row written as a label alone (Appendix A), the variable spelt as its label. The variables *bound* by a pattern are the variables

that occur in it outside its pattern rows, the variables of its pattern rows, and the variables after its occurrences of `as`. the following: the parameters of a lambda expression, taken together; the parameters of a function binding, taken together; the pattern of a match rule; the pattern of a destructuring binding; and the definitions of one `let`, taken together, which bind the variables they define and the variables their destructuring patterns bind. The *scope* of a binding site is the body of the lambda expression, the right-hand side of the function binding, the expression of the match rule, or, for a destructuring binding and for the definitions of a `let`, the definitions of that `let` and its body.

- No binding site may bind the same *var* twice.
- No binding site may bind a *var* that is already bound by a binding site in whose scope it lies. Constructors, type constructors and type variables are not subject to this restriction.
- No float constant may occur in a pattern.

The number of arguments in a constructed pattern is not restricted here: a constructed pattern with too few or too many arguments for its constructor does not elaborate (Chapter 4).

Value bindings.

- In source text each annotation *var* `:` *ty* is followed directly by a definition of the same *var*, laid out as Section 2.6 requires.
- In a `let`, a definition *refers to* a definition (possibly itself) if a variable that the second binds occurs in the first. Every definition that lies on a cycle of this relation is a function binding *var* $atpat_1 \cdots atpat_n$ `=` *exp* with $n \geq 1$; in particular, no destructuring binding and no definition *var* `=` *exp* without parameters lies on a cycle, even when *exp* is a lambda expression. The corresponding restriction for top-level declarations is given in Chapter 3.

There is no limit on the size of a phrase: the number of parameters, components, fields or constructors is unbounded, and the limits of an implementation are listed in Appendix E.

Chapter 3

Syntax of Modules

For Modules there are further reserved words and one further symbol, but no further special constants or identifier classes; comments and lexical analysis are as for the Core. Every module begins with a header, which is followed by a sequence of imports and a sequence of top-level declarations; a module makes its top-level names available to other modules by name only. The derived forms for modules appear in Appendix A (Chapter A). The `infix` declarations, `effect module` headers and imports of kernel modules that Elm admits only in its own core packages are not part of Standard Elm.

3.1 Reserved Words

The following reserved words are used in Modules.

```
as    exposing    import    module    port    type    ..
```

The words are among the reserved words of the Core (Chapter 2), so none of them may be used as an identifier anywhere in a program; `as` is also used in patterns. The symbol `..` occurs only in exposing lists, where white space may separate it from the parentheses that enclose it; since it is reserved, it is not an operator.

The word `alias` is not reserved. It has its special meaning only directly after the reserved word `type`, where a lower-case word cannot otherwise stand, since every declared *tycon* begins with an upper-case letter; elsewhere `alias` is a *var*.

3.2 Identifiers

There are no further identifier classes. The class ModId of module identifiers is introduced with the Core, because a module identifier is the qualifier of a long identifier *longvar*, *longcon* or *longtycon*. A *modid* is a sequence of one or more upper-case identifiers joined by `.` with no white space between them.

At this level a *modid* has two further uses: it names a module, in a header and in an import, and it names an alias for an imported module, after `as` in an import.

The qualifiers that may be used in a module, and the module each denotes, are determined by the module's imports: an import `import` *modid* `as` *modid′* provides the qualifier *modid′* and not *modid*, and an import without `as` provides its own *modid* (Chapter 5). A qualifier never denotes the module in which it occurs. The identifiers declared at this level (the *var* of a value or a port, the *tycon* of a custom type or an alias, the *con* of a constructor) are never qualified where they are declared. As in the Core, the class of each identifier occurrence is determined by the grammatical rules which follow.

3.3 Infixed Operators

Standard Elm has no fixity declarations. The operators that may be used in infix position in a module are those of the initial basis (Section 2.7). The default imports (Section 3.6) expose every one of them to every module, and the meaning of an operator is fixed by the initial basis, whatever the module imports. Each has the precedence and associativity given for it in Section 2.7, and infixed expressions are resolved as described there.

3.4 Layout

The layout rule of Section 2.6 governs the phrases within a top-level declaration. At the top level of a module it takes the following form.

- The header, each import and each top-level declaration begins in column 1.

- Every other token of a header, an import or a top-level declaration lies in a column greater than 1. Hence a token in column 1 ends the header, import or declaration before it.

- The annotation *var* : *ty* of a top-level value begins in column 1, and the definition it annotates begins in column 1 on a later line, with only white space and comments between them; the definition repeats *var*.

- A doc comment may stand directly after the header and directly before a top-level declaration, and nowhere else at the top level; the token that follows a doc comment begins in column 1. Doc comments have no meaning and are omitted from the grammar.

3.5 Grammar for Modules

The phrase classes for Modules are shown in Figure 8. We use the variable *module* to range over Module, *item* to range over ExpItem, and the lower-case name of each other class to range over that class. The conventions adopted in presenting the grammatical rules for Modules are the same as for the Core. The grammatical rules are shown in Figures 9 and 10.

Module	modules
Header	module headers
Imports	import lists
ExpList	exposing lists
ExpItem	exposing items
TopDec	top-level declarations
DatBind	custom-type bindings
ConBind	constructor bindings
TypBind	type-alias bindings
TyVarSeq	type-variable sequences

Figure 8: Modules Phrase Classes

The grammar is that of the bare language. A module as written is brought to this form by the derived forms of Appendix A, which gather the custom types of a module into a single custom-type binding, and arrange its aliases and value definitions in dependency order, the values in groups. The header is written in every module and no derived form supplies it. Two symbols of the bare grammar are never written in a program. The symbol `and` joins the members of one group: the custom types of a *datbind*, as it joins the members of a Core *valbind*. The symbol `;` separates two top-level declarations, and stands for the boundary between declarations that the layout of Section 3.4 marks by column 1. Neither symbol is reserved; in particular `and` is an ordinary *var*.

A top-level value declaration is a *valbind* of the Core, whose bindings carry their annotations. A port declaration consists of a *var* and its type, and has no definition; which types a port may have is decided by elaboration (Section 5.4).

In a constructor binding *con* $ty_1 \cdots ty_n$ each ty_i is a type variable, a *longtycon* without arguments, or a type enclosed in parentheses or braces, as for the arguments of a type constructor in the Core (Chapter 2); so a function type, or a type constructor applied to arguments, is parenthesised when it is the argument of a constructor. The constructor bindings of a *conbind* are separated by `|`, which binds less tightly than any type. In a type-alias binding and a port declaration the type extends to the end of the declaration.

module	::=	*header imports topdec*	module
header	::=	`module` *modid* `exposing` *explist*	plain
		`port module` *modid* `exposing` *explist*	port module
imports	::=	`import` *modid* ⟨`as` *modid′*⟩ ⟨`exposing` *explist*⟩ *imports*	import
			empty
explist	::=	`(..)`	everything
		`(` $item_1$ `,` $\cdots$ `,` $item_n$ `)`	explicit ($n \geq 1$)
item	::=	*var*	value
		`(`*op*`)`	operator
		tycon	type
		tycon `(..)`	open type

Figure 9: Grammar: Modules, Headers, Imports and Exposing Lists

topdec	::=	*valbind*	value
		`type` *datbind*	custom type
		`type alias` *typbind*	type alias
		`port` *var* `:` *ty*	port
		$topdec_1$ `;` $topdec_2$	sequential
datbind	::=	*tycon tyvarseq* `=` *conbind* ⟨`and` *datbind*⟩	custom-type binding
conbind	::=	*con* ty_1 $\cdots$ ty_n ⟨`|` *conbind*⟩	constructor binding ($n \geq 0$)
typbind	::=	*tycon tyvarseq* `=` *ty*	type-alias binding
tyvarseq	::=	$tyvar_1$ $\cdots$ $tyvar_k$	type parameters ($k \geq 0$)

Figure 10: Grammar: Top-level Declarations and Bindings

3.6 Default Imports

The imports of every module, as elaborated (Chapter 5) and evaluated (Chapter 7), are the default imports of Figure 11 followed by the imports written in the module. Since the default imports come first, an import written in the module may replace a type that a default import exposes.

```
import Basics exposing (..)
import Debug
import List exposing ((::))
import Maybe exposing (Maybe(..))
import Result exposing (Result(..))
import String exposing (String)
import Char exposing (Char)
import Tuple
import Platform exposing (Program)
import Platform.Cmd as Cmd exposing (Cmd)
import Platform.Sub as Sub exposing (Sub)
```

Figure 11: Default Imports

3.7 Syntactic Restrictions

The following restrictions are conditions on a module as a phrase, except those of the last item, which are conditions on a program. Each is decidable without type information. In them, the *declarations* of a *topdec* are the value declarations, custom type declarations, type alias declarations and port declarations of which it is the sequence.

- In a module, the *topdec* contains a port declaration only if the header is a port module header, and the *topdec* of a module with a port module header contains at least one port declaration.

- Every module begins with a header, `module` *modid* `exposing` *explist* or `port module` *modid* `exposing` *explist*. A text that does not begin with a header is not a module of Standard Elm.

- In an import, the *modid* after `as` consists of a single upper-case identifier.

- The qualifier `Basics` is provided only by an import of the module `Basics`: an import `import` *modid* `as Basics` requires *modid* to be `Basics`. Since no module of a program has the identifier of a module of the initial basis (the last item), the module `Basics` is always that of the initial basis, and the long identifiers `Basics.True`, `Basics.False`

and `Basics.negate` that derived forms introduce (Appendix A) always denote its bindings.

- An item `(`*op*`)` occurs only in the exposing list of an import: a module declares no operator, so it has none to expose.

- In the exposing list of a header, each item *var* is bound by a value declaration or a port declaration of the module's *topdec*; each item *tycon* is the *tycon* of a *datbind* or a *typbind* of the *topdec*; and each item *tycon* `(..)` is the *tycon* of a *datbind*, never of a *typbind*. In particular a constructor cannot be exposed by name: an item *tycon* names a type, and the constructors of a custom type are exposed only by *tycon* `(..)` or by `(..)`.

- No two items of the exposing list of a header may have the same identifier; thus *tycon* and *tycon* `(..)` do not both occur.

- No value declaration of a *topdec* may contain a binding other than one of the form *var* ⟨`:` *ty*⟩ `=` *exp*; there is no pattern binding at the top level.

- No *topdec* may bind the same *var* twice by its value declarations and port declarations together.

- No *topdec* may declare the same *tycon* twice: custom types and aliases share one name space.

- No *topdec* may declare the same *con* twice. This counts the *con* of each constructor binding of every *datbind*, and the *tycon* of each *record alias*, that is, each *typbind tycon tyvarseq* `=` *ty* whose *ty* is, apart from enclosing parentheses, a record type `{` ⟨*tyrow*⟩ `}` without an extension variable; a record alias also declares a constructor of that name (Chapter 5).

- No variable bound within an expression of a *topdec* (a parameter of a lambda, a variable of a pattern, or a variable bound in a `let`) may be a *var* bound by a value declaration or a port declaration of that *topdec*, wherever the declaration stands. This extends to top-level names the restriction on shadowing of Section 2.10. Variables declared by imported modules may be bound again.

- No *tyvarseq* may contain the same *tyvar* twice.

- In a custom-type binding *tycon tyvarseq* `=` *conbind*, every *tyvar* occurring in *conbind*, including the extension variable of a record type, occurs in *tyvarseq*. A *tyvar* of *tyvarseq* need not occur in *conbind*.

- In a type-alias binding *tycon tyvarseq* `=` *ty*, the *tyvarseq* contains exactly the type variables occurring in *ty*, including extension variables of record types: an alias has no unused parameter.

- No alias may be recursive. For the type-alias bindings of a *topdec*, say that *tycon refers to tycon′* if *tycon′* is declared by a type-alias binding of the *topdec* and occurs, unqualified, in the type of the binding of *tycon*; this relation has no cycle. The *tycon* of a custom type is not related by it, so a cycle through a custom type is not excluded.

- An occurrence of a variable in an expression *exp* is *delayed* if it lies within a lambda expression in *exp*, and *direct* otherwise. For a value declaration *valbind* of a *topdec*, say that $var \to var'$ if *valbind* contains a binding *var* ⟨`:` *ty*⟩ `=` *exp* and *var′* is bound by *valbind* and has a direct occurrence in *exp*. No value declaration of a *topdec* may have a cycle $var_1 \to \cdots \to var_n \to var_1$ ($n \geq 1$). This takes the place, for the value declarations of a *topdec*, of the restriction on recursive groups of Section 2.10. Since a binding with parameters is a derived form whose equivalent form is a lambda expression (Appendix A), every occurrence in its body is delayed, and a rejected cycle passes only through values without parameters. A delayed occurrence used during the evaluation of the very value it refers to passes this restriction; no evaluation rule applies to it (Chapter 7).

- The following restrictions concern a program, and Chapter 9 imposes them. The *var* of every port declaration of every module of a program is distinct from the *var* of every other port declaration of the program and from the name of every manager of the platform (Chapter 8). No module of a program has the identifier of a module of the initial basis. In the main module, `main` does not occur free in the expression of its own declaration, so `main` is not recursive, even through a delayed occurrence.

For an import, the conditions corresponding to those on the exposing list of a header depend on what the imported module exposes: an item *tycon* `(..)` that names an alias, an item *tycon* that names a constructor rather than a type, and an item that the imported module does not expose are rejected by the elaboration of imports (Chapter 5).

Chapter 4

Static Semantics for the Core

This chapter defines the semantic objects of the static semantics of the Core and the rules by which Core phrases elaborate. The simple objects are type variables, each with a class, row variables, type names and identifier status descriptors. From them we build types, record types, type functions, type schemes, environments and contexts, and on these we define substitution, instantiation and closure. The rules infer sentences $C \vdash \mathit{phrase} \Rightarrow A$, in which C is a context and A is a type, a row, a value environment, or a value environment paired with a type. A Core phrase is legal when it satisfies the syntactic restrictions of Chapter 2, elaborates by the rules of Section 4.9, and its principal derivation (Section 4.10) meets the further restrictions of Section 4.11.

4.1 Simple Objects

All semantic objects of the static semantics are built from identifiers (Chapter 2), labels among them, and four further kinds of simple object: type variables, row variables, type names and identifier status descriptors, together with the classes that type variables carry. Type names are the values taken by type constructors; they are distinct from type variables and from type constructors. The simple object classes, and the metavariables ranging over them, are shown in Figure 12. We include TyVar in the figure to show that α ranges over it.

α or $\mathit{tyvar} \in$	TyVar	type variables
$\epsilon \in$	RowVar	row variables
$t \in$	TyName	type names
$\kappa \in$	Class	classes
$\mathit{is} \in$	IdStatus $= \{\mathrm{c}, \mathrm{v}\}$	identifier status descriptors

Figure 12: Simple Semantic Objects

The classes are

$$\text{Class} = \{\text{any}, \text{number}, \text{comparable}, \text{appendable}, \text{compappend}\}.$$

Each $\alpha \in \text{TyVar}$ has a class $\text{class}(\alpha) \in \text{Class}$, and there are infinitely many type variables of each class. A row variable has no class. Each $t \in \text{TyName}$ has an arity $k \geq 0$; we write $\text{TyName}^{(k)}$ for the type names of arity k.

A type variable *tyvar* of the syntax names one type variable, also written *tyvar*, whose class is fixed by the beginning of its name: number if the name begins with `number`, comparable if it begins with `comparable`, appendable if it begins with `appendable`, compappend if it begins with `compappend`, and any otherwise. When that class is any, *tyvar* also names one row variable; a type variable of the syntax of any other class names no row variable. We write $tyvar = \alpha$ and $tyvar = \epsilon$ when α, respectively ϵ, is the variable named by *tyvar*. No other type or row variable is named by a type variable of the syntax.

A label and a value variable may be spelt alike. For a label *lab* we write $var = lab$ when *var* is spelt as *lab*.

The rules mention a few type names directly. They are shown in Figure 13; they belong to the initial basis (Appendix C), and a rule that names one of them means that type name, whatever a program binds to the type constructor of the same spelling.

$\texttt{Int}, \texttt{Float} \in \text{TyName}^{(0)}$	integers, floats
$\texttt{Char}, \texttt{String} \in \text{TyName}^{(0)}$	characters, strings
$() \in \text{TyName}^{(0)}$	unit
$\texttt{List} \in \text{TyName}^{(1)}$	lists
$\texttt{Tuple}_2 \in \text{TyName}^{(2)}$	pairs
$\texttt{Tuple}_3 \in \text{TyName}^{(3)}$	triples

Figure 13: Built-in Type Names

4.2 Compound Objects

When A and B are sets, $\text{Fin}\, A$ denotes the set of finite subsets of A, and $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B. The domain and range of a finite map f are denoted $\text{Dom}\, f$ and $\text{Ran}\, f$. A finite map is written explicitly as $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$. We write $\{x \mapsto e\,;\, \phi\}$ for the finite map f whose domain is the set of x satisfying the condition ϕ and whose value on that domain is $f(x) = e$. A finite map is regarded as the set of its pairs, so $f \subseteq g$ means that $\text{Dom}\, f \subseteq \text{Dom}\, g$ and that g agrees with f on $\text{Dom}\, f$.

When f and g are finite maps, the map $f + g$, called f *modified* by g, is the finite map with domain $\mathrm{Dom}\, f \cup \mathrm{Dom}\, g$ and values

$$(f + g)(a) = \text{if } a \in \mathrm{Dom}\, g \text{ then } g(a) \text{ else } f(a).$$

The compound objects of the static semantics of the Core are shown in Figure 14. We take $\cup$ to mean disjoint union over semantic object classes, and we understand all the defined object classes to be disjoint. The classes Type, Row, RecType, FunType and ConsType are the smallest sets satisfying their equations; hence every type is finite, and no type occurs within itself. A sequence $(\tau_1, \cdots, \tau_k)$ of types is also written $\tau^{(k)}$, and a sequence $(\xi_1, \cdots, \xi_k)$ of type and row variables $\xi^{(k)}$; the metavariable ξ ranges over $\mathrm{TyVar} \cup \mathrm{RowVar}$. In a context, $T \in \mathrm{Fin}\,(\mathrm{TyName})$ and $U \in \mathrm{Fin}\,(\mathrm{TyVar} \cup \mathrm{RowVar})$.

$$\begin{array}{rll}
\tau \in \mathrm{Type} & = & \mathrm{TyVar} \cup \mathrm{RecType} \cup \mathrm{FunType} \cup \mathrm{ConsType} \\
\varrho \in \mathrm{Row} & = & \mathrm{Lab} \xrightarrow{\mathrm{fin}} \mathrm{Type} \\
\{\varrho\} \text{ or } \{\varrho \mid \epsilon\} \in \mathrm{RecType} & = & \mathrm{Row} \times (\mathrm{RowVar} \cup \{\emptyset\}) \\
\tau \to \tau' \in \mathrm{FunType} & = & \mathrm{Type} \times \mathrm{Type} \\
\in \mathrm{ConsType} & = & \bigcup_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t \in \mathrm{ConsType}^{(k)} & = & \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\xi^{(k)}.\tau \in \mathrm{TypeFcn} & = & \bigcup_{k \geq 0} (\mathrm{TyVar} \cup \mathrm{RowVar})^k \times \mathrm{Type} \\
\sigma \text{ or } \forall\xi^{(k)}.\tau \in \mathrm{TypeScheme} & = & \bigcup_{k \geq 0} (\mathrm{TyVar} \cup \mathrm{RowVar})^k \times \mathrm{Type} \\
(\theta,\, VE) \in \mathrm{TyStr} & = & \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
TE \in \mathrm{TyEnv} & = & \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{TyStr} \\
VE \in \mathrm{ValEnv} & = & (\mathrm{Var} \cup \mathrm{Con}) \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
QE \in \mathrm{QualEnv} & = & \mathrm{ModId} \xrightarrow{\mathrm{fin}} \mathrm{Env} \\
E \text{ or } QE, TE, VE \in \mathrm{Env} & = & \mathrm{QualEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
C \text{ or } T, U, E \in \mathrm{Context} & = & \mathrm{Fin}\,(\mathrm{TyName}) \times \mathrm{Fin}\,(\mathrm{TyVar} \cup \mathrm{RowVar}) \\
 & & \times\, \mathrm{Env}
\end{array}$$

Figure 14: Compound Semantic Objects

A record type is a pair of a row and either a row variable or $\emptyset$. The record type $(\varrho, \emptyset)$, written $\{\varrho\}$, is *closed*: its fields are exactly those of ϱ. The record type (ϱ, ϵ), written $\{\varrho \mid \epsilon\}$, is *open*: its fields are those of ϱ together with those that ϵ stands for, none of which has a label in $\mathrm{Dom}\, \varrho$ (Section 4.5). The unit type is the constructed type $()()$, and the type of k-tuples with components $\tau_1, \cdots, \tau_k$ is $(\tau_1, \cdots, \tau_k)\mathtt{Tuple}_k$. When t has arity 0 we write t for the type $()t$; thus `Int`, $()$ and so on are types.

Note that Λ and $\forall$ bind type and row variables. For any semantic object A, $\mathrm{tynames}\, A$ and $\mathrm{tyvars}\, A$ denote respectively the set of type names and the set of type and row variables occurring free in A. For a context we take $\mathrm{tyvars}\, C = \mathrm{tyvars}\,(E \text{ of } C) \cup (U \text{ of } C)$. For a type expression ty, $\mathrm{tyvars}\, ty$ denotes the set of type variables of the syntax occurring in ty.

A value environment maps value variables and value constructors to a pair of a type scheme and an identifier status. If $VE(var) = (\sigma, is)$ or $VE(con) = (\sigma, is)$, we say that *var*, respectively *con*, *has status is* in VE. Value variables

have status v. The constructors of custom types have status c, and only they may occur in patterns; the constructor that a record type alias defines has status v (Chapter 5). We take Con to contain, besides the constructors spelt by identifiers, the two list constructor names `[]` and `::`, which no identifier spells; the initial basis binds them with status c to $\forall\alpha.\alpha\,\texttt{List}$ and $\forall\alpha.\alpha \to \alpha\,\texttt{List} \to \alpha\,\texttt{List}$, where $\text{class}(\alpha) = \text{any}$ (Appendix C).

A qualified environment maps each qualifier to the environment of the names it qualifies; every environment in the range of a qualified environment has the empty qualified environment. In a context $C = T, U, E$, T is the set of type names generated so far, and U is the set of explicit type and row variables in scope (Section 4.7).

4.3 Projection, Injection and Modification

Projection: We often select a component of a tuple of objects, and we rely on metavariable names to say which component. For instance "VE of E" means the value environment component of E, and, for a record type τ, "ϱ of τ" means its row.

When a tuple contains a finite map we apply the tuple to an argument, relying on the syntactic class of the argument to select the map. Thus $C(var)$ means $(VE \text{ of } (E \text{ of } C))(var)$, and likewise $C(con)$; $C(tycon)$ means $(TE \text{ of } (E \text{ of } C))(tycon)$. A long identifier is looked up through the qualified environment: if $longvar = modid.var$ then $E(longvar)$ means $(VE \text{ of } (QE \text{ of } E)(modid))(var)$, and likewise for $longcon$ and $longtycon$; an unqualified $longvar$ is a var. $C(longvar)$ means $(E \text{ of } C)(longvar)$. The identifiers that the equivalent forms of derived forms introduce (Appendix A) are looked up in the same way. Because only an import of the module `Basics` provides the qualifier `Basics` (Section 3.7), `Basics.True`, `Basics.False` and `Basics.negate` always denote the bindings of the initial basis (Appendix C); the constructor `::` is bound in the initial environment E_0 of that appendix, from which every module is elaborated.

Operators are not in the domain of an environment, and $C(op)$ is not defined. Since no program declares an operator, the operators are fixed by the initial basis: Appendix C gives the *static operator table*

$$\text{OT}_{\text{STAT}} : \text{Op} \xrightarrow{\text{fin}} \text{TypeScheme} \times \text{IdStatus},$$

which maps each operator of Figure 3 to the type scheme and status of the identifier that the operator stands for; in particular $\text{OT}_{\text{STAT}}(::)$ is the type scheme of the list constructor `::`, with status c. The operators of library modules outside the initial basis are outside the definition.

Injection: Components may be injected into tuple classes; for example, "VE in Env" means the environment $(\{\}, \{\}, VE)$.

Modification: The modification of one map by another has been defined above. It is extended to environments component by component, $E + E'$, and empty components are left implicit: $E + VE$ means $E + (\{\}, \{\}, VE)$. For set components modification means union, so that $C + (T, U, VE)$ means

$$\big((T \text{ of } C) \cup T,\ (U \text{ of } C) \cup U,\ (E \text{ of } C) + VE\big),$$

and $C + VE$ and $C + U$ leave the other components unchanged. Finally, $C \oplus TE$ means $C +$ (tynames TE, TE): the type environment is added and T is extended with its type names.

4.4 Classes and Admissible Types

The classes are ordered by $\leq$, the least partial order on Class such that

$$\begin{array}{cc} \text{number} \leq \text{comparable}, & \text{compappend} \leq \text{comparable}, \\ \text{compappend} \leq \text{appendable}, & \kappa \leq \text{any for every } \kappa \in \text{Class}. \end{array}$$

For each class κ, the set $\text{Adm}_\kappa \subseteq \text{Type}$ of types *admissible* for κ is defined in Figure 15; $\text{Adm}_{\text{comparable}}$ is the smallest set satisfying its equation. A type variable of class κ may stand only for a type in Adm_κ (Section 4.5). A type variable belongs to Adm_κ exactly when its class is at most κ; no record type and no function type is admissible for any class but any.

$$\begin{aligned} \text{Adm}_{\text{any}} &= \text{Type} \\ \text{Adm}_{\text{number}} &= \{\texttt{Int}, \texttt{Float}\} \cup \{\alpha\,;\, \text{class}(\alpha) = \text{number}\} \\ \text{Adm}_{\text{comparable}} &= \{\texttt{Int}, \texttt{Float}, \texttt{Char}, \texttt{String}\} \\ &\quad \cup \{\alpha\,;\, \text{class}(\alpha) \leq \text{comparable}\} \\ &\quad \cup \{\tau\,\texttt{List}\,;\, \tau \in \text{Adm}_{\text{comparable}}\} \\ &\quad \cup \{(\tau_1, \tau_2)\texttt{Tuple}_2\,;\, \tau_1, \tau_2 \in \text{Adm}_{\text{comparable}}\} \\ &\quad \cup \{(\tau_1, \tau_2, \tau_3)\texttt{Tuple}_3\,;\, \tau_1, \tau_2, \tau_3 \in \text{Adm}_{\text{comparable}}\} \\ \text{Adm}_{\text{appendable}} &= \{\texttt{String}\} \cup \{\tau\,\texttt{List}\,;\, \tau \in \text{Type}\} \\ &\quad \cup \{\alpha\,;\, \text{class}(\alpha) \leq \text{appendable}\} \\ \text{Adm}_{\text{compappend}} &= \{\texttt{String}\} \cup \{\tau\,\texttt{List}\,;\, \tau \in \text{Adm}_{\text{comparable}}\} \\ &\quad \cup \{\alpha\,;\, \text{class}(\alpha) = \text{compappend}\} \end{aligned}$$

Figure 15: Admissible Types

There is no class of types that admit equality. The initial basis gives the equality operators the type scheme $\forall\alpha.\alpha \to \alpha \to \texttt{Bool}$ with $\text{class}(\alpha) = \text{any}$ (Appendix C), so equality elaborates at every type, function types included.

4.5 Types and Type Functions

Substitution. Let $\xi^{(k)}$ be distinct type and row variables, and $\tau^{(k)}$ types such that $\tau_i \in \text{RecType}$ whenever $\xi_i \in \text{RowVar}$. For a type τ, the type $\tau\{\tau^{(k)}/\xi^{(k)}\}$ is the result of replacing each ξ_i by τ_i throughout τ. Writing $\tau\{\cdot\}$ for $\tau\{\tau^{(k)}/\xi^{(k)}\}$, and $\varrho\{\cdot\}$ for $\{lab \mapsto \varrho(lab)\{\cdot\}\,;\, lab \in \text{Dom}\,\varrho\}$, it is defined by

$$\begin{aligned}
\alpha\{\cdot\} &= \tau_i && \text{if } \alpha = \xi_i \\
\alpha\{\cdot\} &= \alpha && \text{if } \alpha \notin \{\xi_1, \cdots, \xi_k\} \\
\{\varrho\}\{\cdot\} &= \{\varrho\{\cdot\}\} && \\
\{\varrho \mid \epsilon\}\{\cdot\} &= \{\varrho\{\cdot\} + \varrho'\} && \text{if } \epsilon = \xi_i \text{ and } \tau_i = \{\varrho'\} \\
\{\varrho \mid \epsilon\}\{\cdot\} &= \{\varrho\{\cdot\} + \varrho' \mid \epsilon'\} && \text{if } \epsilon = \xi_i \text{ and } \tau_i = \{\varrho' \mid \epsilon'\} \\
\{\varrho \mid \epsilon\}\{\cdot\} &= \{\varrho\{\cdot\} \mid \epsilon\} && \text{if } \epsilon \notin \{\xi_1, \cdots, \xi_k\} \\
(\tau \to \tau')\{\cdot\} &= \tau\{\cdot\} \to \tau'\{\cdot\} && \\
(\tau'_1, \cdots, \tau'_l)t\{\cdot\} &= (\tau'_1\{\cdot\}, \cdots, \tau'_l\{\cdot\})t &&
\end{aligned}$$

The result is defined only if, wherever a row variable is replaced, $\text{Dom}\,\varrho \cap \text{Dom}\,\varrho' = \emptyset$. Thus a row variable can never give a record type a second field with a label it already has. The substitution *respects classes* if $\tau_i \in \text{Adm}_{\text{class}(\xi_i)}$ for each $\xi_i \in \text{TyVar}$. Substitution is extended to rows, as above, and to other objects in the obvious way, renaming bound variables where necessary.

The rules rely on the following property, which may easily be demonstrated.

> If $\kappa \leq \kappa'$ then $\text{Adm}_\kappa \subseteq \text{Adm}_{\kappa'}$. If $\tau \in \text{Adm}_\kappa$, and $\tau\{\tau^{(k)}/\xi^{(k)}\}$ is defined and respects classes, then $\tau\{\tau^{(k)}/\xi^{(k)}\} \in \text{Adm}_\kappa$.

Type functions. A type function $\theta = \Lambda\xi^{(k)}.\tau$ has arity k; the bound variables must be distinct. Two type functions are equal if they differ only in their choice of bound variables, a type variable being renamed to a type variable and a row variable to a row variable. The class of a bound type variable of a type function has no significance. If t has arity k, then we write t for the type function $\Lambda\alpha^{(k)}.\alpha^{(k)}t$, where the α_i are distinct type variables; thus $\text{TyName} \subseteq \text{TypeFcn}$.

The application of $\theta = \Lambda\xi^{(k)}.\tau$ to types $\tau^{(k)}$ is

$$\theta(\tau^{(k)}) = \tau\{\tau^{(k)}/\xi^{(k)}\},$$

defined when the substitution is defined; it need not respect classes. In particular a parameter of θ that is a row variable accepts only a record type.

4.6 Type Schemes

A type scheme $\sigma = \forall\xi^{(k)}.\tau$ *generalises* a type τ', written $\sigma \succ \tau'$, if $\tau' = \tau\{\tau^{(k)}/\xi^{(k)}\}$ for some $\tau^{(k)}$ for which the substitution is defined and respects

classes; τ' is then an *instance* of σ. If $\sigma' = \forall \xi'^{(l)}.\tau'$, then σ generalises σ', written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and no ξ'_j occurs free in σ. By the property of Section 4.5, if $\sigma \succ \sigma'$ and $\sigma' \succ \tau''$ then $\sigma \succ \tau''$.

Two type schemes are equal if they can be obtained from each other by renaming and reordering bound variables, and by deleting bound variables that do not occur in the body, other than the variables of class number that closure adds to the scheme of a binding of a group (Section 4.8), which are kept. of the same class and a row variable by a row variable. We consider a type τ to be a type scheme, identifying it with $\forall().\tau$. For a finite set U of type and row variables, $\forall U.\tau$ means $\forall \xi^{(k)}.\tau$ where $\xi^{(k)}$ is any enumeration of U.

With each special constant *scon* we associate a type scheme type(*scon*): it is $\forall\alpha.\alpha$, where $\text{class}(\alpha) = \text{number}$, if *scon* is an integer constant, and the type `Float`, `Char` or `String` if *scon* is a floating point, character or string constant. Thus the instances of type(*scon*) for an integer constant are the types in $\text{Adm}_{\text{number}}$.

4.7 Scope of Explicit Type Variables

The type variables written in a type annotation are *explicit*: within the annotated definition each stands for one type or row variable that the definition may not instantiate. Explicit variables in scope are recorded in U of C.

For a set U of type and row variables, $\text{names}\, U$ is the set of type variables *tyvar* of the syntax such that *tyvar* $= \xi$ for some $\xi \in U$. The set U is *unambiguous* if no *tyvar* names two members of U. We consider only contexts C in which U of C is unambiguous; the rules preserve this.

Let *ty* be the type expression of an annotation elaborated in context C. A set U' of type and row variables is *new for ty in C* if

$$\text{names}\, U' = (\text{tyvars}\, ty) \setminus \text{names}\, (U \text{ of } C) \text{ and}$$
$$(U \text{ of } C) \cup U' \text{ is unambiguous.}$$

Thus U' holds, for each type variable of *ty* that is not already explicit in C, the type variable or the row variable that it names: the row variable when *ty* uses it as the extension of a record type (`{ r | ... }`), and the type variable otherwise. The rules for type expressions accept only variables of U of C, and both kinds could elaborate a name written alone (rules (22) and (23)), so the kind is fixed here: in `f : a -> a` the name `a` is a type variable, and in `f : { r | x : Int } -> r` the name `r` is a row variable. The members of U' are added to U while the annotation and the body of its definition elaborate, so that an annotation nested within the body that writes one of these names means the same variable. They are bound in the type scheme of the annotated variable, and they must not occur free in the context of the definition: an explicit variable does not escape its annotation.

4.8 Closure

Let A be a semantic object and $\sigma = \forall \xi^{(k)}.\tau$ a type scheme. Then $\mathrm{Clos}_A\sigma$, the *closure* of σ with respect to A, is the type scheme $\forall \xi^{(k)}\xi'^{(l)}.\tau$, where $\xi'^{(l)}$ enumerates $\mathrm{tyvars}\,\sigma \setminus \mathrm{tyvars}\,A$. In particular, for a type τ, $\mathrm{Clos}_A\tau = \forall \xi^{(l)}.\tau$ where $\xi^{(l)}$ enumerates $\mathrm{tyvars}\,\tau \setminus \mathrm{tyvars}\,A$. For a value environment, $\mathrm{Clos}_A\,VE$ is the value environment obtained from VE by replacing each pair (σ, is) in its range by $(\mathrm{Clos}_A\sigma, is)$.

Commonly A is a context C. Since U of $C \subseteq \mathrm{tyvars}\,C$, closure never generalises an explicit variable in scope. Closure applies to every group of value bindings, pattern bindings included, whatever the form of their expressions.

A group of value bindings is closed as one. When rules (19) and (56) take $\mathrm{Clos}_C\,VE$ of the value environment VE of a group, the scheme σ of each binding is closed also over the type variables of class number in $\mathrm{tyvars}\,VE \setminus (\mathrm{tyvars}\,\sigma \cup \mathrm{tyvars}\,C)$: those of the other bindings of the group that its own type does not mention. Since such a variable does not occur in the binding's type, it changes no instance of the scheme. It matters only to number specialisation (Section 6.2): an occurrence of the binding gives it a type that nothing fixes, which is then `Float`.

4.9 Inference Rules

Each rule of the semantics allows inferences among sentences of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

where A is usually a context, *phrase* is a phrase of the Core, and A' is a semantic object, usually a type, a row, a value environment, or a pair of a value environment and a type. It is pronounced "*phrase* elaborates to A' in (context) A". Some rules have extra hypotheses not of this form; they are called *side conditions*.

In the presentation of the rules, phrases within single angle brackets $\langle\ \rangle$ are called *first options*, and those within double angle brackets $\langle\langle\ \rangle\rangle$ are called *second options*. To reduce the number of rules, we have adopted the following convention:

> In each instance of a rule, the first options must be either all present or all absent; similarly the second options must be either all present or all absent.

No Core phrase generates type names, so T of C is the same in every sentence of a derivation for a Core phrase; it is used by the static semantics of Modules (Chapter 5). A phrase elaborates to A' when some derivation of the sentence exists; where the premises of a rule leave a type in its conclusion

undetermined, the derivation may choose any type that satisfies them. The relation $\vdash$ is overloaded for all syntactic classes.

Atomic Expressions

$\boxed{C \vdash \mathit{atexp} \Rightarrow \tau}$

$$\frac{\mathrm{type}(\mathit{scon}) \succ \tau}{C \vdash \mathit{scon} \Rightarrow \tau} \tag{1}$$

$$\frac{C(\mathit{longvar}) = (\sigma, \mathit{is}) \qquad \sigma \succ \tau}{C \vdash \mathit{longvar} \Rightarrow \tau} \tag{2}$$

$$\frac{C(\mathit{longcon}) = (\sigma, \mathit{is}) \qquad \sigma \succ \tau}{C \vdash \mathit{longcon} \Rightarrow \tau} \tag{3}$$

$$\frac{\mathrm{OT}_{\mathrm{STAT}}(\mathit{op}) = (\sigma, \mathit{is}) \qquad \sigma \succ \tau}{C \vdash \texttt{(}\,\mathit{op}\,\texttt{)} \Rightarrow \tau} \tag{4}$$

$$\frac{}{C \vdash \texttt{()} \Rightarrow ()} \tag{5}$$

$$\frac{C \vdash \mathit{exp}_i \Rightarrow \tau_i \quad (1 \le i \le k)}{C \vdash \texttt{(}\,\mathit{exp}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{exp}_k\,\texttt{)} \Rightarrow (\tau_1, \cdots, \tau_k)\texttt{Tuple}_k} \tag{6}$$

$$\frac{}{C \vdash \texttt{[ ]} \Rightarrow \tau\,\texttt{List}} \tag{7}$$

$$\frac{\langle C \vdash \mathit{exprow} \Rightarrow \varrho\rangle}{C \vdash \texttt{\{}\,\langle \mathit{exprow}\rangle\,\texttt{\}} \Rightarrow \{\{\}\langle + \varrho\rangle\}} \tag{8}$$

$$\frac{C \vdash \mathit{var} \Rightarrow \tau \qquad \tau \in \mathrm{RecType} \qquad C \vdash \mathit{exprow} \Rightarrow \varrho \qquad \varrho \subseteq (\varrho \text{ of } \tau)}{C \vdash \texttt{\{}\ \mathit{var}\ \texttt{|}\ \mathit{exprow}\ \texttt{\}} \Rightarrow \tau} \tag{9}$$

$$\frac{C \vdash \mathit{dec} \Rightarrow \mathit{VE} \qquad C + \mathit{VE} \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp} \Rightarrow \tau} \tag{10}$$

$$\frac{C \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \texttt{(}\,\mathit{exp}\,\texttt{)} \Rightarrow \tau} \tag{11}$$

Comments:

(1) For an integer constant, τ may be any type in $\mathrm{Adm}_{\mathrm{number}}$: `Int`, `Float` or a type variable of class number. A floating point, character or string constant has exactly its type. The type that counts for evaluation is the one the principal derivation gives (Section 4.10); where it is a variable that nothing binds, number specialisation takes it to be `Float` (Section 6.2).

(2) The instantiation of type schemes allows different occurrences of one *longvar* to take different types. The identifier status is not used in this rule, nor in rules (3) and (4); in particular the constructor of a record type alias, of status v, elaborates by rule (3) as any constructor does.

(4) Infixed applications of operators are derived forms of this phrase (Appendix A), so this rule gives the type of every application of an operator. The context C is absent from the premise, so an operator has the same type scheme in every module, whatever the module declares or imports: a declaration of `add` in a program does not change the meaning of `+`.

(6) By the grammar k is 2 or 3.

(7) Note that τ does not occur in any premise; thus `[ ]` has type τ `List` for every τ.

(8) A record expression has a closed record type: its fields are exactly those written.

(9) The side conditions require the type of *var* to be a record type that has every updated field, each at the type of its new value. The result has the type of *var*, so an update neither adds a field nor changes the type of one: if `r` has type $\{\{\texttt{x} \mapsto \texttt{Int}\}\}$ then `{ r | x = "s" }` does not elaborate. The type of *var* may be open, so a function may update a field of any record that has it.

(10) The variables bound by *dec* are in scope in *exp* at the type schemes that rule (19) gives them, so *exp* may use them polymorphically.

Expression Rows

$\boxed{C \vdash \mathit{exprow} \Rightarrow \varrho}$

$$\frac{C \vdash \mathit{exp} \Rightarrow \tau \qquad \langle C \vdash \mathit{exprow} \Rightarrow \varrho \rangle}{C \vdash \mathit{lab}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{,}\ \mathit{exprow} \rangle \Rightarrow \{\mathit{lab} \mapsto \tau\} \langle + \varrho \rangle} \tag{12}$$

Comments:

(12) The syntactic restrictions ensure $\mathit{lab} \notin \mathrm{Dom}\,\varrho$.

Expressions

$\boxed{C \vdash \mathit{exp} \Rightarrow \tau}$

$$\frac{C \vdash \mathit{atexp} \Rightarrow \tau}{C \vdash \mathit{atexp} \Rightarrow \tau} \tag{13}$$

$$\frac{C \vdash \mathit{exp} \Rightarrow \tau' \to \tau \qquad C \vdash \mathit{atexp} \Rightarrow \tau'}{C \vdash \mathit{exp}\ \ \mathit{atexp} \Rightarrow \tau} \tag{14}$$

$$\frac{C \vdash atpat \Rightarrow (VE, \tau) \qquad C + VE \vdash exp \Rightarrow \tau'}{C \vdash \backslash\ atpat\ \texttt{->}\ exp \Rightarrow \tau \to \tau'} \tag{15}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash match \Rightarrow \tau \to \tau'}{C \vdash \texttt{case}\ exp\ \texttt{of}\ match \Rightarrow \tau'} \tag{16}$$

Comments:

(13) The relational symbol $\vdash$ is overloaded for all syntactic classes (here atomic expressions and expressions).

(15) The variables bound by *atpat* receive types, not type schemes, so a parameter has one type throughout *exp*. Restriction 2 of Section 4.11 requires *atpat* to be exhaustive.

Matches

$\boxed{C \vdash match \Rightarrow \tau}$

$$\frac{C \vdash mrule \Rightarrow \tau \qquad \langle C \vdash match \Rightarrow \tau \rangle}{C \vdash mrule\ \langle\texttt{;}\ match \rangle \Rightarrow \tau} \tag{17}$$

Match Rules

$\boxed{C \vdash mrule \Rightarrow \tau}$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C + VE \vdash exp \Rightarrow \tau'}{C \vdash pat\ \texttt{->}\ exp \Rightarrow \tau \to \tau'} \tag{18}$$

Comments:

(18) This rule lets new free type variables enter the context: the types of the variables of *pat* are chosen in elaborating *pat*, and they may have to agree with the explicit variables of annotations within *exp*.

Declarations

$\boxed{C \vdash dec \Rightarrow VE}$

$$\frac{C + VE \vdash valbind \Rightarrow VE \qquad VE' = \mathrm{Clos}_C\, VE}{C \vdash valbind \Rightarrow VE'} \tag{19}$$

Comments:

(19) A declaration is one group of mutually dependent bindings; a `let` with several definitions is a derived form that divides them into groups (Appendix A). Modifying C by VE on the left puts every member of the group in scope in every member's expression. Since VE is the same on both sides, an unannotated member has one type throughout the group: it is monomorphic within the group, and is generalised only by the closure, for the body of the `let`. An annotated member enters VE with

the type scheme of its annotation, so it may be used polymorphically within the group, in its own body included.

The closure applies to every member whatever its expression, so there is no value restriction: in `let n = 5 in ( n // 2, n / 2 )` the variable `n` is used at `Int` and at `Float`. The closure leaves the scheme of an annotated member unchanged, since its free variables are explicit variables of C.

Value Bindings

$\boxed{C \vdash \mathit{valbind} \Rightarrow \mathit{VE}}$

$$\frac{\langle U' \text{ is new for } \mathit{ty} \text{ in } C \qquad C + U' \vdash \mathit{ty} \Rightarrow \tau \qquad U' \cap \operatorname{tyvars} C = \emptyset\rangle \qquad C\langle + U'\rangle \vdash \mathit{exp} \Rightarrow \tau \qquad \langle\!\langle C \vdash \mathit{valbind} \Rightarrow \mathit{VE}\rangle\!\rangle}{C \vdash \mathit{var}\ \langle \texttt{:}\ \mathit{ty}\rangle\ \texttt{=}\ \mathit{exp}\ \langle\!\langle \texttt{and}\ \mathit{valbind}\rangle\!\rangle \Rightarrow \{\mathit{var} \mapsto (\langle\forall U'.\rangle\tau, \mathrm{v})\}\ \langle\!\langle + \mathit{VE}\rangle\!\rangle} \tag{20}$$

$$\frac{C \vdash \mathit{atpat} \Rightarrow (\mathit{VE}, \tau) \qquad C \vdash \mathit{exp} \Rightarrow \tau}{C \vdash \mathit{atpat}\ \texttt{=}\ \mathit{exp} \Rightarrow \mathit{VE}} \tag{21}$$

Comments:

(20) When the second option is present, $\mathit{var} \notin \operatorname{Dom} \mathit{VE}$ by the syntactic restrictions. With an annotation, U' holds the variables of *ty* not already explicit in C (Section 4.7). They are in U while *ty* and *exp* elaborate, so an annotation nested in *exp* that writes one of their names means the same variable. Since τ is fixed by *ty*, *exp* must have exactly that type, with the variables of U' uninstantiated: an annotation may be less general than the type *exp* could have, but never more general. Thus `f : a -> a` with `f x = x + 1` does not elaborate, nor does `g : number -> number` with `g x = x // 2`.

The condition $U' \cap \operatorname{tyvars} C = \emptyset$ prevents an explicit variable from escaping its annotation. In `f x = let g : a -> a ; g y = x in g x`, the body of `g` has type `a -> a` only if `x` has the explicit type `a`, which is then free in the context of `g`; so the expression does not elaborate.

(21) A pattern binding is a group by itself. The variables of *atpat* receive types here and are generalised by rule (19). That rule also puts them in scope in *exp*, but the syntactic restrictions forbid *exp* to use them, since no pattern binding lies on a cycle of definitions. A binding *var* `=` *exp* is read by rule (20).

Type Expressions

$\boxed{C \vdash \mathit{ty} \Rightarrow \tau}$

$$\frac{\mathit{tyvar} = \alpha \qquad \alpha \in U \text{ of } C}{C \vdash \mathit{tyvar} \Rightarrow \alpha} \tag{22}$$

$$\frac{tyvar = \epsilon \qquad \epsilon \in U \text{ of } C}{C \vdash tyvar \Rightarrow \{\{\} \mid \epsilon\}} \tag{23}$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \texttt{\{}\, \langle tyrow \rangle \,\texttt{\}} \Rightarrow \{\{\} \langle + \varrho \rangle\}} \tag{24}$$

$$\frac{tyvar = \epsilon \qquad \epsilon \in U \text{ of } C \qquad C \vdash tyrow \Rightarrow \varrho}{C \vdash \texttt{\{}\ tyvar\ \texttt{|}\ tyrow\ \texttt{\}} \Rightarrow \{\varrho \mid \epsilon\}} \tag{25}$$

$$\frac{}{C \vdash \texttt{()} \Rightarrow ()} \tag{26}$$

$$\frac{C \vdash ty_i \Rightarrow \tau_i \quad (1 \le i \le k)}{C \vdash \texttt{(}\ ty_1\texttt{,}\ \cdots\texttt{,}\ ty_k\ \texttt{)} \Rightarrow (\tau_1, \cdots, \tau_k)\texttt{Tuple}_k} \tag{27}$$

$$\frac{C(longtycon) = (\theta, VE) \qquad C \vdash ty_i \Rightarrow \tau_i \quad (1 \le i \le k)}{C \vdash longtycon\ ty_1 \cdots ty_k \Rightarrow \theta(\tau_1, \cdots, \tau_k)} \tag{28}$$

$$\frac{C \vdash ty \Rightarrow \tau \qquad C \vdash ty' \Rightarrow \tau'}{C \vdash ty\ \texttt{->}\ ty' \Rightarrow \tau \to \tau'} \tag{29}$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash \texttt{(}\ ty\ \texttt{)} \Rightarrow \tau} \tag{30}$$

Comments:

(22) Rules (22) and (23) both cover the phrase *tyvar*. Since U of C is unambiguous, at most one of them applies, according to whether the variable of U of C named by *tyvar* is a type variable or a row variable. A type variable that is not explicit in C does not elaborate: every type variable of an annotation is made explicit by rule (20), and Chapter 5 makes the parameters of type declarations explicit in the same way.

(23) A row variable written alone denotes the open record type with no fields of its own: the record of the fields that ϵ stands for.

(25) Only a type variable of class any names a row variable, so `{ number | x : Int }` does not elaborate.

(27) By the grammar k is 2 or 3.

(28) For $\theta(\tau_1, \cdots, \tau_k)$ to be defined, θ must have arity k, so a type constructor is never partially applied. Application of a type function performs the substitution, so a type alias denotes its expansion, and an alias with a row parameter applies only to a record type.

Type-expression Rows

$\boxed{C \vdash tyrow \Rightarrow \varrho}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab \ \texttt{:} \ ty \ \langle \texttt{,} \ tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle} \tag{31}$$

Comments:

(31) The syntactic restrictions ensure $lab \notin \mathrm{Dom}\,\varrho$.

Atomic Patterns

$\boxed{C \vdash atpat \Rightarrow (VE, \tau)}$

$$\frac{}{C \vdash \texttt{_} \Rightarrow (\{\}, \tau)} \tag{32}$$

$$\frac{\mathrm{type}(scon) \succ \tau \qquad \tau \in \{\texttt{Int}, \texttt{Char}, \texttt{String}\}}{C \vdash scon \Rightarrow (\{\}, \tau)} \tag{33}$$

$$\frac{}{C \vdash var \Rightarrow (\{var \mapsto (\tau, \mathrm{v})\}, \tau)} \tag{34}$$

$$\frac{C(longcon) = (\sigma, \mathrm{c}) \qquad \sigma \succ \tau^{(k)}t}{C \vdash longcon \Rightarrow (\{\}, \tau^{(k)}t)} \tag{35}$$

$$\frac{}{C \vdash \texttt{()} \Rightarrow (\{\}, ())} \tag{36}$$

$$\frac{C \vdash pat_i \Rightarrow (VE_i, \tau_i) \quad (1 \le i \le k)}{C \vdash \texttt{(} \ pat_1 \texttt{,} \ \cdots \texttt{,} \ pat_k \ \texttt{)} \Rightarrow (VE_1 + \cdots + VE_k, \ (\tau_1, \cdots, \tau_k)\texttt{Tuple}_k)} \tag{37}$$

$$\frac{}{C \vdash \texttt{[ ]} \Rightarrow (\{\}, \tau \, \texttt{List})} \tag{38}$$

$$\frac{\langle C \vdash patrow \Rightarrow (VE, \varrho) \rangle \qquad \tau \in \mathrm{RecType} \qquad \{\}\langle + \varrho \rangle \subseteq (\varrho \text{ of } \tau)}{C \vdash \texttt{\{} \ \langle patrow \rangle \ \texttt{\}} \Rightarrow (\{\}\langle + VE \rangle, \ \tau)} \tag{39}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash \texttt{(} \ pat \ \texttt{)} \Rightarrow (VE, \tau)} \tag{40}$$

Comments:

(33) The side conditions give an integer constant the type `Int` in a pattern, never `Float` or a type variable of class number; a character or string constant has its own type. Thus `case 1.5 of 1 -> 0 ; _ -> 1` does not elaborate. Floating point constants are not patterns, by the grammar.

(34) The variable receives a type, not a type scheme. Constructors and variables are spelt differently, so no status test is needed here.

(35) Only status c is accepted, so the constructor of a record type alias, of status v, cannot occur in a pattern. The instance must be a constructed type, so this rule accepts only a constructor without arguments; a constructor with arguments is covered by rule (43).

(37) By the grammar k is 2 or 3, and by the syntactic restrictions the domains of the VE_i are disjoint.

(38) As in rule (7), τ is arbitrary.

(39) The record type τ may have fields besides those the pattern names, and it may be open: a record pattern matches every record with the fields it names.

Pattern Rows

$\boxed{C \vdash \mathit{patrow} \Rightarrow (\mathit{VE}, \varrho)}$

$$\frac{\langle C \vdash \mathit{patrow} \Rightarrow (\mathit{VE}, \varrho)\rangle}{C \vdash \mathit{lab}\ \texttt{=}\ \mathit{var}\ \langle\texttt{,}\ \mathit{patrow}\rangle \Rightarrow (\{\mathit{var} \mapsto (\tau, \mathrm{v})\}\langle + \mathit{VE}\rangle,\ \{\mathit{lab} \mapsto \tau\}\langle + \varrho\rangle)} \tag{41}$$

Comments:

(41) A field of a record pattern binds the variable of its row to the type of the field. In a program the row is written as a label alone, whose equivalent form lab = var has $\mathit{var} = \mathit{lab}$ (Appendix A); a copy made by number specialisation gives var a new name and keeps lab (Section 6.2). The syntactic restrictions ensure $\mathit{lab} \notin \mathrm{Dom}\,\varrho$. That $\mathit{var} \notin \mathrm{Dom}\,\mathit{VE}$ follows, in a program, from the restriction that a binding site binds no variable twice, and in a copy from the distinct new names it gives.

Patterns

$\boxed{C \vdash \mathit{pat} \Rightarrow (\mathit{VE}, \tau)}$

$$\frac{C \vdash \mathit{atpat} \Rightarrow (\mathit{VE}, \tau)}{C \vdash \mathit{atpat} \Rightarrow (\mathit{VE}, \tau)} \tag{42}$$

$$\frac{\begin{array}{c} C(\mathit{longcon}) = (\sigma, \mathrm{c}) \qquad \sigma \succ \tau_1 \to \cdots \to \tau_n \to \tau'^{(k)}t \\ C \vdash \mathit{atpat}_i \Rightarrow (\mathit{VE}_i, \tau_i) \quad (1 \le i \le n) \end{array}}{C \vdash \mathit{longcon}\ \mathit{atpat}_1 \cdots \mathit{atpat}_n \Rightarrow (\mathit{VE}_1 + \cdots + \mathit{VE}_n,\ \tau'^{(k)}t)} \tag{43}$$

$$\frac{C \vdash \mathit{pat}_1 \Rightarrow (\mathit{VE}_1, \tau) \qquad C \vdash \mathit{pat}_2 \Rightarrow (\mathit{VE}_2, \tau\,\texttt{List})}{C \vdash \mathit{pat}_1\ \texttt{::}\ \mathit{pat}_2 \Rightarrow (\mathit{VE}_1 + \mathit{VE}_2,\ \tau\,\texttt{List})} \tag{44}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash pat \texttt{ as } var \Rightarrow (VE + \{var \mapsto (\tau, \mathrm{v})\},\ \tau)} \quad (45)$$

Comments:

(43) The instance must be a constructed type after exactly n arguments, so a constructor in a pattern is applied to as many patterns as it takes. Thus a constructed pattern with too few or too many arguments does not elaborate. The syntactic restrictions ensure that the domains of the VE_i are disjoint.

(44) The type name is `List` of the initial basis, whatever the program binds to `List`.

(45) The syntactic restrictions ensure $var \notin \mathrm{Dom}\ VE$.

4.10 Principal Derivations

A phrase may elaborate in more than one way. An integer constant may take any type in $\mathrm{Adm}_{\mathrm{number}}$ (rule (1)), and an occurrence of a variable any instance of its type scheme (rule (2)), so two derivations for one phrase may give its integer constants and its occurrences different types. Whether a phrase elaborates does not depend on the choice, but evaluation does: `2 ^ -1` is 0 at `Int` and 0.5 at `Float`. The dynamic semantics is therefore given for one derivation, the *principal* one, which gives no type that the phrase does not force; the further restrictions (Section 4.11) are judged on it too, although for them the choice makes no difference.

In a derivation for a phrase, each node is a sentence about an occurrence of a phrase within it; the nodes of two derivations for the same phrase correspond when they are about the same occurrence. Let D and D' be derivations for the same phrase in the same context C_0, the type and row variables of D that are not free in C_0 having been renamed apart from those of D'. Then D' is an *instance* of D when there is a substitution $S = \{\tau^{(k)}/\xi^{(k)}\}$ (Section 4.5) that respects classes, with no ξ_i free in C_0, defined on every type of D, such that at each pair of corresponding nodes, with contexts C in D and C' in D',

- S takes each variable in U of C to a variable, and U of C' is the set of the variables so obtained;
- each type in the sentence of D' is S applied to the corresponding type in the sentence of D; and
- each type scheme σ' in the sentence of D' is no more general than $S\sigma$, for the corresponding type scheme σ of D: every type τ with $\sigma' \succ \tau$ has $S\sigma \succ \tau$.

Since S is defined on every type of D, it never gives a record type a second field with a label it already has; no Core phrase generates type names, so T of $C = T$ of C'. The third clause allows a derivation to give a binding a less general scheme than another derivation gives it, as when `y = 2 ^ 64 * 4` is given the type `Int` where another derivation gives it the scheme $\forall\alpha.\alpha$ with α of class number. A derivation D for a phrase in a context is *principal* when every derivation for the phrase in that context is an instance of D. Chapter 5 extends the definition to modules (Section 5.6).

> Every Core phrase that elaborates in a context C_0 has a principal derivation in C_0, and any two of its principal derivations are instances of each other, so that they differ only by a renaming of type and row variables not free in C_0.

This is the principal type property of Damas and Milner, extended to classes, rows and annotations. Unification that respects classes has a most general solution whenever it has a solution, since two classes have a greatest common subclass or none: number and comparable give number, appendable and comparable give compappend, and number and appendable none (Section 4.4). The labels of a row are distinct (Section 4.5), the explicit variables of an annotation are fixed throughout its scope (Section 4.7), and a group of value bindings is monomorphic within itself (rule (19)), so the inference algorithm of Damas and Milner, extended in these ways, finds a principal derivation. The proof is not given here.

Only the variables of class number of a principal derivation affect evaluation: the value of an integer constant, and the result of a number operation of the initial basis, depend on whether a type is `Int` or `Float`, while no value depends on how a variable of another class, or of none, is instantiated. Number specialisation (Section 6.2) starts from the principal derivation and fixes its variables of class number.

4.11 Further Restrictions

There are a few restrictions on programs which are better expressed apart from the inference rules. They are judged on the principal derivation (Section 4.10): the type τ with which the patterns of a match elaborate, and the type of a pattern, are those that derivation gives. A phrase that elaborates by the rules but whose principal derivation breaks one of them is not legal; it is treated as a phrase that does not elaborate. In fact every derivation for the phrase gives the same answer. A pattern that the principal derivation elaborates at a type variable, at the top of a match or within another pattern, is a wildcard, a variable, or one of these with `as` or parentheses, since every other pattern (a special constant, a constructor, `()`, a tuple, `[ ]`, `::` or a record pattern) elaborates only at a type that is not a variable; and such a pattern covers

every inhabitant. Another derivation may replace the variable by any type, which has inhabitants (below), and the same patterns cover them.

The first two restrictions speak of the *inhabitants* of a type, which are phrases, possibly infinite, built from constructors and special constants, and of the patterns that *cover* them. The *constructors* of a type name t are those that the declaration of the custom type with type name t binds with status c (Chapter 5), or, for a type name of the initial basis, those that Appendix C gives it; the constructors of `List` are `[]` and `::`. The inhabitants of a type τ are given by the first of the following clauses that applies:

- if τ is `Int`, `Char` or `String`, each special constant *scon* such that type(*scon*) $\succ \tau$;
- if $\tau = ()$, the phrase `()`;
- if $\tau = (\tau_1, \cdots, \tau_k)\texttt{Tuple}_k$, the phrases `(`$h_1$`, ` $\cdots$ `, `h_k`)`, where each h_i is an inhabitant of τ_i;
- if $\tau = \tau'^{(k)}t$ and t has constructors, the phrases *con* $h_1 \cdots h_n$ (written h_1 `::` h_2 when *con* is `::`), where *con* is a constructor of t whose type scheme σ satisfies $\sigma \succ \tau_1 \to \cdots \to \tau_n \to \tau$, and each h_i is an inhabitant of τ_i;
- otherwise (a function type, a record type, a type variable, or a type whose type name has no constructors), the single phrase `_`.

The clauses are read coinductively: an inhabitant may be infinite, provided each of its parts is given by a clause. So `type V = V V` has the one inhabitant `V (V (V ...))`, and `Never`, whose one constructor `JustOneMore` takes a `Never` (Appendix C), has the one inhabitant `JustOneMore (JustOneMore ...)`. Every type has an inhabitant, and so every constructor of a type gives inhabitants of it, at every instance. Inhabitants are not values: no value has type `Never`. An inhabitant describes what a pattern may be asked to match, which depends only on the constructors of a type. Thus with `r` of type `Result Never Int`, the second rule of `case r of Ok n -> n ; Err e -> never e` covers `Err (JustOneMore ...)` and is not redundant, while with `m` of type `Maybe Never`, `case m of Nothing -> 0` is not exhaustive. Where every type that a match involves has a finite inhabitant, the infinite inhabitants change neither restriction, since a pattern examines only a finite part of an inhabitant. The wildcard `_`, a variable and a record pattern cover every inhabitant. A special constant covers each special constant that denotes the same character or string or, for integer constants, has the same value as an `Int` (Chapter 2), however the two are written. A pattern *longcon* $atpat_1 \cdots atpat_n$ ($n \geq 0$) covers *con* $h_1 \cdots h_n$ when *longcon* is *con* or *modid*.*con* and each $atpat_i$ covers h_i; likewise `[ ]` covers `[]`, and pat_1 `::` pat_2 covers h_1 `::` h_2 when pat_1 covers h_1 and pat_2 covers h_2. The pattern `()` covers `()`, a tuple pattern covers a

tuple component by component, and *pat* `as` *var* and `(`*pat*`)` cover what *pat* covers. The restrictions are:

1. *Irredundancy.* In a match pat_1 `->` exp_1 `;` $\cdots$ `;` pat_n `->` exp_n whose patterns elaborate with type τ, each pat_j must cover some inhabitant of τ that no pat_i with $i < j$ covers.

2. *Exhaustiveness.* In a match as above, every inhabitant of τ must be covered by some pat_i. Likewise, in a pattern binding *atpat* `=` *exp* and in a lambda `\` *atpat* `->` *exp*, the pattern *atpat* must cover every inhabitant of its type. The restrictions are inherited by derived forms: in a function definition *var* $atpat_1 \cdots atpat_n$ `=` *exp*, each $atpat_i$ must be exhaustive by itself. In deciding exhaustiveness, no set of special constants covers every inhabitant of `Int`, `Char` or `String`, whatever the range of `Int` (Appendix E).

Chapter 5

Static Semantics for Modules

This chapter defines the static semantics of Modules. It introduces interfaces, which record what a module makes available to the modules that import it, module environments, port environments and static bases, together with provider maps, which record which imported module supplies each identifier that a module may use. The rules infer that a module elaborates, in a basis, to the part of a basis it contributes and to the environment of its own declarations; on the way they infer what the imports, exposing lists, top-level declarations and type declarations of the module elaborate to. Since Elm modules export top-level names only, there are no signatures, functors or sharing, and no matching of an environment against a specification. The admissible types of ports are defined in Section 5.4.

5.1 Semantic Objects

The simple objects for the static semantics of Modules are exactly as for the Core (Section 4.1). The compound objects are those for the Core, augmented by those in Figure 16.

$$
\begin{aligned}
I \in \text{Interface} &= \text{Env} \\
ME \in \text{ModEnv} &= \text{ModId} \xrightarrow{\text{fin}} \text{Interface} \\
PE \in \text{PortEnv} &= \text{Var} \xrightarrow{\text{fin}} \{\text{out}, \text{in}\} \times \text{Type} \\
B \text{ or } T, ME, PE \in \text{Basis} &= \text{Fin}\,(\text{TyName}) \times \text{ModEnv} \times \text{PortEnv} \\
\pi \in \text{ProvMap} &= \text{LongId} \xrightarrow{\text{fin}} (\text{Fin}\,(\text{ModId}) \setminus \{\emptyset\})
\end{aligned}
$$

Figure 16: Further Compound Semantic Objects

An *interface* I is an environment whose qualified environment is empty. It is what a module *exposes*: the types, values and constructors that another module may name after importing it. Every type that a module declares is either *private*, if $tycon \notin \text{Dom}\,(TE \text{ of } I)$, or exposed. An exposed custom

type is *open* if its constructors belong to Dom (VE of I), and *closed* if none of them does; in both cases $I(tycon)$ is the type structure of the type, constructors included. An exposed type alias is bound in TE of I, and an exposed record alias brings its constructor, of status v, into VE of I. In a type structure (θ, VE) of an interface, VE contains a constructor of status c exactly when the type constructor is a custom type, since a custom type has at least one constructor and an alias has none of status c (Section 5.5).

A *module environment* ME maps the identifier of each module elaborated so far to its interface. A *port environment* PE maps the name of each port to its *direction*, out for an outgoing port and in for an incoming one, and to its *payload type*, the type of the values it carries. A *static basis* $B = T, ME, PE$ consists of the set of type names generated so far, the interfaces of the modules elaborated so far, and the ports they declare; port names are global to a program (Chapter 9).

We write Id for the union of Var $\cup$ Con and TyCon, and LongId for the union of LongVar $\cup$ LongCon and LongTyCon; as in the Core, the unions are disjoint, so a type constructor and a value constructor spelt alike are different members of Id. We let *id* range over Id and *longid* over LongId, and N over Fin (LongId). For an environment E whose qualified environment is empty, Dom E means Dom (TE of E) $\cup$ Dom (VE of E), a subset of Id. For a set $N \subseteq$ Id and a module identifier *modid*, *modid*.N means $\{modid.id\,;\, id \in N\}$.

A *provider map* π maps each identifier that the imports of a module make available, unqualified or qualified, to the set of modules that provide it. An identifier with more than one provider is *ambiguous*.

The operations of projection, injection and modification are as for the Core (Section 4.3). Moreover, for an environment E we write $C \oplus E$ for $C +$ (tynames $E, \emptyset, E$), that is, E of C modified by E and T of C extended with the type names of E; this extends the notation $C \oplus TE$ of the Core.

5.2 Interfaces

The interface that a module exposes is determined by the environment E of its own declarations and by the exposing list of its header. For an environment E whose qualified environment is empty, Expose$(E, explist)$ is defined as follows:

$$
\begin{array}{ll}
\text{Expose}(E, \texttt{(..)}) & = E \\
\text{Expose}(E, \texttt{(}item_1\texttt{, } \cdots\texttt{, } item_n\texttt{)}) & = \text{Expose}(E, item_1) + \cdots \\
 & \qquad + \text{Expose}(E, item_n) \\
\text{Expose}(E, var) & = \{var \mapsto E(var)\} \text{ in Env} \\
\text{Expose}(E, tycon) & = (\{\},\ \{tycon \mapsto (\theta, VE)\},\ VE_{\mathrm{v}}) \\
\text{Expose}(E, tycon\ \texttt{(..)}) & = (\{\},\ \{tycon \mapsto (\theta, VE)\},\ VE)
\end{array}
$$

where, in the last two lines, $E(tycon) = (\theta, VE)$ and $VE_{\mathrm{v}} = \{con \mapsto (\sigma, \mathrm{v})\,;\, VE(con) = (\sigma, \mathrm{v})\}$. It is defined when every identifier of an item is

bound in E, which the syntactic restrictions of a header ensure (Section 3.7); a header exposes no operator. Thus `(..)` exposes every type open and every value, constructor and port; an item *tycon* exposes a custom type closed and a record alias with its constructor; and an item *tycon* `(..)` exposes a custom type open. No condition relates the exposed values to the exposed types: a value may be exposed whose type mentions a private type.

An interface I is *well formed* if, for every $(\theta, VE) \in \mathrm{Ran}\,(TE \text{ of } I)$ and every *con* with $VE(con) = (\sigma, \mathrm{v})$, also $(VE \text{ of } I)(con) = (\sigma, \mathrm{v})$. The rules rely on the following property, which may easily be demonstrated from the rules for type declarations.

> If E is the environment to which the top-level declarations of a module elaborate, and $I = \mathrm{Expose}(E, explist)$ is defined, then I is well formed and $I \subseteq E$ componentwise.

The interfaces of the initial basis (Appendix C) are well formed.

5.3 Provider Maps

The imports of a module are read in order, and each one adds to a provider map. For a module identifier *modid* and sets $N, N' \subseteq \mathrm{LongId}$, the provider map $\pi \uplus_{modid} (N, N')$ is π in which *modid replaces* the providers of the identifiers in N and *joins* the providers of those in N': if $\pi' = \pi \uplus_{modid} (N, N')$, then

$$\pi'(longid) = \begin{cases} \{modid\} & \text{if } longid \in N, \\ \pi(longid) \cup \{modid\} & \text{if } longid \in N' \setminus N, \\ \pi(longid) & \text{if } longid \in \mathrm{Dom}\,\pi \setminus (N \cup N'), \end{cases}$$

where $\pi(longid)$ is taken to be $\emptyset$ when $longid \notin \mathrm{Dom}\,\pi$.

For $modid \in \mathrm{Dom}\,ME$, let $TE_{modid} = TE \text{ of } ME(modid)$ and $VE_{modid} = VE \text{ of } ME(modid)$, except that for the module `List` of the initial basis

$$\begin{aligned} TE_{\texttt{List}} &= TE_0 + TE \text{ of } ME(\texttt{List}), \\ VE_{\texttt{List}} &= VE_0 + VE \text{ of } ME(\texttt{List}), \end{aligned}$$

where $E_0 = (\{\}, TE_0, VE_0)$ is the environment of the type constructor `List` and the list constructors `[]` and `::` given in Appendix C. The environment that a provider map π gives in ME is $\mathrm{env}(ME, \pi) = (QE, TE, VE)$, where

$$\begin{aligned} TE &= \{tycon \mapsto TE_{modid}(tycon)\,;\, \pi(tycon) = \{modid\}\} \\ VE &= \{id \mapsto VE_{modid}(id)\,;\, id \in \mathrm{Var} \cup \mathrm{Con},\ \pi(id) = \{modid\}\} \\ QE &= \{modid' \mapsto (\{\}, TE'_{modid'}, VE'_{modid'})\,;\, modid'.id \in \mathrm{Dom}\,\pi \text{ for some } id\} \end{aligned}$$

and, for each such $modid'$,

$$TE'_{modid'} = \{tycon \mapsto TE_{modid}(tycon)\,;\, \pi(modid'.tycon) = \{modid\}\}$$
$$VE'_{modid'} = \{id \mapsto VE_{modid}(id)\,;\, id \in \mathrm{Var} \cup \mathrm{Con},\ \pi(modid'.id) = \{modid\}\}.$$

Thus an ambiguous identifier is bound in no component of $\mathrm{env}(ME, \pi)$, and a phrase that uses it does not elaborate, while one that does not use it is unaffected. The rules add a module *modid* to $\pi(longid)$ only when the last part of *longid* is bound by TE_{modid} or VE_{modid}, so $\mathrm{env}(ME, \pi)$ is defined for every provider map that arises.

Before any import is read, the type `List` and the list constructors of E_0 are available, provided by the module `List`:

$$\pi_0 = \{\texttt{List} \mapsto \{\texttt{List}\},\ \texttt{[]} \mapsto \{\texttt{List}\},\ \texttt{::} \mapsto \{\texttt{List}\}\},$$

where the first `List` is a type constructor and the others are module identifiers.

Operators are not bound in environments: the type scheme and status of an operator are given by the static operator table $\mathrm{OT}_{\mathrm{STAT}}$ of the initial basis, in every module alike (Section 4.3). For a module identifier *modid*, $\mathrm{ops}(modid) \subseteq \mathrm{Dom}\,\mathrm{OT}_{\mathrm{STAT}}$ is the set of operators that the module *modid* declares. For a module of the initial basis it is given in Appendix C (Figure 40); for every other module it is empty, since no program declares an operator.

The default imports of Figure 11, taken in order, form a phrase of the class Imports, which we write $imports_0$. For any *imports*, we write $imports_0$ *imports* for the import list consisting of the default imports followed by those of *imports*.

5.4 Port Types

The values that may cross a port are those of the types τ for which $\mathrm{Port}(\tau)$ holds. Port is the smallest predicate on types that satisfies the conditions of Figure 17; it holds for no other type. In the figure, `Maybe`, `Array`, `Bytes` and `Value` are the type names of the initial basis for `Maybe`, `Array`, `Bytes.Bytes` and `Json.Encode.Value` (Appendix C), whatever a program binds to type constructors of the same spelling. The type `Json.Decode.Value` is an alias of `Json.Encode.Value`, and so denotes `Value` as well.

Types are semantic objects, in which every alias has been replaced by its expansion (Section 4.5), so an alias of a port type is a port type. Every port type is built from type names of the initial basis alone. Figure 18 states the same predicate by kinds of type, with the reason each excluded kind is excluded; the encoding of the values of each port type as boundary values, which are JSON values in which byte sequences may stand, is given in Section 8.7.

$\mathrm{Port}(\tau)$	if $\tau \in \{\texttt{Int}, \texttt{Float}, \texttt{Bool}, \texttt{String}, \texttt{Bytes}, (), \texttt{Value}\}$
$\mathrm{Port}(\tau\,\texttt{List})$	if $\mathrm{Port}(\tau)$
$\mathrm{Port}(\tau\,\texttt{Maybe})$	if $\mathrm{Port}(\tau)$
$\mathrm{Port}(\tau\,\texttt{Array})$	if $\mathrm{Port}(\tau)$
$\mathrm{Port}((\tau_1, \tau_2)\texttt{Tuple}_2)$	if $\mathrm{Port}(\tau_1)$ and $\mathrm{Port}(\tau_2)$
$\mathrm{Port}((\tau_1, \tau_2, \tau_3)\texttt{Tuple}_3)$	if $\mathrm{Port}(\tau_1)$, $\mathrm{Port}(\tau_2)$ and $\mathrm{Port}(\tau_3)$
$\mathrm{Port}(\{\varrho\})$	if $\mathrm{Port}(\varrho(lab))$ for every $lab \in \mathrm{Dom}\,\varrho$

Figure 17: Port Types

5.5 Inference Rules

As for the Core, the rules of the static semantics of Modules allow sentences of the form

$$A \vdash \mathit{phrase} \Rightarrow A'$$

to be inferred, where in this case A is a basis, a basis paired with a provider map, an environment, a module identifier paired with an interface, a context, or a context paired with a type, and A' is a semantic object or a tuple of semantic objects. The sentence is pronounced "*phrase* elaborates to A' in A". Hypotheses not of this form are side conditions. The convention for options is the same as for the Core static semantics.

The rules use the following notation for phrases. For a type-variable sequence $\mathit{tyvarseq} = \mathit{tyvar}_1 \cdots \mathit{tyvar}_k$ that parametrises a phrase (the *conbind* of a custom-type binding or the *ty* of a type-alias binding), we write $\mathit{tyvarseq} = \xi^{(k)}$ when, for each i, ξ_i is the variable named by tyvar_i (Section 4.1) that is a row variable if tyvar_i occurs in the phrase as the extension variable of a record type expression `{` tyvar_i `|` *tyrow* `}`, and a type variable otherwise. For a type constructor *tycon* we write $\mathit{con} = \mathit{tycon}$ when *con* is spelt as *tycon*. The partial function labs is defined on a type expression *ty* that is a record type expression `{` lab_1 `:` ty_1`,` $\cdots$`,` lab_n `:` ty_n `}` ($n \geq 0$), possibly enclosed in parentheses, and gives the sequence of its labels in the order written:

$$\mathrm{labs}(\mathit{ty}) = \mathit{lab}_1 \cdots \mathit{lab}_n.$$

It is undefined on every other type expression, an extensible record type expression or a type constructor among them.

The type names of a basis are intended to include those of its interfaces. The following property can be demonstrated.

Type	Port?	Reason
`Int`, `Float`, `String`, `Bool`	yes	JSON numbers, strings and Booleans
`Bytes`	yes	byte sequences, which are not JSON values
`()`	yes	JSON `null`
`Json.Encode.Value`, `Json.Decode.Value`	yes	any JSON value, never a byte sequence
`List` t, `Array` t	if t is	JSON arrays
`Maybe` t	if t is	`null` or the value
2- and 3-tuples	if each component is	JSON arrays
closed records	if each field is	JSON objects
`Char`	no	excluded in Standard Elm
function types	no	no JSON representation
type variables, constrained ones included	no	no JSON representation
extensible records	no	unknown fields
custom types other than `Bool` and `Maybe`: `Result`, `Order`, `Never`, `Dict`, `Set`, and every custom type of a program	no	no canonical encoding
`Cmd`, `Sub`, `Task`, and other abstract types of the basis	no	no JSON representation

Figure 18: What can and cannot be sent over a port

Let B be a basis with $\mathrm{tynames}\,(ME \text{ of } B) \subseteq T \text{ of } B$, and let $B \vdash \mathit{module} \Rightarrow (T', ME', PE'), E$ be inferred. Then $T' \cap (T \text{ of } B) = \emptyset$, and $\mathrm{tynames}\,E$, $\mathrm{tynames}\,ME'$ and $\mathrm{tynames}\,PE'$ are subsets of $(T \text{ of } B) \cup T'$.

Moreover, in every instance of rule (57), TE is determined by C and the phrase, up to the choice of the type names in $\mathrm{Ran}\,TE$ that are not in T of C.

Modules

$$\boxed{B \vdash \mathit{module} \Rightarrow B', E}$$

$$\frac{\begin{array}{cc} B, \pi_0 \vdash \mathit{imports}_0\ \mathit{imports} \Rightarrow \pi & C = (T \text{ of } B,\ \emptyset,\ \mathrm{env}(ME \text{ of } B, \pi)) \\ C \vdash \mathit{topdec} \Rightarrow E, PE & E \vdash \mathit{header} \Rightarrow (\mathit{modid}, I) \\ \mathit{modid} \notin \mathrm{Dom}\,(ME \text{ of } B) & T' = \mathrm{tynames}\,E \setminus (T \text{ of } B) \end{array}}{B \vdash \mathit{header}\ \mathit{imports}\ \mathit{topdec} \Rightarrow (T', \{\mathit{modid} \mapsto I\}, PE), E} \tag{46}$$

Comments:

(46) The default imports are read before the imports of the module, so an import written in the module may replace a type that a default import

exposes. The context C holds only what the imports provide. The declarations of the module are added by rule (60) to the unqualified environment and never under a qualifier, so a module cannot refer to its own names by qualification: in a module `M`, `M.x` refers to an import with the qualifier `M`, if there is one. The header is elaborated last, in the environment E of the declarations of the module alone; hence a module never exposes what it imports. The set T' holds every type name that the module generates, private types included, since an exposed value may have a type that mentions a private type. An import names a module of ME of B, and $modid \notin \mathrm{Dom}\,(ME \text{ of } B)$, so no module imports itself and no cycle of imports elaborates; Chapter 9 elaborates the modules of a program in an order in which each follows the modules it imports. Nothing here constrains a value named `main`: only the `main` of the main module of a program is constrained, in Chapter 9.

Headers

$\boxed{E \vdash header \Rightarrow (modid, I)}$

$$\frac{I = \mathrm{Expose}(E, explist)}{E \vdash \langle \texttt{port} \rangle\ \texttt{module}\ modid\ \texttt{exposing}\ explist \Rightarrow (modid, I)} \tag{47}$$

Comments:

(47) Whether a module is a port module matters only to the syntactic restrictions, which allow port declarations in it and require at least one. The exposing list of a header elaborates by the operation Expose of Section 5.2, which the syntactic restrictions make defined; the rules of the groups Exposing Lists and Exposing Items below are for the exposing lists of imports.

Imports

$\boxed{B, \pi \vdash imports \Rightarrow \pi'}$

In rule (48), $modid''$ is $modid'$ if the first option is present and $modid$ if it is absent.

$$\frac{\begin{array}{c}(ME \text{ of } B)(modid) = I \qquad \pi_1 = \pi \uplus_{modid} (\emptyset,\ modid''.\mathrm{Dom}\, I) \\ \langle\!\langle modid, I \vdash explist \Rightarrow (N, N') \rangle\!\rangle \qquad \pi_2 = \pi_1 \ \langle\!\langle \uplus_{modid} (N, N') \rangle\!\rangle \\ B, \pi_2 \vdash imports \Rightarrow \pi'\end{array}}{B, \pi \vdash \texttt{import}\ modid\ \langle \texttt{as}\ modid' \rangle\ \langle\!\langle \texttt{exposing}\ explist \rangle\!\rangle\ imports \Rightarrow \pi'} \tag{48}$$

$$\frac{}{B, \pi \vdash \Rightarrow \pi} \tag{49}$$

Comments:

(48) Every identifier that the module exposes becomes available under the qualifier *modid*″, and the identifiers listed by the exposing list become available unqualified. With `as`, the qualifier is *modid*′ instead of *modid*: after `import Dict as D` the identifier `Dict.empty` is not available through this import. Several imports may share a qualifier; their identifiers are joined under it, and an identifier exposed under one qualifier by two modules is ambiguous. Since the imports are read from left to right, a later import replaces what an earlier one provides for the identifiers in its N. A module that is not in ME of B cannot be imported.

Exposing Lists $\boxed{modid, I \vdash explist \Rightarrow (N, N')}$

$$\frac{}{modid, I \vdash \texttt{(..)} \Rightarrow (\emptyset, \mathrm{Dom}\, I)} \tag{50}$$

$$\frac{modid, I \vdash item_i \Rightarrow (N_i, N_i') \quad (1 \le i \le n)}{modid, I \vdash \texttt{(}item_1\texttt{, } \cdots\texttt{, } item_n\texttt{)} \Rightarrow (N_1 \cup \cdots \cup N_n,\ N_1' \cup \cdots \cup N_n')} \tag{51}$$

Comments:

(50) Everything is joined and nothing replaced, so two imports with `exposing (..)` of modules that both expose `x` make `x` ambiguous. The imports elaborate nevertheless, and only a use of `x` fails to elaborate. The operators of ops(*modid*) are also exposed, in the sense of Section 3.3, but no operator is bound.

Exposing Items $\boxed{modid, I \vdash item \Rightarrow (N, N')}$

$$\frac{var \in \mathrm{Dom}\,(VE \text{ of } I)}{modid, I \vdash var \Rightarrow (\emptyset, \{var\})} \tag{52}$$

$$\frac{op \in \mathrm{ops}(modid)}{modid, I \vdash \texttt{(}op\texttt{)} \Rightarrow (\emptyset, \emptyset)} \tag{53}$$

$$\frac{I(tycon) = (\theta, VE) \qquad N' = \{con\,;\ VE(con) = (\sigma, \mathrm{v}) \text{ for some } \sigma\}}{modid, I \vdash tycon \Rightarrow (\{tycon\}, N')} \tag{54}$$

$$\frac{I(tycon) = (\theta, VE) \qquad VE(con) = (\sigma, \mathrm{c}) \text{ for some } con, \sigma}{modid, I \vdash tycon\ \texttt{(..)} \Rightarrow (\{tycon\},\ \mathrm{Dom}\ VE \cap \mathrm{Dom}\,(VE \text{ of } I))} \tag{55}$$

Comments:

(52) A value is joined with what earlier imports provide, so a value exposed by name by two modules is ambiguous.

(53) The rule only checks that the module declares the operator, and the item binds nothing: the operators exposed to a module are those that may be used in it in infix position (Section 3.3), and the meaning of each is given by $\mathrm{OT}_{\mathrm{STAT}}$ (rule (4)), whatever the module imports.

(54) A type listed by name replaces any type of that name provided by an earlier import: after `import Foo exposing (Maybe)` the type constructor `Maybe` denotes the type of `Foo`, not that of the default import. For a record alias, N' holds its constructor, which is joined; for a custom type it is empty, so the constructors of a type are not imported by this item. A constructor cannot be imported by name: in `import Maybe exposing (Just)`, $I(\texttt{Just})$ is undefined, and no rule applies.

(55) The side condition holds only for a custom type, so `Alias(..)` does not elaborate for an alias. For a type that its module exposes closed, no constructor is in $\mathrm{Dom}\,(VE \text{ of } I)$, so the item elaborates, imports the type and imports no constructors: after `import A exposing (T(..))`, where `A` exposes `T` closed, the constructors of `T` are not available.

Top-level Declarations

$\boxed{C \vdash \mathit{topdec} \Rightarrow E, PE}$

$$\frac{C + VE \vdash \mathit{valbind} \Rightarrow VE \qquad VE' = \mathrm{Clos}_C\, VE}{C \vdash \mathit{valbind} \Rightarrow VE' \text{ in Env},\ \{\}} \tag{56}$$

$$\frac{\begin{array}{c}\langle C \oplus TE \vdash \mathit{datbind} \Rightarrow VE', TE' \qquad \forall (t, VE'') \in \mathrm{Ran}\, TE',\ t \notin T \text{ of } C\rangle \\ C \oplus TE \vdash \mathit{typbind}_i \Rightarrow VE_i, TE_i \quad (1 \le i \le l) \\ TE = \langle TE' + \rangle\, TE_1 + \cdots + TE_l \qquad VE = \langle VE' + \rangle\, VE_1 + \cdots + VE_l \\ \mathrm{tyvars}\ TE = \emptyset\end{array}}{\begin{array}{c}C \vdash \langle \texttt{type } \mathit{datbind}\ \texttt{;}\rangle\ \texttt{type alias } \mathit{typbind}_1\ \texttt{;}\ \cdots \\ \texttt{; type alias } \mathit{typbind}_l \Rightarrow (\{\}, TE, VE),\ \{\}\end{array}} \tag{57}$$

$$\frac{\begin{array}{c}U \text{ is new for } ty \text{ in } C \qquad C + U \vdash ty \Rightarrow \tau \\ \tau = \tau' \to \alpha\,\texttt{Cmd} \qquad \mathrm{Port}(\tau')\end{array}}{C \vdash \texttt{port } var\ \texttt{:}\ ty \Rightarrow \{var \mapsto (\mathrm{Clos}_C\tau, \mathrm{v})\} \text{ in Env},\ \{var \mapsto (\mathrm{out}, \tau')\}} \tag{58}$$

$$\frac{\begin{array}{c}U \text{ is new for } ty \text{ in } C \qquad C + U \vdash ty \Rightarrow \tau \\ \tau = (\tau' \to \alpha) \to \alpha\,\texttt{Sub} \qquad \mathrm{Port}(\tau')\end{array}}{C \vdash \texttt{port } var\ \texttt{:}\ ty \Rightarrow \{var \mapsto (\mathrm{Clos}_C\tau, \mathrm{v})\} \text{ in Env},\ \{var \mapsto (\mathrm{in}, \tau')\}} \tag{59}$$

$$\frac{C \vdash \mathit{topdec}_1 \Rightarrow E_1, PE_1 \qquad C \oplus E_1 \vdash \mathit{topdec}_2 \Rightarrow E_2, PE_2 \quad \text{the separator is not one of a type-declaration phrase}}{C \vdash \mathit{topdec}_1 \ \texttt{;}\ \mathit{topdec}_2 \Rightarrow E_1 + E_2,\ PE_1 + PE_2} \tag{60}$$

Comments:

(56) The rule is that for declarations in the Core (rule (19)): a group of top-level values is recursive, an unannotated member is monomorphic within its group, and the closure makes every top-level value polymorphic in its other uses. At the top level the restriction on recursive values is syntactic and weaker than in a `let` (Section 3.7), so `f = \n -> f n` elaborates here.

(57) The phrase consists of the custom types of a module, as one group, followed by its aliases, as Appendix A arranges them (Section A.2); here $l \geq 0$, and $l \geq 1$ when the option is absent, in which case the phrase begins with the first alias, and when $l = 0$ it is `type` *datbind*. Since TE occurs on both sides, every type constructor of the phrase is in scope in all of it: a constructor may take an argument whose type is an alias declared in the phrase, and an alias may mention a custom type of the phrase or another alias. The equation has a solution determined by C and the phrase up to the choice of new type names (Section 5.5), because the type function of a custom type is its type name alone, and no alias refers to itself through aliases. The side condition on T of C makes the type names of the custom types new. The side condition tyvars $TE = \emptyset$ keeps the equation from justifying itself: the closures of rules (61) and (64) are taken in $C \oplus TE$, so without it an environment whose constructors keep a free variable, as `K : a -> T a` for `type T a = K a`, would be a solution too; with it, the constructors are closed and TE is determined as the property above says. Rule (60) does not split such a phrase (its side condition), so the type declarations of a module are elaborated together, and their order does not matter: a custom type may mention an alias that follows it. A local type spelt as an imported one is a different type, and the unqualified type constructor denotes the local one, in the declarations before it as well as after it: with `type T = T String` followed by `type alias String = Int`, the constructor `T` takes an `Int`.

(58) The variables of the annotation are made explicit, as for an annotated value. The argument of `Cmd` must be a type variable, so `port send : String -> Cmd Msg` does not elaborate, nor does `port send : Cmd msg`, which has no payload, nor `port send : Int -> Int -> Cmd msg`. Since Port(τ') excludes type variables, α is the only variable of τ, and the closure generalises it. The type `Cmd` is that of the initial

basis; a type of a program spelt `Cmd` is a different type. A port is a value of status v, so it is used and exposed like any other value; *PE* records its direction and payload type, by which Chapter 8 gives it its meaning, and Chapter 9 requires the names of the ports of a program to be distinct. Thus `port send : Char -> Cmd msg` does not elaborate, while `port send : List { id : Int, name : String } -> Cmd msg` does.

(59) The variable α occurs in both places, so `port recv : (Int -> a) -> Sub b` does not elaborate. As for an outgoing port, `port recv : (Json.Decode.Value -> msg) -> Sub msg` elaborates, and `port recv : (Result String Int -> msg) -> Sub msg` does not.

(60) Each declaration is elaborated in a context that contains the declarations before it. A type-declaration phrase, whose parts are separated by `;` too, is not split: rule (57) elaborates it as a whole, so that a custom type sees the aliases that follow it. C, a declaration of the module replaces an imported binding of the same identifier, or makes an ambiguous one available. The syntactic restrictions ensure that $\mathrm{Dom}\, E_1 \cap \mathrm{Dom}\, E_2 = \emptyset$ and $\mathrm{Dom}\, PE_1 \cap \mathrm{Dom}\, PE_2 = \emptyset$.

Custom-type Bindings

$\boxed{C \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE}}$

$$\frac{\begin{array}{c}\mathit{tyvarseq} = \xi^{(k)} \qquad U = \{\xi_1, \cdots, \xi_k\} \\ C + U \vdash \mathit{tyvar}_i \Rightarrow \tau_i \quad (1 \le i \le k) \\ C + U, \tau^{(k)} t \vdash \mathit{conbind} \Rightarrow \mathit{VE} \qquad t \in \mathrm{TyName}^{(k)} \\ \langle C \vdash \mathit{datbind}' \Rightarrow \mathit{VE}', \mathit{TE}' \qquad \forall (t', \mathit{VE}'') \in \mathrm{Ran}\, \mathit{TE}',\ t \neq t' \rangle\end{array}}{\begin{array}{c}C \vdash \mathit{tycon}\ \mathit{tyvarseq}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind}' \rangle \Rightarrow \\ \mathrm{Clos}_C\, \mathit{VE}\ \langle +\ \mathit{VE}' \rangle,\ \{\mathit{tycon} \mapsto (t, \mathrm{Clos}_C\, \mathit{VE})\}\ \langle +\ \mathit{TE}' \rangle\end{array}} \tag{61}$$

Comments:

(61) The parameters are made explicit while the constructors elaborate, and are generalised by the closure, since U of $C = \emptyset$ at the top level. A parameter need not occur in *conbind*, so phantom types are allowed. A parameter used as the extension variable of a record type is a row variable; τ_i is then the open record type with no fields of its own, and an argument of t in that position is a record type. The side condition $t \neq t'$ gives the custom types of one group distinct type names; rule (57) makes them new. The syntactic restrictions ensure that $\mathrm{Dom}\, \mathit{VE} \cap \mathrm{Dom}\, \mathit{VE}' = \emptyset$ and $\mathit{tycon} \notin \mathrm{Dom}\, \mathit{TE}'$.

Constructor Bindings

$\boxed{C, \tau \vdash \mathit{conbind} \Rightarrow \mathit{VE}}$

$$\frac{C \vdash \mathit{ty}_i \Rightarrow \tau_i \quad (1 \le i \le n) \qquad \langle C, \tau \vdash \mathit{conbind}' \Rightarrow \mathit{VE} \rangle}{\begin{array}{c} C, \tau \vdash \mathit{con}\ \mathit{ty}_1 \cdots \mathit{ty}_n\ \langle \mid\ \mathit{conbind}' \rangle \Rightarrow \\ \{\mathit{con} \mapsto (\tau_1 \to \cdots \to \tau_n \to \tau, \mathrm{c})\}\ \langle +\ \mathit{VE} \rangle \end{array}} \tag{62}$$

Comments:

(62) When $n = 0$ the type of *con* is τ, so a constructor without arguments is a value of the custom type. The syntactic restrictions ensure $\mathit{con} \notin \mathrm{Dom}\ \mathit{VE}$.

Alias Bindings

$\boxed{C \vdash \mathit{typbind} \Rightarrow \mathit{VE}, \mathit{TE}}$

$$\frac{\mathit{tyvarseq} = \xi^{(k)} \qquad C + \{\xi_1, \cdots, \xi_k\} \vdash \mathit{ty} \Rightarrow \tau \qquad \mathit{ty} \notin \mathrm{Dom\,labs}}{C \vdash \mathit{tycon}\ \mathit{tyvarseq}\ \texttt{=}\ \mathit{ty} \Rightarrow \{\},\ \{\mathit{tycon} \mapsto (\Lambda\xi^{(k)}.\tau, \{\})\}} \tag{63}$$

$$\frac{\begin{array}{c} \mathit{tyvarseq} = \xi^{(k)} \qquad C + \{\xi_1, \cdots, \xi_k\} \vdash \mathit{ty} \Rightarrow \tau \\ \mathrm{labs}(\mathit{ty}) = \mathit{lab}_1 \cdots \mathit{lab}_n \qquad \tau = \{\varrho\} \qquad \varrho = \{\mathit{lab}_1 \mapsto \tau_1, \cdots, \mathit{lab}_n \mapsto \tau_n\} \\ \mathit{con} = \mathit{tycon} \qquad \mathit{VE} = \mathrm{Clos}_C\{\mathit{con} \mapsto (\tau_1 \to \cdots \to \tau_n \to \tau, \mathrm{v})\} \end{array}}{C \vdash \mathit{tycon}\ \mathit{tyvarseq}\ \texttt{=}\ \mathit{ty} \Rightarrow \mathit{VE},\ \{\mathit{tycon} \mapsto (\Lambda\xi^{(k)}.\tau, \mathit{VE})\}} \tag{64}$$

Comments:

(63) An alias generates no type name: it denotes a type function, and a type expression that applies it denotes the expansion (rule (28)). Thus after `type alias Id = Int`, the types `Id` and `Int` are the same type. By the syntactic restrictions the parameters are exactly the type variables of *ty*, so the type function has no unused parameter. Rules (63) and (64) both cover the production; which applies is decided by the form of *ty*. An alias of an extensible record type, or of another alias, defines no constructor, even when that alias is a record alias.

(64) A record alias also defines a constructor, spelt as the alias, whose arguments are the fields in the order written: after `type alias P = { y : Int, x : Int }`, the expression `P 1 2` denotes the record `{ y = 1, x = 2 }`. Its status is v, so it may be applied but may not occur in a pattern (rule (43)). It is exposed and imported together with the alias (rule (54)).

5.6 Principal Derivations

Instances and principal derivations are defined for a module as for a Core phrase (Section 4.10), with one addition: a module generates type names, which a derivation chooses freely outside T of B (rules (61) and (57)). Let D and D' be derivations for a module in a basis B. Then D' is an *instance* of D when, for some bijection ρ from the type names that D generates to those that D' generates, D' is an instance, in the sense of Section 4.10, of the derivation obtained from D by replacing each such type name t by $\rho(t)$; the sets T of corresponding contexts are then equal. A derivation for a module in B is *principal* when every derivation for the module in B is an instance of it.

> Every module that elaborates in a basis B has a principal derivation in B, and any two of its principal derivations differ only by a renaming of type and row variables and of the type names they generate.

A derivation for a module combines derivations for its top-level declarations, whose contexts the rules determine from B and the earlier declarations, up to the type names chosen; so the property follows from that of the Core. The proof is not given here.

5.7 Further Restrictions

The further restrictions of the Core (Section 4.11) apply to the expressions and patterns of every top-level declaration, judged, as in the Core, on the principal derivation for the module (Section 5.6). The following conditions concern a program rather than a module, and Chapter 9 imposes them: the type of `main` in the main module, a `main` in any other module being an ordinary value; the distinctness of the names of the ports of a program from one another and from the names of the managers of the platform (Chapter 8); the absence of any reference of `main` to itself, directly or through a delayed occurrence; and the distinctness of the identifiers of the modules of a program from those of the modules of the initial basis.

Some implementations accept further programs, such as those with a further type of flags for `main`, or reject programs that Standard Elm admits, such as those with `Bytes` in a payload type, or those that exceed limits on the sizes of records, constructors and functions. None of these acceptances or limits is part of Standard Elm; each is recorded in Appendix E.

Chapter 6

Dynamic Semantics for the Core

The dynamic semantics of the Core defines the values that phrases evaluate to, the environments that declarations and patterns produce, and the closures by which functions are represented. Its sentences infer that a phrase evaluates to a value or an environment in a given environment, or that a pattern, matched against a value, gives an environment or fails. Evaluation in Standard Elm has no state and no exceptions: sentences carry no states, and no rule produces an exception packet (Section 6.7). Every phrase evaluated here is assumed to have elaborated in the static semantics of Chapter 4.

6.1 Reduced Syntax

Types play no part in evaluation, so for the dynamic semantics the Core syntax is reduced as follows:

- The type annotation "`:` *ty*" is omitted from every value binding, which thus has the form *var* `=` *exp* ⟨`and` *valbind*⟩ or *atpat* `=` *exp*.
- The Core phrase classes Ty and TyRow are omitted.

6.2 Number Specialisation

Erasing types would leave the value of an integer constant undetermined when its type is Float or a type variable of class number. The dynamic semantics of every level is therefore given for the elaborated program after *number specialisation*, a translation of the whole program, which we define here, where the dynamic semantics begins. It starts from the principal derivation of the program (Section 4.10), with the following default.

Defaulting of number. In the principal derivation, each type variable of class number that is bound neither by a closure (rules (19), (56) and (61)), nor

by the quantifier $\forall U'$ of an annotated binding (rule (20)), nor as a parameter of a type declaration (rules (61), (63) and (64)), is taken to be `Float`: `Float` is substituted for it throughout the derivation, which remains a derivation for the program, since `Float` $\in \mathrm{Adm}_{\mathrm{number}}$. Every constant of such a type is an integer constant, since a floating-point constant has the type `Float` itself, and every integer constant may have the type `Float`. A variable of class number bound in any of these ways is not affected: one bound by a closure or an annotation is quantified in the type scheme of a value, and the copies below assign it; a parameter of a type declaration is bound by a type function, whose applications substitute for it. Since closure never generalises an explicit variable in scope (Section 4.8), an explicit variable is bound by the annotation that introduces it, and so is never defaulted, even where a nested binding uses it: with `f : number -> number` and `f x = let g y = y + x in g 1`, the `number` of `x` is bound by the annotation of `f`, not by the closure of `g`. The default changes no type that the program forces, and so no type a program elaborates with: it decides only types the principal derivation leaves open.

Then each declaration of values, of a `let` or at the top level of a module, whose bindings have type schemes that quantify type variables of class number is replaced by the following copies of it:

- the copy that assigns `Float` to each of those variables, the *default copy*, which binds the variables of the declaration under their own names; and
- one copy for each other assignment of `Int` or `Float` to those variables at which the program uses the declaration, the copies binding distinct new variables.

A copy renames the variables the declaration binds, the variables of its pattern rows among them (Section 2.10), and never a label: the copy at `Int` of the destructuring binding `{ h } = { h = \x -> x - 3 }`, whose equivalent form is `{ h = h } = ...`, is `{ h = h' } = ...`, which binds a new variable `h'` to the field `h` of the same record. A copy is not written in a program, so the syntactic restrictions do not judge it; it keeps every property they give a program, except that the variable of a pattern row need not be spelt as its label, which the restriction on pattern rows requires only of a program. Each occurrence of a variable the declaration binds is made to refer to the copy for the instance at which that occurrence was elaborated. Every binding of a declaration quantifies all the variables of class number of the declaration, those its own type does not mention included (Section 4.8), so an instance assigns a type to each of them; a variable that nothing at the occurrence fixes is `Float`, as a variable left free in the program is by the default above. A declaration whose bindings quantify no type variable of class number is not copied, and is its own default copy. Since

the default copy is made whether or not the program uses the declaration at that instance, every declaration has a copy: in particular every top-level value of a module, including `main` and every value that no phrase uses, has a copy and is evaluated when its module is evaluated. The default copy of the declaration of `main` is the one that Chapter 9 names by Dflt , at the instance Dflt σ of the scheme σ of `main`. Copies are made from the outside in, so that a variable of class number that a declaration does not quantify is already fixed by the copy of the declaration that encloses it, and a variable of class number left free in the program has been taken to be `Float` by the default above. With `check h = if h == 0 then h else Debug.todo "h"` and `x = check (2 ^ -1)` used only as `x // 2`, the copy of `x` at `Int`, where `2 ^ -1` is 0 (Appendix D), is 0; but the default copy, at `Float`, where it is 0.5, has no value, so the program has none. A variable of class number that the type of `x` does not mention, such as that of `2 ^ -1` in `if 2 ^ -1 == 0 then 1 else 2`, is not a variable of the declaration: no closure generalises it, and it is `Float` in every copy. After specialisation every integer constant has the type `Int` or `Float`. The specialisation is finite: $\mathrm{Adm}_{\mathrm{number}}$ contains only `Int` and `Float` (Section 4.4), so a declaration with k variables of class number has at most 2^k copies, even when a binding is used recursively at an instance other than its own.

Where the type of an occurrence of an integer constant matters below, it is its type after specialisation.

6.3 Simple Objects

The dynamic semantics builds its objects from the identifier classes of Chapter 2 and the simple object classes of Figure 19.

sv	$\in$ SVal	special values
b	$\in$ BasVal	basic values
con	$\in$ Con	constructor names
	{FAIL}	failure

Figure 19: Simple Semantic Objects

SVal is the class of *special values*: the integers of the Int range IntRange, the floating-point numbers (among them the two infinities and NaN), the characters, and the strings, which are finite sequences of characters. The range IntRange and the representation of characters are implementation parameters, given in Appendix E. Each occurrence of a special constant *scon* denotes a special value, written val(*scon*). An integer constant spells an integer n, and its value depends on the type of the occurrence after number specialisation (Section 6.2): at `Int`, val(*scon*) = norm(n), where norm maps an integer into IntRange by the same normalisation as Int arithmetic (Section E.1);

at `Float`, val(*scon*) is the floating-point number nearest to n, rounded as Appendix D prescribes. A floating-point constant denotes the real number it spells, rounded to the nearest floating-point number; a character constant denotes a character, and a string constant a sequence of characters, as explained in Chapter 2.

BasVal is the class of *basic values*: the names of the operations of the initial basis, listed in Appendix D. Each basic value b has an *arity* $\text{arity}(b) \geq 1$, the number of arguments it takes before it acts (Section 6.5).

The class Con of value constructors serves also as the class of *constructor names*. We take Con to contain, besides the constructors spelt by identifiers, the two list constructor names `[]` and `::`, which no identifier spells. Each constructor *con* has an *arity* $\text{arity}(con) \geq 0$, the number of arguments it takes: for a constructor of a custom type, the number of types that follow it in its declaration (Chapter 3); $\text{arity}(\texttt{[]}) = 0$ and $\text{arity}(\texttt{::}) = 2$. Constructors of different custom types may be spelt alike and have different arities; $\text{arity}(con)$ always means the arity of *con* in the custom type of the value at hand, which elaboration determines. Since only phrases that elaborate are evaluated, a constructor value is only ever compared with the constructors of its own custom type, whose names are distinct, so no confusion arises.

FAIL is the outcome of matching a value against a pattern that does not fit it. It is not a value: it serves only within the rules, to record that a match has not succeeded.

6.4 Compound Objects

The compound objects of the dynamic semantics are shown in Figure 20. The classes Val, Closure, Env, QualEnv and ValEnv are defined together; they are the smallest classes closed under the constructions of the figure.

$$
\begin{aligned}
v \in \text{Val} &= \{()\} \cup \text{SVal} \cup \text{BasApp} \cup \text{ConVal} \cup \text{Record} \\
&\quad \cup \text{Tuple} \cup \text{Closure} \cup \text{EffVal} \cup \text{JVal} \\
(b, v_1 \cdots v_k) \in \text{BasApp} &= \textstyle\bigcup_{k \geq 0} \text{BasVal} \times \text{Val}^k \\
con\ v_1 \cdots v_k \in \text{ConVal} &= \textstyle\bigcup_{k \geq 0} \text{Con} \times \text{Val}^k \\
r \in \text{Record} &= \text{Lab} \xrightarrow{\text{fin}} \text{Val} \\
(v_1, v_2\langle, v_3\rangle) \in \text{Tuple} &= \text{Val}^2 \cup \text{Val}^3 \\
(atpat, exp, E, VE) \in \text{Closure} &= \text{AtPat} \times \text{Exp} \times \text{Env} \times \text{ValEnv} \\
(QE, VE) \text{ or } E \in \text{Env} &= \text{QualEnv} \times \text{ValEnv} \\
QE \in \text{QualEnv} &= \text{ModId} \xrightarrow{\text{fin}} \text{Env} \\
VE \in \text{ValEnv} &= (\text{Var} \cup \text{Con}) \xrightarrow{\text{fin}} \text{Val}
\end{aligned}
$$

Figure 20: Compound Semantic Objects

The value () is the unit value. A member $(b, v_1 \cdots v_k)$ of BasApp is a basic value applied to the arguments it has received so far, and always

has $k <$ arity(b); we identify (b) with b, so that BasVal $\subseteq$ BasApp. A member *con* $v_1 \cdots v_k$ of ConVal is a *constructor value*, and always has $k \leq$ arity(*con*): when $k =$ arity(*con*) it is complete, and when $k <$ arity(*con*) it is a constructor awaiting further arguments. A constructor name *con* with $k = 0$ is itself a constructor value. We write v_1 `::` v_2 for the constructor value `::` v_1 v_2; the lists are thus the constructor values `[]` and v_1 `::` v_2 with v_2 a list. Records are finite maps from labels to values; the empty record is {}. A tuple has two or three components. The classes EffVal of effect values (task, command, subscription, process-identifier and program values) and JVal of JSON values are defined in Chapter 8; this chapter uses only the fact that their members are values.

Projection, injection and modification are as defined for the static semantics (Chapter 4); the convention that a decorated metavariable ranges over the class of the undecorated one is stated in Chapter 1. The unions in Figure 20 are disjoint unions, and the classes defined there are disjoint from one another; the injections into Val are left implicit. An environment E is modified by a value environment as $E + VE$, meaning $E + (\{\}, VE)$. The names E, VE, Env and the like denote dynamic objects here and static objects in Chapter 4; the two semantics are presented apart, so the reuse is harmless.

A long identifier is looked up in an environment $E = (QE, VE)$ as follows:

$$E(var) = VE(var), \qquad E(modid\,.\,var) = (QE(modid))(var),$$

and likewise for constructors. The identifiers that the equivalent forms of derived forms introduce (Appendix A) are looked up in the same way. Because only an import of the module `Basics` provides the qualifier `Basics` (Section 3.7), `Basics.True`, `Basics.False` and `Basics.negate` always denote the values of the initial dynamic basis (Appendix D); the constructor `::` is bound in the initial environment E_0 of that appendix, from which every module is evaluated. Operators are not looked up in E, and are not in the domain of a value environment. Since no program declares an operator, every operator is one of the initial basis, and its value is the same in every environment: the value $\mathrm{OT}_{\mathrm{DYN}}(op)$ that the dynamic operator table of Appendix D (Figure 46) gives it. For `::` this value is the list constructor `::`.

6.5 Basic Values

The meaning of the basic values is represented by a function APPLY which, for a basic value b of arity n and values $v_1, \ldots, v_n$, gives a finite set of values:

$$\mathrm{APPLY}(b, v_1 \cdots v_n) \in \mathrm{Fin\,Val}.$$

APPLY is defined in Appendix D (Section D.4), which also gives the arity of every basic value. Equality and the order relations of the initial basis are

basic values, and their meaning on structured values is defined there, not in this chapter.

APPLY gives a set, rather than a value, so that the definition is exact about the two operations of the initial basis that do not determine one result. For every basic value the set has exactly one member, with two exceptions. First, when an equality test reaches two functions its result is an unspecified Bool, so that the set may hold both `True` and `False`; evaluation is then non-deterministic. Second, for the basic value b bound to `Debug.todo`, $\mathrm{APPLY}(b, v)$ is empty, so that an application of `Debug.todo` has no result. Every other basic value gives a result for every argument: arithmetic gives a defined value even where no mathematically correct one exists, and `Debug.log` returns its second argument, any output it makes lying outside the definition. Thus no evaluation ends in a run-time error: a phrase either has a value or has none.

6.6 Function Closures

A *function closure* $(atpat, exp, E, VE)$ is applied to a value v by matching v against *atpat* and evaluating *exp* in the resulting environment. The environment is not $E + VE$ but $E + \mathrm{Rec}\ VE$, where the domain of the fourth component VE holds the variables treated recursively, and

$$\mathrm{Rec} : \mathrm{ValEnv} \to \mathrm{ValEnv}$$

is defined by:

- $\mathrm{Dom}\,(\mathrm{Rec}\ VE) = \mathrm{Dom}\ VE$;
- if $VE(var) \notin \mathrm{Closure}$, then $(\mathrm{Rec}\ VE)(var) = VE(var)$;
- if $VE(var) = (atpat, exp, E, VE')$, then $(\mathrm{Rec}\ VE)(var) = (atpat, exp, E, VE)$.

Thus Rec VE gives each closure in Ran VE the bindings of VE afresh, so that a recursive application made during the evaluation of *exp* finds them; the recursion is unwound one step at a time, and every semantic object stays finite. Rec is used in two places in the rules: for a recursive declaration, and for the application of a closure.

A declaration is recursive when one of its bindings refers to a variable it binds. For a value binding *valbind*, $\mathrm{FV}(valbind)$ is the set of variables that occur free in the right-hand sides of the bindings of *valbind*, with "free" and "right-hand side" as defined in Section A.1; the variables bound by *valbind* are the domain of the value environment it evaluates to. By the restrictions of Section 2.10, every binding of a recursive declaration has the form *var* `=` `\` *atpat* `->` *exp*.

6.7 Inference Rules

The rules infer sentences of the forms

$$E \vdash \mathit{phrase} \Rightarrow A, \qquad E, v \vdash \mathit{phrase} \Rightarrow A, \qquad \vdash v_1\, v_2 \Rightarrow v$$

where E is an environment, v is a value against which the phrase is matched (r, a record, for pattern rows), and A is the result: a value, a record, a value environment, or FAIL (a pattern row gives a value environment, never FAIL). The first form is pronounced "*phrase* evaluates to A in E", the second "*phrase*, matched against v, gives A in E", and the third "v_1 applied to v_2 gives v". The hypotheses of a rule that are not sentences are its *side conditions*. The convention for options is the same as for the Core static semantics: in each instance of a rule, the phrases in first options $\langle\ \rangle$ are either all present or all absent.

Sentences have no states, and there are no exception packets: the Core has neither assignable references nor exceptions. Hence no state convention and no exception convention is needed, and every rule is given in full. In particular the order of the premises of a rule carries no meaning. A rule says which results the parts of a phrase must have, and the conclusion holds only when all of them do; since nothing but the result is observable, an implementation may evaluate the parts of a phrase in any order. Only the parts that some premise names are evaluated: the body of a lambda is evaluated when the closure is applied, and a `case` evaluates the body of the chosen match rule only.

We allow compound variables, built from the variables of Figures 19 and 20 with "/", to range over unions of classes: v/FAIL ranges over Val $\cup$ {FAIL} and VE/FAIL over ValEnv $\cup$ {FAIL}. We extend modification to failure:

$$VE + \text{FAIL} \;=\; \text{FAIL} + VE \;=\; \text{FAIL} + \text{FAIL} \;=\; \text{FAIL}.$$

A phrase has no result in E when no sentence with it on the left of $\Rightarrow$ can be inferred. This is so when its evaluation does not terminate, and when it needs an application of `Debug.todo`. The rules place no bound on the resources evaluation uses; an implementation that stops for lack of resources is outside the definition.

Atomic Expressions

$\boxed{E \vdash \mathit{atexp} \Rightarrow v}$

$$\frac{}{E \vdash \mathit{scon} \Rightarrow \text{val}(\mathit{scon})} \tag{65}$$

$$\frac{E(\mathit{longvar}) = v}{E \vdash \mathit{longvar} \Rightarrow v} \tag{66}$$

$$\frac{E(\mathit{longcon}) = v}{E \vdash \mathit{longcon} \Rightarrow v} \tag{67}$$

$$\frac{}{E \vdash \texttt{(}op\texttt{)} \Rightarrow \mathrm{OT}_{\mathrm{DYN}}(op)} \tag{68}$$

$$\frac{}{E \vdash \texttt{()} \Rightarrow ()} \tag{69}$$

$$\frac{E \vdash exp_1 \Rightarrow v_1 \qquad E \vdash exp_2 \Rightarrow v_2 \qquad \langle E \vdash exp_3 \Rightarrow v_3 \rangle}{E \vdash \texttt{(}exp_1\texttt{,}\ exp_2\ \langle\texttt{,}\ exp_3\rangle\texttt{)} \Rightarrow (v_1, v_2\langle, v_3\rangle)} \tag{70}$$

$$\frac{}{E \vdash \texttt{[ ]} \Rightarrow \texttt{[]}} \tag{71}$$

$$\frac{\langle E \vdash exprow \Rightarrow r \rangle}{E \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + r\rangle} \tag{72}$$

$$\frac{E(var) = r \qquad E \vdash exprow \Rightarrow r'}{E \vdash \texttt{\{}\ var\ \texttt{|}\ exprow\ \texttt{\}} \Rightarrow r + r'} \tag{73}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash \texttt{(}exp\texttt{)} \Rightarrow v} \tag{74}$$

$$\frac{E \vdash dec \Rightarrow VE \qquad E + VE \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp \Rightarrow v} \tag{75}$$

Comments:

(65) The value of an integer constant depends on the type of its occurrence after number specialisation (Section 6.2). Thus, with `x : Float` defined by `x = 2`, the constant denotes a floating-point number and `x ^ -1` evaluates to `0.5`; and with `inc n = n + 1`, the application `inc 0.5` uses the default copy of `inc`, at `Float`, in which `1` denotes a floating-point number, and evaluates to `1.5`, while `inc (7 // 2)` uses its copy at `Int` and evaluates to the Int value 4.

(66) As in the static semantics, long variables are looked up in the environment. The identifier status of the static semantics plays no part.

(67) A constructor of a custom type is bound to the constructor value *con*, awaiting all its arguments; the value bound to the constructor of a record type alias is given in Chapter 7.

(68) The value of an operator does not depend on E, so a variable that a program declares cannot capture an operator: a top-level `add` of a module leaves `(+)` unchanged. The operator values `(&&)` and `(||)` denote functions of the initial basis, which receive both arguments as values: in `(&&) False e` the expression `e` is evaluated. Only the infixed forms `e1 && e2` and `e1 || e2`, which are derived forms (Appendix A), evaluate their second operand conditionally.

(73) By elaboration, $\mathrm{Dom}\, r' \subseteq \mathrm{Dom}\, r$, so the result has the fields of r, with the values of the fields named in *exprow* replaced. The record bound to *var* is unaffected, there being no state.

(75) Every binding of *dec* is evaluated, whether or not *exp* uses it: `let _ = Debug.todo "x" in 1` has no value. A `let` binds a single group of bindings (Appendix A), so the evaluation of the groups of a `let` as written is ordered by the nesting of the equivalent form, and within a group the order does not matter.

Expression Rows

$\boxed{E \vdash \mathit{exprow} \Rightarrow r}$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad \langle E \vdash \mathit{exprow} \Rightarrow r \rangle}{E \vdash \mathit{lab}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{,}\ \mathit{exprow} \rangle \Rightarrow \{\mathit{lab} \mapsto v\} \langle + r \rangle} \tag{76}$$

Comments:

(76) The fields are evaluated in no specified order, neither as written nor by label (Section 6.7).

Expressions

$\boxed{E \vdash \mathit{exp} \Rightarrow v}$

$$\frac{E \vdash \mathit{atexp} \Rightarrow v}{E \vdash \mathit{atexp} \Rightarrow v} \tag{77}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v_1 \qquad E \vdash \mathit{atexp} \Rightarrow v_2 \qquad \vdash v_1\, v_2 \Rightarrow v}{E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow v} \tag{78}$$

$$\frac{}{E \vdash \backslash\ \mathit{atpat}\ \texttt{->}\ \mathit{exp} \Rightarrow (\mathit{atpat}, \mathit{exp}, E, \{\})} \tag{79}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad E, v \vdash \mathit{match} \Rightarrow v'}{E \vdash \texttt{case}\ \mathit{exp}\ \texttt{of}\ \mathit{match} \Rightarrow v'} \tag{80}$$

Comments:

(78) The argument is evaluated whether or not the function uses it: `(\_ -> 1) (Debug.todo "x")` has no value.

(79) The fourth component of the closure is empty because a lambda introduces no recursively defined values; rule (91) supplies it for a recursive declaration. The closure holds the environment E of this evaluation, so closures made in successive calls of a function hold the values of their own calls. Two evaluations of one lambda give closures that no rule distinguishes from one another by identity.

(80) There is no rule in which *match* gives FAIL: the static semantics requires every match of a `case` to be exhaustive (Chapter 4), so for a value of the right type some match rule applies.

Application of Values

$\boxed{\vdash v_1\, v_2 \Rightarrow v}$

$$\frac{E + \text{Rec } VE, v \vdash atpat \Rightarrow VE' \qquad E + \text{Rec } VE + VE' \vdash exp \Rightarrow v'}{\vdash (atpat, exp, E, VE)\, v \Rightarrow v'} \tag{81}$$

$$\frac{}{\vdash (con\ v_1 \cdots v_k)\, v \Rightarrow con\ v_1 \cdots v_k\ v} \tag{82}$$

$$\frac{k + 1 < \text{arity}(b)}{\vdash (b, v_1 \cdots v_k)\, v \Rightarrow (b, v_1 \cdots v_k\, v)} \tag{83}$$

$$\frac{k + 1 = \text{arity}(b) \qquad v' \in \text{APPLY}(b, v_1 \cdots v_k\, v)}{\vdash (b, v_1 \cdots v_k)\, v \Rightarrow v'} \tag{84}$$

Comments:

(81) There is no rule in which *atpat*, matched against v, gives FAIL: the static semantics requires the pattern of a lambda to be exhaustive (Chapter 4). Were it to fail, no rule would apply.

(82) No side condition $k < \text{arity}(con)$ is needed: in a program that elaborates a constructor value is applied only while it awaits arguments. Thus `Just` applied to `1` gives the complete constructor value `Just 1`.

(84) The result is any member of the set $\text{APPLY}(b, v_1 \cdots v_k\, v)$ (Section 6.5). When the set has two members the rule does not determine the result: this happens only when an equality test reaches two functions, as in `(\x -> x) == (\x -> x)`, which may evaluate to `True` or to `False`. When the set is empty, as for `Debug.todo`, the rule never applies, and the application has no result.

No rule applies the unit value, a special value, a complete constructor value, a record, a tuple or an effect value: in a program that elaborates the function of an application is always a closure, a constructor value awaiting arguments, or a basic value awaiting arguments. The rules of this group are the only definition of application; Chapters 8 and 9 use the sentence $\vdash v_1\, v_2 \Rightarrow v$ to apply function values outside any phrase.

Matches

$\boxed{E, v \vdash match \Rightarrow v'/\text{FAIL}}$

$$\frac{E, v \vdash mrule \Rightarrow v'}{E, v \vdash mrule\ \langle ;\ match \rangle \Rightarrow v'} \tag{85}$$

$$\frac{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL}}{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL}} \tag{86}$$

$$\frac{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL} \qquad E, v \vdash \mathit{match} \Rightarrow v'/\mathrm{FAIL}}{E, v \vdash \mathit{mrule}\ \texttt{;}\ \mathit{match} \Rightarrow v'/\mathrm{FAIL}} \tag{87}$$

Comments:

(85) The value being matched stands to the left of the turnstile. The match rules are tried in the order written, and the first whose pattern matches gives the result; the match rules after it are not evaluated, even when their patterns would also match. This order is part of the meaning, and is unaffected by the freedom of evaluation order (Section 6.7), since a rule for a later match rule requires every earlier one to give FAIL.

Match Rules

$\boxed{E, v \vdash \mathit{mrule} \Rightarrow v'/\mathrm{FAIL}}$

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE} \qquad E + \mathit{VE} \vdash \mathit{exp} \Rightarrow v'}{E, v \vdash \mathit{pat}\ \texttt{->}\ \mathit{exp} \Rightarrow v'} \tag{88}$$

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathrm{FAIL}}{E, v \vdash \mathit{pat}\ \texttt{->}\ \mathit{exp} \Rightarrow \mathrm{FAIL}} \tag{89}$$

Comments:

(89) The body *exp* is not evaluated, so the result is FAIL even when *exp* has no value.

Declarations

$\boxed{E \vdash \mathit{dec} \Rightarrow \mathit{VE}}$

$$\frac{E \vdash \mathit{valbind} \Rightarrow \mathit{VE} \qquad \mathrm{Dom}\ \mathit{VE} \cap \mathrm{FV}(\mathit{valbind}) = \emptyset}{E \vdash \mathit{valbind} \Rightarrow \mathit{VE}} \tag{90}$$

$$\frac{E \vdash \mathit{valbind} \Rightarrow \mathit{VE} \qquad \mathrm{Dom}\ \mathit{VE} \cap \mathrm{FV}(\mathit{valbind}) \neq \emptyset}{E \vdash \mathit{valbind} \Rightarrow \mathrm{Rec}\ \mathit{VE}} \tag{91}$$

Comments:

(90) Rules (90) and (91) cover the one production, distinguished by whether the declaration is recursive (Section 6.6). The right-hand sides are evaluated in E, without the variables the declaration binds; in a recursive declaration every right-hand side is a lambda, and Rec gives each of its closures the bindings of the whole declaration. Rec is not applied to a declaration that is not recursive, since a closure it binds may have been made elsewhere, in an environment where a variable spelt as one

that the declaration binds denotes something else. A `let` declaration needs only Rec because every binding of a recursive `let` declaration is a function (Section 2.10); top-level declarations, whose recursive members need not be functions, are treated in Chapter 7.

Value Bindings

$\boxed{E \vdash \mathit{valbind} \Rightarrow \mathit{VE}}$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad \langle E \vdash \mathit{valbind} \Rightarrow \mathit{VE} \rangle}{E \vdash \mathit{var}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \Rightarrow \{\mathit{var} \mapsto v\} \langle + \mathit{VE} \rangle} \tag{92}$$

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}}{E \vdash \mathit{atpat}\ \texttt{=}\ \mathit{exp} \Rightarrow \mathit{VE}} \tag{93}$$

Comments:

(93) There is no rule in which *atpat*, matched against v, gives FAIL: the static semantics requires the pattern of a value binding to be exhaustive (Chapter 4).

Atomic Patterns

$\boxed{E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}}$

$$\frac{}{E, v \vdash \texttt{_} \Rightarrow \{\}} \tag{94}$$

$$\frac{v = \text{val}(\mathit{scon})}{E, v \vdash \mathit{scon} \Rightarrow \{\}} \tag{95}$$

$$\frac{v \neq \text{val}(\mathit{scon})}{E, v \vdash \mathit{scon} \Rightarrow \text{FAIL}} \tag{96}$$

$$\frac{}{E, v \vdash \mathit{var} \Rightarrow \{\mathit{var} \mapsto v\}} \tag{97}$$

$$\frac{E(\mathit{longcon}) = \mathit{con} \qquad v = \mathit{con}}{E, v \vdash \mathit{longcon} \Rightarrow \{\}} \tag{98}$$

$$\frac{E(\mathit{longcon}) = \mathit{con} \qquad v \neq \mathit{con}}{E, v \vdash \mathit{longcon} \Rightarrow \text{FAIL}} \tag{99}$$

$$\frac{}{E, () \vdash \texttt{()} \Rightarrow \{\}} \tag{100}$$

$$\frac{\begin{array}{c} v = (v_1, v_2\langle, v_3\rangle) \qquad E, v_1 \vdash \mathit{pat}_1 \Rightarrow \mathit{VE}_1/\text{FAIL} \\ E, v_2 \vdash \mathit{pat}_2 \Rightarrow \mathit{VE}_2/\text{FAIL} \qquad \langle E, v_3 \vdash \mathit{pat}_3 \Rightarrow \mathit{VE}_3/\text{FAIL} \rangle \end{array}}{E, v \vdash \texttt{(}\mathit{pat}_1\texttt{,}\ \mathit{pat}_2\ \langle\texttt{,}\ \mathit{pat}_3\rangle\texttt{)} \Rightarrow \mathit{VE}_1/\text{FAIL} + \mathit{VE}_2/\text{FAIL} \langle + \mathit{VE}_3/\text{FAIL} \rangle} \tag{101}$$

$$\frac{v = \texttt{[]}}{E, v \vdash \texttt{[ ]} \Rightarrow \{\}} \tag{102}$$

$$\frac{v \neq \texttt{[]}}{E, v \vdash \texttt{[ ]} \Rightarrow \text{FAIL}} \tag{103}$$

$$\frac{v = r \qquad \langle E, r \vdash \mathit{patrow} \Rightarrow \mathit{VE} \rangle}{E, v \vdash \texttt{\{}\ \langle \mathit{patrow} \rangle\ \texttt{\}} \Rightarrow \{\} \langle + \mathit{VE} \rangle} \tag{104}$$

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}}{E, v \vdash \texttt{(}\mathit{pat}\texttt{)} \Rightarrow \mathit{VE}/\text{FAIL}} \tag{105}$$

Comments:

(95) A constant in a pattern is an integer, character or string constant, never a floating-point one, so the test $v = \text{val}(\mathit{scon})$ is identity of special values. An integer constant matches the normalised integer it denotes.

(97) A variable pattern always binds: constructors and variables are spelt differently (Chapter 2), so no identifier status is consulted.

(100) For rules (100), (101) and (104): in a program that elaborates the value matched has the form the pattern requires, so no rule for these patterns gives FAIL on its own account; a tuple fails only when a component pattern fails. A record pattern matches a record with any further fields.

Pattern Rows

$\boxed{E, r \vdash \mathit{patrow} \Rightarrow \mathit{VE}}$

$$\frac{\mathit{lab} \in \text{Dom}\, r \qquad \langle E, r \vdash \mathit{patrow} \Rightarrow \mathit{VE} \rangle}{E, r \vdash \mathit{lab}\ \texttt{=}\ \mathit{var}\ \langle\texttt{,}\ \mathit{patrow}\rangle \Rightarrow \{\mathit{var} \mapsto r(\mathit{lab})\} \langle + \mathit{VE} \rangle} \tag{106}$$

Comments:

(106) The pattern row binds its variable to the field *lab* of r; in a row written as a label alone the variable is spelt as the label (Appendix A). The side condition $\mathit{lab} \in \text{Dom}\, r$ makes the rule apply exactly when r has the field; by elaboration it always holds, since a record of the type the pattern requires has every field the pattern row names. Hence a pattern row cannot fail, and its result is a value environment, never FAIL, as rule (104) expects.

Patterns

$\boxed{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\text{FAIL}}$

$$\frac{E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}}{E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\text{FAIL}} \tag{107}$$

$$\frac{E(longcon) = con \qquad v = con\ v_1 \cdots v_n \qquad E, v_1 \vdash atpat_1 \Rightarrow VE_1/\text{FAIL} \quad \cdots \quad E, v_n \vdash atpat_n \Rightarrow VE_n/\text{FAIL}}{E, v \vdash longcon\ atpat_1 \cdots atpat_n \Rightarrow VE_1/\text{FAIL} + \cdots + VE_n/\text{FAIL}} \tag{108}$$

$$\frac{E(longcon) = con \qquad v \notin \{con\} \times \text{Val}^n}{E, v \vdash longcon\ atpat_1 \cdots atpat_n \Rightarrow \text{FAIL}} \tag{109}$$

$$\frac{v = v_1 :: v_2 \qquad E, v_1 \vdash pat_1 \Rightarrow VE_1/\text{FAIL} \qquad E, v_2 \vdash pat_2 \Rightarrow VE_2/\text{FAIL}}{E, v \vdash pat_1 :: pat_2 \Rightarrow VE_1/\text{FAIL} + VE_2/\text{FAIL}} \tag{110}$$

$$\frac{v = \texttt{[]}}{E, v \vdash pat_1 :: pat_2 \Rightarrow \text{FAIL}} \tag{111}$$

$$\frac{E, v \vdash pat \Rightarrow VE/\text{FAIL}}{E, v \vdash pat\ \texttt{as}\ var \Rightarrow VE/\text{FAIL} + \{var \mapsto v\}} \tag{112}$$

Comments:

(108) Here $n \geq 1$, and by elaboration $n = \text{arity}(con)$. All the argument patterns are matched, and the result is FAIL when any of them fails. Since matching has no effect and always has a result, the order in which the arguments are matched is immaterial.

(109) Rules (96), (99), (103), (109) and (111) are the only rules that give FAIL without a premise giving FAIL; every other FAIL comes from one of them.

(112) By the syntactic restrictions, *var* is not bound by *pat*, so the modification adds a binding and replaces none.

Chapter 7

Dynamic Semantics for Modules

The dynamic semantics of Modules defines the dynamic interfaces that modules evaluate to, and the environments that imports and top-level declarations produce. Its sentences infer that a module evaluates to an interface in a basis, that a sequence of imports evaluates to an environment, that an exposing list cuts an interface down to the part it exposes, and that a top-level declaration evaluates to an interface in an environment. Modules are flat: a module makes its top-level values and constructors available by name, and there are no signatures or functors. Every module evaluated here is assumed to have elaborated in the static semantics of Chapter 5. The order in which the modules of a program are evaluated, and the meaning of `main`, are given in Chapter 9.

7.1 Reduced Syntax

Types play no part in evaluation, except that a port must know what it sends or receives. For the dynamic semantics the syntax of Modules is reduced as follows, in addition to the reduction of the Core (Section 6.1):

- The type annotations of top-level values are omitted, as in the Core. Since a module has no pattern bindings at the top level (Section 3.7), every top-level value declaration has the form

$$var_1 \texttt{ = } exp_1 \texttt{ and } \cdots \texttt{ and } var_n \texttt{ = } exp_n \qquad (n \geq 1).$$

- The type-variable sequence is omitted from every custom-type binding and every type-alias binding; the class TyVarSeq is omitted.

- Each constructor binding $con\ ty_1 \cdots ty_n$ is replaced by con, whose arity $\mathrm{arity}(con)$ is n (Section 6.3). Thus a constructor binding has the form $con\ \langle \texttt{|}\ conbind \rangle$.

- A type-alias binding *tycon* `=` *ty* is replaced by *tycon* `={`lab_1`,` $\cdots$`,` lab_n`}` ($n \geq 0$) if it is a record alias (Section 3.7) whose record type has the labels $lab_1, \ldots, lab_n$, in the order written; every other type-alias binding is replaced by *tycon* alone.

- A port declaration `port` *var* `:` *ty* is replaced by `port` *var* `:` (out, τ) for an outgoing port and by `port` *var* `:` (in, τ) for an incoming one, where τ is the payload type of the port; the direction and the payload type are those that elaboration records in the port environment (Chapter 5, Section 5.4).

- After number specialisation (Section 6.2), an exposing item *var* naming a top-level variable whose declaration has been copied stands for the items naming *var* and each of the new variables that the copies bind, so that a module that uses *var* at an instance other than the default one finds the copy for that instance.

- An import `import` *modid* without `as` is read as `import` *modid* `as` *modid*: it provides its own *modid* as qualifier (Section 3.2).

As everywhere in the semantics, the derived forms of Appendix A have been applied, so the top-level declarations of a module stand in the order of Section A.2; and the imports of a module are the default imports of Figure 11 followed by the imports written in it.

7.2 Compound Objects

There are no simple objects beyond those of the Core. The compound objects for the dynamic semantics of Modules, extra to those of the Core (Figure 20), are shown in Figure 21.

$$\begin{aligned} CE \in \text{ConEnv} &= \text{TyCon} \xrightarrow{\text{fin}} \text{ValEnv} \\ (RA, CE, VE) \text{ or } I \in \text{Interface} &= \text{Fin}\,\text{TyCon} \times \text{ConEnv} \times \text{ValEnv} \\ ME \in \text{ModEnv} &= \text{ModId} \xrightarrow{\text{fin}} \text{Interface} \\ ME \text{ or } B \in \text{Basis} &= \text{ModEnv} \end{aligned}$$

Figure 21: Compound Semantic Objects for Modules

A *constructor environment* CE maps the type constructor *tycon* of each custom type to a value environment of its constructors. A dynamic *interface* (RA, CE, VE) records what a module makes available to other modules: VE binds every value it exposes, namely its exposed values and ports, the constructors of each custom type whose constructors it exposes, and the constructor of each record alias it exposes; CE maps every custom type that it exposes to the constructors of that type that it exposes, which is

the empty map for a type exposed closed; and RA is the set of the type constructors *tycon* of the record aliases whose constructors it exposes. For a type constructor *tycon* we write $con = tycon$ when *con* is spelt as *tycon*, as in Chapter 5: the constructor of a record alias *tycon* is bound in VE to the constructor *con* with $con = tycon$. A module environment maps each module identifier to the interface of that module, and a dynamic basis is a module environment. Interface, ModEnv and Basis are the dynamic objects of the names that Chapter 5 uses for static objects; Chapter 9 distinguishes them where both are in play.

Projection, injection and modification are as for the Core. In particular "CE of I" is the second component of I; $B(modid)$ means $ME(modid)$; "VE in Interface" means $(\emptyset, \{\}, VE)$; and interfaces are modified component by component, the set components by union. The interfaces inferred by the rules of this chapter have the following property, on which rule (121) relies.

> If (RA, CE, VE) is an interface inferred by the rules, then $con \in \mathrm{Dom}\ VE$ for every $tycon \in RA$ and $con = tycon$, and $VE(con) = v$ whenever $(CE(tycon))(con) = v$.

7.3 Merging Environments

Several imports of one module may provide the same qualifier, and the names they provide under it are then all available under that qualifier. Imports are therefore combined by the following modification, which merges, rather than replaces, the environment of a qualifier common to both of its arguments. For environments $E_1 = (QE_1, VE_1)$ and $E_2 = (QE_2, VE_2)$,

$$E_1 \boxplus E_2 \ = \ (QE,\ \ VE_1 + VE_2),$$

where $\mathrm{Dom}\ QE = \mathrm{Dom}\ QE_1 \cup \mathrm{Dom}\ QE_2$, and $QE(modid)$ is $QE_1(modid) + QE_2(modid)$ if $modid \in \mathrm{Dom}\ QE_1 \cap \mathrm{Dom}\ QE_2$, and otherwise whichever of $QE_1(modid)$ and $QE_2(modid)$ is defined. Where both arguments bind a name, the binding of E_2 is kept.

7.4 Record Constructors

The constructor of a record alias is a function of the fields of the record, taken in the order in which the alias writes them. For $n \geq 1$ and distinct labels $lab_1, \ldots, lab_n$ we extend BasVal with the basic value record($lab_1 \cdots lab_n$), of arity n, and extend APPLY to it by

$$\mathrm{APPLY}(\mathrm{record}(lab_1 \cdots lab_n),\ v_1 \cdots v_n) \ = \ \{\{lab_1 \mapsto v_1, \ldots, lab_n \mapsto v_n\}\}.$$

For $n = 0$ we let record() stand for the empty record $\{\}$, which is a value but not a basic value, since every basic value has arity at least 1. These values

are not bound in the initial basis (Appendix D); they are bound only by the record aliases of modules.

BasVal includes also the values of ports, the basic values outport(var, τ) and inport(var, τ) for a port name var and a payload type τ. They are defined in Chapter 8 (Section 8.7); this chapter uses only the fact that they are basic values.

7.5 Top-level Value Groups

A recursive group of top-level values need not consist of functions: the restrictions of Section 3.7 admit any group in which every cycle passes through an occurrence within a lambda expression. The operation Rec (Section 6.6) gives the bindings of a group only to the closures that the group binds, not to a closure held within a record, a tuple or a constructor value. We have therefore adopted the following translation, under which every member of a group is reached through a function of the unit value.

Let $valbind = var_1$ `=` exp_1 `and` $\cdots$ `and` var_n `=` exp_n be a top-level value declaration, and let $var'_1, \ldots, var'_n$ be distinct variables that occur nowhere in the program. For a phrase of the Core, $phrase^\bullet$ is the phrase obtained from it by replacing simultaneously, for $1 \le i \le n$,

- each free occurrence of the atomic expression var_i by the atomic expression `(`var'_i `())`, and
- each atomic expression `{` var_i `|` $exprow$ `}` in which var_i is free by `let` var_i `=` var'_i `()` `in {` var_i `|` $exprow^\bullet$ `}`,

where "free" is as in Section A.1. These are the only positions at which a variable occurs free in an expression. Then

$$valbind^\bullet \;=\; var'_1 \texttt{ = \textbackslash_ -> } exp_1^\bullet \texttt{ and } \cdots \texttt{ and } var'_n \texttt{ = \textbackslash_ -> } exp_n^\bullet .$$

Every binding of $valbind^\bullet$ is a lambda expression, so Rec ties the knot for it exactly as for a recursive declaration of the Core, and var'_i `()` then evaluates $exp_i^\bullet$. When no var_i is free in any exp_j, the declaration is not recursive and $exp_i^\bullet = exp_i$.

7.6 Inference Rules

The rules infer sentences of the forms

$$B \vdash phrase \Rightarrow A, \quad E \vdash phrase \Rightarrow A, \quad I \vdash phrase \Rightarrow A, \quad \vdash phrase \Rightarrow A,$$

where B is a basis, E an environment, I an interface, and A is the result: a basis with a value environment, a module identifier with an interface, an

environment, an interface, a constructor environment with a value environment, or a value environment. Each is pronounced "*phrase* evaluates to A", followed by "in B", "in E" or "in I". The hypotheses of a rule that are not sentences are its side conditions; some premises are sentences of the Core (Chapter 6). The convention for options is the same as for the Core static semantics.

As in the Core, sentences have no states and there are no exception packets, so every rule is given in full, and the order of the premises of a rule carries no meaning (Section 6.7).

Modules

$\boxed{B \vdash \mathit{module} \Rightarrow B', \mathit{VE}}$

$$\frac{B \vdash \mathit{imports} \Rightarrow E \qquad E_0 + E \vdash \mathit{topdec} \Rightarrow I \qquad I \vdash \mathit{header} \Rightarrow (\mathit{modid}, I')}{B \vdash \mathit{header}\ \mathit{imports}\ \mathit{topdec} \Rightarrow \{\mathit{modid} \mapsto I'\},\ \mathit{VE}\ \text{of}\ I} \tag{113}$$

Comments:

(113) Here E_0 is the environment of Appendix D that binds the list constructor names `[]` and `::`; every module starts from it, as its elaboration starts from π_0 (rule (46)). The top-level declarations are evaluated in E_0 modified by the environment that the imports give, so a name that a module declares replaces, within it, an imported name spelt alike (rule (129)). The basis $\{\mathit{modid} \mapsto I'\}$ holds only what the header exposes; the value environment VE of I of all the module's own values is the second result, as in the static semantics, because Chapter 9 takes the value of `main` from it, and `main` need not be exposed. A module is evaluated once, after the modules it imports, and its top-level values are evaluated then (Chapter 9).

Headers

$\boxed{I \vdash \mathit{header} \Rightarrow (\mathit{modid}, I')}$

$$\frac{I \vdash \mathit{explist} \Rightarrow I'}{I \vdash \langle \texttt{port} \rangle\ \texttt{module}\ \mathit{modid}\ \texttt{exposing}\ \mathit{explist} \Rightarrow (\mathit{modid}, I')} \tag{114}$$

Comments:

(114) The word `port` has no dynamic content: a port module and a plain module evaluate alike.

Imports

$\boxed{B \vdash \mathit{imports} \Rightarrow E}$

$$\frac{B(\mathit{modid}) = I \qquad \langle I \vdash \mathit{explist} \Rightarrow I' \rangle \qquad B \vdash \mathit{imports} \Rightarrow E}{\begin{array}{l} B \vdash \texttt{import}\ \mathit{modid}\ \texttt{as}\ \mathit{modid}'\ \langle \texttt{exposing}\ \mathit{explist} \rangle\ \mathit{imports} \\ \quad \Rightarrow (\{\mathit{modid}' \mapsto (\{\}, \mathit{VE}\ \text{of}\ I)\},\ \{\}\langle + \mathit{VE}\ \text{of}\ I' \rangle) \boxplus E \end{array}} \tag{115}$$

$$\frac{}{B \vdash \Rightarrow (\{\}, \{\})} \tag{116}$$

Comments:

(115) $B(modid)$ is defined, since a module is evaluated only after every module it imports (Chapter 9). The qualifier $modid'$ gives access to every value that the imported module exposes, its constructors included; the exposing list makes a part of them available unqualified. The environment of later imports stands on the right of $\boxplus$, so a name exposed by a later import replaces the same name exposed by an earlier one, and in particular a written import replaces a default import. Where elaboration has found an unqualified name ambiguous, the name is not used, so which binding is kept is immaterial.

Exposing Lists

$\boxed{I \vdash explist \Rightarrow I'}$

$$\frac{}{I \vdash \texttt{(..)} \Rightarrow I} \tag{117}$$

$$\frac{I \vdash item_1 \Rightarrow I_1 \quad \cdots \quad I \vdash item_n \Rightarrow I_n}{I \vdash \texttt{(}item_1\ \texttt{,}\ \cdots\ \texttt{,}\ item_n\texttt{)} \Rightarrow I_1 + \cdots + I_n} \tag{118}$$

Comments:

(117) The sentence $I \vdash explist \Rightarrow I'$ is the dynamic counterpart of the interface Expose of Chapter 5: the values of I' are exactly those whose names elaboration admits through *explist*.

Exposing Items

$\boxed{I \vdash item \Rightarrow I'}$

$$\frac{(VE \text{ of } I)(var) = v}{I \vdash var \Rightarrow \{var \mapsto v\} \text{ in Interface}} \tag{119}$$

$$\frac{}{I \vdash \texttt{(}op\texttt{)} \Rightarrow (\emptyset, \{\}, \{\})} \tag{120}$$

$$\frac{tycon \in RA \text{ of } I \qquad con = tycon \qquad (VE \text{ of } I)(con) = v}{I \vdash tycon \Rightarrow (\{tycon\}, \{\}, \{con \mapsto v\})} \tag{121}$$

$$\frac{tycon \notin RA \text{ of } I \qquad CE = \{tycon \mapsto \{\}\,;\, tycon \in \mathrm{Dom}\,(CE \text{ of } I)\}}{I \vdash tycon \Rightarrow (\emptyset, CE, \{\})} \tag{122}$$

$$\frac{(CE \text{ of } I)(tycon) = VE}{I \vdash tycon\ \texttt{(..)} \Rightarrow (\emptyset, \{tycon \mapsto VE\}, VE)} \tag{123}$$

Comments:

(119) The variable is a value or a port.

(120) An operator is not in the domain of a value environment: its value is given by the operator table $\mathrm{OT}_{\mathrm{DYN}}$, and is the same in every module (rule (68)). Exposing or importing an operator therefore binds nothing.

(121) Rules (121) and (122) cover the one item *tycon*, distinguished by whether *tycon* is a record alias whose constructor I exposes. The property of Section 7.2 ensures that (VE of I)(*con*) is then defined. A record alias exposed by name exposes its constructor, so `import Geometry exposing (Point)` makes `Point 1 2` available.

(122) A custom type exposed without `(..)` exposes no value, even when one of its constructors is spelt as the type: in a module with `type Id = Id Int` and header `exposing (Id)`, the constructor `Id` is not exposed. The type is still exposed, closed: the constructor environment maps it to the empty map, so that an import of `Id(..)` has a meaning (rule (123)). An alias of a type other than a closed record is in neither RA of I nor Dom (CE of I), and exposes nothing.

(123) By elaboration, *tycon* names a custom type that I exposes, so (CE of I)(*tycon*) is defined; an item *tycon* `(..)` for an alias is rejected (Chapters 3 and 5). In a header, I is the module's own interface, which holds every constructor of *tycon*. In an import, VE holds the constructors that the imported module exposes, and is the empty map when it exposes *tycon* closed: with `module A exposing (T)`, `type T = T Int` and `import A exposing (T(..))`, the import makes no constructor available.

Top-level Declarations

$\boxed{E \vdash \mathit{topdec} \Rightarrow I}$

$$\frac{E \vdash \mathit{valbind}^{\bullet} \Rightarrow VE \qquad E + \mathrm{Rec}\ VE \vdash \mathit{var}'_1\ \texttt{()} \Rightarrow v_1 \quad \cdots \quad E + \mathrm{Rec}\ VE \vdash \mathit{var}'_n\ \texttt{()} \Rightarrow v_n}{E \vdash \mathit{valbind} \Rightarrow \{\mathit{var}_1 \mapsto v_1, \ldots, \mathit{var}_n \mapsto v_n\}\ \text{in Interface}} \tag{124}$$

$$\frac{\vdash \mathit{datbind} \Rightarrow (CE, VE)}{E \vdash \texttt{type}\ \mathit{datbind} \Rightarrow (\emptyset, CE, VE)} \tag{125}$$

$$\frac{\vdash \mathit{typbind} \Rightarrow (RA, VE)}{E \vdash \texttt{type alias}\ \mathit{typbind} \Rightarrow (RA, \{\}, VE)} \tag{126}$$

$$\frac{}{E \vdash \texttt{port}\ \mathit{var}\ \texttt{:}\ (\mathrm{out}, \tau) \Rightarrow \{\mathit{var} \mapsto \mathrm{outport}(\mathit{var}, \tau)\}\ \text{in Interface}} \tag{127}$$

$$\frac{}{E \vdash \texttt{port}\ var\ \texttt{:}\ (\mathrm{in}, \tau) \Rightarrow \{var \mapsto \mathrm{inport}(var, \tau)\}\ \text{in Interface}} \tag{128}$$

$$\frac{E \vdash topdec_1 \Rightarrow (RA_1, CE_1, VE_1) \qquad E + VE_1 \vdash topdec_2 \Rightarrow I_2}{E \vdash topdec_1\ \texttt{;}\ topdec_2 \Rightarrow (RA_1, CE_1, VE_1) + I_2} \tag{129}$$

Comments:

(124) Here *valbind* is var_1 `=` exp_1 `and` $\cdots$ `and` var_n `=` exp_n, and $valbind^\bullet$ and $var'_1, \ldots, var'_n$ are as in Section 7.5; the first premise is a sentence of the Core (Value Bindings), and the choice of the new variables does not affect the result. Each var_i is bound to a value obtained once, when its module is evaluated. A reference to var_i within the group evaluates $exp_i^\bullet$ again in the same environment, which gives v_i again, there being no state, unless exp_i relies on the unspecified result of an equality of functions (Section 6.5). A top-level value is thus evaluated once only up to that non-determinism: where exp_i compares functions, a reference to var_i within its group may give another of the values that the equality of functions allows, and so differ from v_i. Thus with `ones = { head = 1, tail = \_ -> ones }`, the expression `(ones.tail ()).head` evaluates to `1`. In `x = let f _ = x in f ()` the occurrence of `x` is delayed, so the restrictions of Section 3.7 admit it, but evaluating var'_1 `()` needs var'_1 `()` again, without end: no sentence about the declaration can be inferred, and the module has no meaning.

(127) Rules (127) and (128): the payload type τ comes from elaboration through the reduced syntax; it has no type variables, so the value of a port is the same at every use. Applied to a value, the value of an outgoing port gives a command that sends its encoding at τ; applied to a function, the value of an incoming port gives a subscription (Section 8.7). Port names are unique in a program, so the name identifies the port.

(129) By the derived forms of Section A.2, *topdec* consists of the custom-type group, then the aliases, then the ports and value groups in dependency order, so every declaration is evaluated after each declaration it refers to. Where that order leaves two declarations unordered, either order gives the same result.

Custom-type Bindings

$\boxed{\vdash datbind \Rightarrow (CE, VE)}$

$$\frac{\vdash conbind \Rightarrow VE \qquad \langle \vdash datbind \Rightarrow (CE, VE') \rangle}{\vdash tycon\ \texttt{=}\ conbind\ \langle \texttt{and}\ datbind \rangle \Rightarrow (\{tycon \mapsto VE\} \langle + CE \rangle,\ VE \langle + VE' \rangle)} \tag{130}$$

Comments:

(130) No type names are generated: a custom type has no dynamic content beyond its constructors. The constructors of a module are distinct (Section 3.7), so the modification of *VE* replaces no binding.

Constructor Bindings

$\boxed{\vdash \mathit{conbind} \Rightarrow \mathit{VE}}$

$$\frac{\langle \vdash \mathit{conbind} \Rightarrow \mathit{VE} \rangle}{\vdash \mathit{con}\ \langle \mid\ \mathit{conbind} \rangle \Rightarrow \{\mathit{con} \mapsto \mathit{con}\} \langle + \mathit{VE} \rangle} \tag{131}$$

Comments:

(131) The constructor value *con* awaits all arity(*con*) of its arguments; a constructor of arity 0 is thus a complete value. Constructors spelt alike in different modules give equal constructor values, which never meet in one rule, since they have different types (Section 6.3).

Alias Bindings

$\boxed{\vdash \mathit{typbind} \Rightarrow (\mathit{RA}, \mathit{VE})}$

$$\frac{\langle \mathit{con} = \mathit{tycon} \rangle}{\begin{array}{l}\vdash \mathit{tycon}\ \langle \texttt{=}\ \texttt{\{}\mathit{lab}_1\texttt{,}\ \cdots\texttt{,}\ \mathit{lab}_n\texttt{\}} \rangle \\ \Rightarrow (\emptyset \langle \cup \{\mathit{tycon}\} \rangle,\ \{\} \langle + \{\mathit{con} \mapsto \mathrm{record}(\mathit{lab}_1 \cdots \mathit{lab}_n)\} \rangle)\end{array}} \tag{132}$$

Comments:

(132) Here $n \geq 0$, and by the syntactic restrictions the labels are distinct. The constructor takes the fields in the order in which the alias writes them: with `type alias P = { y : Int, x : Int }`, `P 1 2` evaluates to `{ y = 1, x = 2 }`. An alias of the empty record binds its name to the empty record itself (Section 7.4). An alias of any other type binds nothing, and its type constructor is not in *RA*; so rule (126) records in *RA* exactly the record aliases.

Chapter 8

Tasks, Commands, Subscriptions and Ports

This chapter defines the effect values of Standard Elm: the tasks, which describe work that may wait, fail and start other processes; the commands and subscriptions, which a program hands to the platform to have tasks performed and values exchanged with its host (the platform and the host are as in Chapter 1); and the program values. It defines the boundary values that cross a port, which are JSON values in which byte sequences may stand, and the encoding and decoding of the values of every type that a port admits, which settles exactly what can and cannot be sent over a port. Its sentences infer that a task reduces to another task, silently or by performing a primitive action; that a value encodes as a boundary value, or a boundary value decodes as a value, at a type; and that a leaf of a command or subscription, receiving an input, gives a message. These objects are defined here; the running of a program, which creates and schedules processes and handles their messages one at a time, is defined by Chapter 9 in terms of them. Every phrase whose value is used here is assumed to have elaborated (Chapter 5).

8.1 Simple Objects

The simple objects of this chapter are shown in Figure 22; the classes of Chapter 6 are used as there.

A *process identifier* names a process (Section 8.4); ProcId is an infinite set. The class Prim of *primitives* is

$$\mathrm{Prim} = \{\mathrm{sleep}, \mathrm{spawn}, \mathrm{kill}\} \cup \mathrm{Prim}_{\mathrm{plat}},$$

where sleep, spawn and kill are the primitives of the module `Process`, and $\mathrm{Prim}_{\mathrm{plat}}$ is a set of further primitives (reading a clock, an HTTP request, and so on) which is a parameter of the platform on which a program runs. Each primitive π has an *arity* $\mathrm{arity}(\pi) \geq 1$; the arity of each of sleep, spawn and

pid	$\in$ ProcId	process identifiers
π	$\in$ Prim	primitives
p or var	$\in$ Var	port names
	$\in$ $\text{Manager}_{\text{plat}}$	managers of the platform
	$\in$ JNum	JSON numbers
	$\in$ Byt	byte sequences

Figure 22: Simple Semantic Objects of the Effects

kill is 1. A port name p is the variable of a `port` declaration (Chapter 3). Port names are global to a program and distinct from one another (Section 3.7). $\text{Manager}_{\text{plat}}$, a parameter of the platform, is a set of names of *managers*, other than the managers of tasks and ports defined in Section 8.5, and disjoint from the port names.

JNum is the set of *JSON numbers*: the rational numbers $m \times 10^e$ with m and e integers, that is, the numbers a JSON text can spell. Every integer of IntRange and every finite floating-point number is a member of JNum; we omit the injections from these numbers into JNum. The injection takes both floating-point zeros, 0 and -0, to the JSON number 0: a JSON number has no sign apart from its value. We write Str for the set of strings of SVal, Flt for the set of floating-point numbers of SVal, and $\text{Flt}_{\text{fin}} \subseteq \text{Flt}$ for its finite members, so that $\text{Flt} \setminus \text{Flt}_{\text{fin}}$ holds the two infinities and NaN. For $j \in \text{JNum}$, float(j) is the floating-point number nearest j, rounded as the value of a floating-point constant is (Section 6.3); float(0) is the float 0, not -0.

Byt is the set of *byte sequences*: the finite sequences of integers from 0 to 255. We write bytes($o_1 \cdots o_k$), with $k \geq 0$ and $0 \leq o_i \leq 255$ for each i, for the byte sequence of $o_1, \ldots, o_k$ in order. A byte sequence is not a JSON number, a string or an array: Byt is disjoint from JNum, from Str and from every other class of this chapter.

8.2 Compound Objects

The compound objects of this chapter are shown in Figure 23. The classes EffVal and JVal, which Chapter 6 includes in Val (Figure 20), are defined here. TaskVal, CmdVal, SubVal and JVal are defined together with the classes of Figure 20; all are the smallest classes closed under the constructions of both figures. The class BVal of boundary values, which includes JVal, is defined here too; its other members are not in Val.

$$
\begin{array}{rl}
ev \in \text{EffVal} & = \text{TaskVal} \cup \text{CmdVal} \cup \text{SubVal} \cup \text{ProcId} \cup \text{ProgVal} \\
tk \in \text{TaskVal} & = (\{\text{succeed}, \text{fail}\} \times \text{Val}) \\
& \quad \cup (\{\text{andThen}, \text{onError}\} \times \text{Val} \times \text{TaskVal}) \\
& \quad \cup \bigcup_{k \geq 1} (\{\text{prim}\} \times \text{Prim} \times \text{Val}^k) \\
cm \in \text{CmdVal} & = \bigcup_{n \geq 0} (\{\text{batch}\} \times \text{CmdVal}^n) \\
& \quad \cup (\{\text{map}\} \times \text{Val} \times \text{CmdVal}) \\
& \quad \cup (\{\text{leaf}\} \times \text{Manager} \times (\text{Val} \cup \text{BVal})) \\
sb \in \text{SubVal} & = \bigcup_{n \geq 0} (\{\text{batch}\} \times \text{SubVal}^n) \\
& \quad \cup (\{\text{map}\} \times \text{Val} \times \text{SubVal}) \\
& \quad \cup (\{\text{leaf}\} \times \text{Manager} \times \text{Val}) \\
m \in \text{Manager} & = \{\text{task}\} \cup (\{\text{out}\} \times \text{Var}) \\
& \quad \cup (\{\text{in}\} \times \text{Var} \times \text{Type}) \cup \text{Manager}_{\text{plat}} \\
a \in \text{Action} & = \bigcup_{k \geq 1} \text{Prim} \times \text{Val}^k \times \text{TaskVal} \\
pv \in \text{ProgVal} & = \{\text{prog}\} \times \text{Record} \\
j \in \text{BVal} & = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \text{JNum} \cup \text{Str} \cup \text{Byt} \\
& \quad \cup \bigcup_{n \geq 0} \text{BVal}^n \cup (\text{Str} \xrightarrow{\text{fin}} \text{BVal}) \\
\in \text{JVal} & = \{\texttt{null}, \texttt{true}, \texttt{false}\} \cup \text{JNum} \cup \text{Str} \\
& \quad \cup \bigcup_{n \geq 0} \text{JVal}^n \cup (\text{Str} \xrightarrow{\text{fin}} \text{JVal})
\end{array}
$$

Figure 23: Compound Semantic Objects of the Effects

We write the members of these classes as follows:

tasks:	$\text{succeed}\, v, \quad \text{fail}\, v, \quad \text{andThen}(v, tk),$ $\text{onError}(v, tk), \quad \text{prim}(\pi, v_1 \cdots v_k);$
commands:	$\text{batch}(cm_1 \cdots cm_n), \quad \text{map}(v, cm), \quad \text{leaf}(m, v);$
managers:	$\text{task}, \quad \text{out}\, p, \quad \text{in}(p, \tau);$
actions:	$\pi(v_1 \cdots v_k) \Downarrow tk;$
program values:	$\text{prog}\, r.$

Subscriptions are written as commands are, and an action $\pi(v_1 \cdots v_k) \Downarrow tk$ is the triple $(\pi, v_1 \cdots v_k, tk)$. We write none for batch( ), the batch with no components, in CmdVal and in SubVal. In a primitive task $\text{prim}(\pi, v_1 \cdots v_k)$, always $k = \text{arity}(\pi)$.

A *boundary value* is `null`, `true`, `false`, a number, a string, a byte sequence, an *array*, which is a member $[j_1, \ldots, j_n]$ of BVal^n, or an *object*, which is a finite map from strings to boundary values. An object has no order on its keys. A boundary value is thus a JSON value in which a byte sequence may stand wherever a value may. A *JSON value* is a boundary value in which no byte sequence occurs; JVal is the class of JSON values, and $\text{JVal} \subseteq \text{BVal}$. A byte sequence is not a JSON value. The metavariable j ranges over BVal; where only a JSON value is meant we write $j \in \text{JVal}$. The values of the type `Json.Encode.Value` (also called `Json.Decode.Value`) are exactly the JSON values, the members of JVal, so no byte sequence occurs in one. As with the

other classes, JVal is disjoint from the rest of Val, and we omit the injection: a JSON string and the String value with the same characters are different values, and `null` is not the unit value (). A boundary value that is not a JSON value is not a value of the language; such a boundary value occurs only at a port, and in the leaves of outgoing ports, which CmdVal admits.

For values $v_1, \ldots, v_n$ we write $[v_1, \ldots, v_n]$ for the list value $v_1 :: \cdots :: v_n :: []$, and $[]$ when $n = 0$. We write the value to which the initial dynamic basis (Appendix D) binds a long identifier of the basis by that identifier in typewriter, as in `Task.succeed`; the basic values `Array.fromList` and `Array.toList` of Appendix D (Section D.4.9), which no identifier of the basis spells, are written likewise. Finite multisets are written $\{\!|\, c_1, \ldots, c_n \,|\!\}$, and $\uplus$ is the union of multisets, which adds multiplicities.

8.3 Inference Rules

The semantic rules of this chapter allow sentences of the forms

$$tk \xrightarrow{\langle a \rangle} tk', \qquad \mathrm{Enc}_\tau(v) = j, \qquad \mathrm{Dec}_\tau(j) = v, \qquad \vdash c \triangleleft x \Rightarrow v$$

to be inferred. The first is pronounced "tk reduces to tk'", and, when the label a is present, "performing a"; a reduction without a label is *silent*. The second is pronounced "v encodes at τ as j", and the third "j decodes at τ as v"; there τ is a type with $\mathrm{Port}(\tau)$ (Section 5.4). In the fourth, c is a command or subscription value with no batch in it (a *leaf command*, Section 8.5), the input x is a value or a pair (p, j) of a port name and a boundary value, and the sentence is pronounced "c, receiving x, gives the message v". Premises also use the sentence $\vdash v_1\, v_2 \Rightarrow v$ of the application of values (Section 6.7). Some hypotheses in rules are not of these forms; they are side conditions.

The convention for options is the same as for the Core static semantics. In particular, in a task reduction whose label is written $\langle a \rangle$, the label is present on every arrow of the rule or absent from every one. As in the Core (Section 6.7), sentences have no states and there are no exception packets, so every rule is given in full. We allow the compound variable cm/sb to range over $\mathrm{CmdVal} \cup \mathrm{SubVal}$.

Enc_τ and Dec_τ are defined by the rules alone: $\mathrm{Enc}_\tau(v)$ is the boundary value j for which $\mathrm{Enc}_\tau(v) = j$ can be inferred, and is undefined when there is none, and likewise for $\mathrm{Dec}_\tau(j)$ (Section 8.7).

8.4 Tasks

A task value describes work; it does nothing until a process performs it, and a task value performed twice does its work twice. The task $\mathrm{succeed}\, v$ has *succeeded* with v, and $\mathrm{fail}\, v$ has *failed* with v; these two are the *outcomes*. The

task andThen(v, tk) performs tk and, if it succeeds with v', goes on with the task to which v applied to v' evaluates; onError(v, tk) is the same for failure. A primitive task prim$(\pi, v_1 \cdots v_k)$ performs the primitive π on $v_1 \cdots v_k$.

A *process* is a process identifier together with the task that it is reducing. A process reduces its task by the rules below, one step at a time, until the task is an outcome; then the process has *ended*. A silent step concerns that process alone. A labelled step performs a primitive action $\pi(v_1 \cdots v_k) \Downarrow tk$, whose outcome tk is a member of the set

$$\mathrm{Out}(\pi, v_1 \cdots v_k)$$

of possible outcomes, and which may act on the world outside the process. For the primitives of `Process`,

$$\begin{array}{rcl} \mathrm{Out}(\mathrm{sleep}, v) & = & \{\mathrm{succeed}\,()\}, \\ \mathrm{Out}(\mathrm{spawn}, tk) & = & \{\mathrm{succeed}\,pid \mid pid \in \mathrm{ProcId}\}, \\ \mathrm{Out}(\mathrm{kill}, pid) & = & \{\mathrm{succeed}\,()\}, \end{array}$$

and their actions mean the following.

- The action $\mathrm{sleep}(v) \Downarrow \mathrm{succeed}\,()$, where $v \in \mathrm{Flt}$, takes place no sooner than $\max(0, v)$ milliseconds after the process reached $\mathrm{prim}(\mathrm{sleep}, v)$, when $v \in \mathrm{Flt}_{\mathrm{fin}}$; when v is an infinity or NaN, when it takes place is not defined.

- $\mathrm{spawn}(tk) \Downarrow \mathrm{succeed}\,pid$ starts a new process, named by pid, reducing tk. The identifier pid names no process that has been started before. The new process is independent of the one that started it: its outcome is delivered to no other process, and when it fails it ends, and nothing else happens.

- $\mathrm{kill}(pid) \Downarrow \mathrm{succeed}\,()$ ends the process named by pid at once, so that it takes no further step, when that process has not ended and is not the process performing the action. Otherwise the action has no effect.

For a primitive $\pi \in \mathrm{Prim}_{\mathrm{plat}}$, the set $\mathrm{Out}(\pi, v_1 \cdots v_k)$, which may hold failures, and the meaning of its actions are given by the platform. Chapter 9 defines the processes of a running program, and it is there that the conditions above on when an action may take place, and on what it does to other processes, are enforced.

Task Reduction

$\boxed{tk \xrightarrow{\langle a \rangle} tk'}$

$$\frac{\vdash v\, v' \Rightarrow tk}{\mathrm{andThen}(v, \mathrm{succeed}\, v') \to tk} \tag{133}$$

$$\frac{}{\mathrm{andThen}(v, \mathrm{fail}\, v') \to \mathrm{fail}\, v'} \tag{134}$$

$$\frac{\vdash v\, v' \Rightarrow tk}{\mathrm{onError}(v, \mathrm{fail}\, v') \to tk} \tag{135}$$

$$\frac{}{\mathrm{onError}(v, \mathrm{succeed}\, v') \to \mathrm{succeed}\, v'} \tag{136}$$

$$\frac{tk \xrightarrow{\langle a \rangle} tk'}{\mathrm{andThen}(v, tk) \xrightarrow{\langle a \rangle} \mathrm{andThen}(v, tk')} \tag{137}$$

$$\frac{tk \xrightarrow{\langle a \rangle} tk'}{\mathrm{onError}(v, tk) \xrightarrow{\langle a \rangle} \mathrm{onError}(v, tk')} \tag{138}$$

$$\frac{tk \in \mathrm{Out}(\pi, v_1 \cdots v_k)}{\mathrm{prim}(\pi, v_1 \cdots v_k) \xrightarrow{\pi(v_1 \cdots v_k) \Downarrow tk} tk} \tag{139}$$

Comments:

(133) The task tk to which v applied to v' evaluates is not reduced by this step; it is the task the process performs next. If the application has no result (for instance, it reaches `Debug.todo`), the task has no reduction, and the process never ends.

(134) The continuation v is skipped: a failure passes over every andThen until an onError handles it, as rule (135) does. Rule (136) is the dual for success.

(137) Rules (137) and (138) carry both silent and labelled steps out of the task that is performed first. Every step of a process is a separate reduction, and another process may take steps between any two of them: a process may be suspended at any andThen, and the definition guarantees no more. In particular, a program cannot rely on a process running without interruption until it reaches a primitive.

(139) This is the only rule with a label, and it does not determine its result: the outcome is any member of $\mathrm{Out}(\pi, v_1 \cdots v_k)$. For spawn, the process identifier is unconstrained here; Chapter 9 chooses one that names no earlier process. For the primitives of the platform, the outcome is decided by the platform, which may also take time to decide it, as for sleep.

The following properties follow from the rules.

An outcome has no reduction. If tk is not an outcome, then either every reduction of tk is silent, and all have the same right-hand side up to the choice of a member of a set given by APPLY; or every reduction of tk is labelled by an action $\pi(v_1 \cdots v_k) \Downarrow tk'$, for one and the same π and $v_1 \cdots v_k$; or tk has no reduction, which happens only when an application in a premise of rule (133) or (135) has no result.

8.5 Commands and Subscriptions

A command value, or a subscription value, is a finite tree whose leaves are requests to managers. A *manager* m is one of the following.

- The *task manager* task. Its leaves $\mathrm{leaf}(\mathrm{task}, tk)$ are commands; it performs tk in a new process, and when tk succeeds with v, the leaf gives a message from v.
- The manager $\mathrm{out}\, p$ of an outgoing port p. Its leaves $\mathrm{leaf}(\mathrm{out}\, p, j)$, with $j \in \mathrm{BVal}$, are commands; each sends j to the host, and gives no message.
- The manager $\mathrm{in}(p, \tau)$ of an incoming port p of payload type τ. Its leaves $\mathrm{leaf}(\mathrm{in}(p, \tau), v)$ are subscriptions, where v is a function value; while the subscription is current, each boundary value the host sends on p that decodes at τ gives a message through v (Section 8.8).
- A manager of the platform, in $\mathrm{Manager}_{\mathrm{plat}}$. The values its leaves hold, whether they are commands or subscriptions, and the messages they give are defined by the platform.

Since port names are distinct (rule (172)), a port name p determines the payload type τ of every manager $\mathrm{in}(p, \tau)$ of a program. A batch groups commands, and $\mathrm{map}(v, cm)$ applies the function value v to every message that cm gives.

A command or subscription value with no batch in it has the form $\mathrm{map}(v_n, \cdots \mathrm{map}(v_1, \mathrm{leaf}(m, v)) \cdots)$ with $n \geq 0$, and is called a *leaf command* (or *leaf subscription*); its manager is m. The *flattening* $\mathrm{Flat}(cm)$ of a command value cm is the finite multiset of leaf commands defined by

$$\begin{aligned} \mathrm{Flat}(\mathrm{batch}(cm_1 \cdots cm_n)) &= \mathrm{Flat}(cm_1) \uplus \cdots \uplus \mathrm{Flat}(cm_n), \\ \mathrm{Flat}(\mathrm{map}(v, cm)) &= \{\!|\, \mathrm{map}(v, c) \mid c \in \mathrm{Flat}(cm) \,|\!\}, \\ \mathrm{Flat}(\mathrm{leaf}(m, v)) &= \{\!|\, \mathrm{leaf}(m, v) \,|\!\}, \end{aligned}$$

where a leaf command that occurs several times in $\mathrm{Flat}(cm)$ gives as many members of the multiset on the right of the second equation. The flattening

Flat(sb) of a subscription value is defined by the same equations. In particular Flat(none) = Flat(batch(none none)) = {| |}.

We have adopted the following convention, the *effect order convention*. A command determines, through its flattening, which leaf commands it holds and how many times each, but no order among them. Wherever the leaf commands of commands are performed (Chapter 9), the leaf commands of one command for one manager are performed in no specified order, and the leaf commands for different managers are performed in no specified order relative to one another. The same holds for subscriptions. The orders that implementations choose are recorded in Appendix E.

A program value prog r is the meaning of a `Program`: r is the record of the functions that make it up, with at least the fields `init`, `update` and `subscriptions`. The function `Platform.worker` makes a program value from its record (Figure 24). A platform may provide further functions whose results are program values, whose records have further fields, such as `view`; what those fields mean is defined by the platform. Chapter 9 defines how a program value runs.

8.6 The Effect Values of the Basis

Figure 24 gives the arity of each basic value of the initial basis that makes an effect value, and the result of APPLY on it (Section 6.5). Each set has exactly one member. One of these basic values, `Task.command`, is bound to no identifier of the basis; it is applied only by the texts of `Task.perform` and `Task.attempt`, as described below (Appendix D).

basic value b	arity(b)	APPLY($b, v_1 \cdots v_n$)
`Task.succeed`	1	APPLY$(b, v) = \{\text{succeed}\, v\}$
`Task.fail`	1	APPLY$(b, v) = \{\text{fail}\, v\}$
`Task.andThen`	2	APPLY$(b, v, tk) = \{\text{andThen}(v, tk)\}$
`Task.onError`	2	APPLY$(b, v, tk) = \{\text{onError}(v, tk)\}$
`Task.command`	1	APPLY$(b, \texttt{Perform}\ tk) = \{\text{leaf}(\text{task}, tk)\}$
`Process.sleep`	1	APPLY$(b, v) = \{\text{prim}(\text{sleep}, v)\}$
`Process.spawn`	1	APPLY$(b, tk) = \{\text{prim}(\text{spawn}, tk)\}$
`Process.kill`	1	APPLY$(b, pid) = \{\text{prim}(\text{kill}, pid)\}$
`Cmd.batch`	1	APPLY$(b, [cm_1, \ldots, cm_n]) = \{\text{batch}(cm_1 \cdots cm_n)\}$
`Cmd.map`	2	APPLY$(b, v, cm) = \{\text{map}(v, cm)\}$
`Sub.batch`	1	APPLY$(b, [sb_1, \ldots, sb_n]) = \{\text{batch}(sb_1 \cdots sb_n)\}$
`Sub.map`	2	APPLY$(b, v, sb) = \{\text{map}(v, sb)\}$
`Platform.worker`	1	APPLY$(b, r) = \{\text{prog}\, r\}$

Figure 24: APPLY on Effect Values

The other functions of these modules that the initial basis provides are

written in Elm in elm/core, and each is defined by its text (Section D.3). They are `Task.map`, `Task.map2` to `Task.map5`, `Task.sequence`, `Task.mapError`, `Task.perform`, `Task.attempt`, `Cmd.none` and `Sub.none`. Thus `Cmd.none` and `Sub.none` are none, the value of `batch []`. The texts of `Task.perform` and `Task.attempt` apply `command` to a constructor value `Perform` tk, where `Perform` is the constructor of the type `MyCmd` of the module `Task`, which that module does not expose. Effect modules are outside Standard Elm, so these two names are given directly: the texts of `Task` are evaluated in an environment that binds `Perform` to the constructor `Perform`, of arity 1, and `command` to the basic value `Task.command` of the figure, and neither is bound to an identifier of the basis, as Appendix D states (Section D.3). Hence `Task.perform` v tk evaluates to leaf(task, tk'), where tk' is the value of `Task.map` v tk, a task that cannot fail and whose success value is the message. Commands of outgoing ports and subscriptions of incoming ports are made by the values of the ports (Section 8.7).

8.7 Ports

A value crosses a port as a boundary value: the port boundary is defined over the boundary values of BVal, which are the JSON values of JVal together with those in which byte sequences occur, and host values that are not boundary values are outside the definition. How a host represents a byte sequence is outside the definition as well (Appendix E). A value v sent on an outgoing port of payload type τ becomes the boundary value $\mathrm{Enc}_\tau(v)$, and a boundary value j arriving at an incoming port of payload type τ becomes the value $\mathrm{Dec}_\tau(j)$, if that is defined. The payload types are the types τ with $\mathrm{Port}(\tau)$ (Section 5.4); aliases have been expanded, and a payload type has no type variables. Labels are identified with the strings of their characters, so that a record type's labels are the keys of the objects that encode its records. The values of the type `Bytes` are the constructor values binary $[o_1, \ldots, o_k]$ of Appendix D (Section D.4.10), each o_i an Int value from 0 to 255.

Encoding

$\boxed{\mathrm{Enc}_\tau(v) = j}$

$$\frac{}{\mathrm{Enc}_{\texttt{Int}}(sv) = sv} \tag{140}$$

$$\frac{sv \in \mathrm{Flt}_{\mathrm{fin}}}{\mathrm{Enc}_{\texttt{Float}}(sv) = sv} \tag{141}$$

$$\frac{sv \in \mathrm{Flt} \setminus \mathrm{Flt}_{\mathrm{fin}}}{\mathrm{Enc}_{\texttt{Float}}(sv) = \texttt{null}} \tag{142}$$

$$\frac{}{\mathrm{Enc}_{\texttt{Bool}}(\texttt{True}) = \texttt{true}} \tag{143}$$

$$\frac{}{\mathrm{Enc}_{\texttt{Bool}}(\texttt{False}) = \texttt{false}} \tag{144}$$

$$\frac{}{\mathrm{Enc}_{\texttt{String}}(sv) = sv} \tag{145}$$

$$\frac{}{\mathrm{Enc}_{\texttt{Bytes}}(\mathrm{binary}\,[o_1, \ldots, o_k]) = \mathrm{bytes}(o_1 \cdots o_k)} \tag{146}$$

$$\frac{}{\mathrm{Enc}_{()}(()) = \texttt{null}} \tag{147}$$

$$\frac{}{\mathrm{Enc}_{\tau\,\texttt{Maybe}}(\texttt{Nothing}) = \texttt{null}} \tag{148}$$

$$\frac{\mathrm{Enc}_{\tau}(v) = j}{\mathrm{Enc}_{\tau\,\texttt{Maybe}}(\texttt{Just}\ v) = j} \tag{149}$$

$$\frac{\mathrm{Enc}_{\tau}(v_1) = j_1 \quad \cdots \quad \mathrm{Enc}_{\tau}(v_n) = j_n}{\mathrm{Enc}_{\tau\,\texttt{List}}([v_1, \ldots, v_n]) = [j_1, \ldots, j_n]} \tag{150}$$

$$\frac{\vdash \texttt{Array.toList}\ v \Rightarrow v' \qquad \mathrm{Enc}_{\tau\,\texttt{List}}(v') = j}{\mathrm{Enc}_{\tau\,\texttt{Array}}(v) = j} \tag{151}$$

$$\frac{\mathrm{Enc}_{\tau_1}(v_1) = j_1 \qquad \mathrm{Enc}_{\tau_2}(v_2) = j_2 \qquad \langle \mathrm{Enc}_{\tau_3}(v_3) = j_3 \rangle \qquad k = 2\langle + 1 \rangle}{\mathrm{Enc}_{(\tau_1, \tau_2 \langle, \tau_3 \rangle)\,\texttt{Tuple}_k}((v_1, v_2 \langle, v_3 \rangle)) = [j_1, j_2 \langle, j_3 \rangle]} \tag{152}$$

$$\frac{\begin{array}{c}\mathrm{Dom}\, r = \mathrm{Dom}\, \varrho \qquad j \in \mathrm{Str} \xrightarrow{\mathrm{fin}} \mathrm{BVal} \qquad \mathrm{Dom}\, j = \mathrm{Dom}\, \varrho \\ \mathrm{Enc}_{\varrho(lab)}(r(lab)) = j(lab) \text{ for each } lab \in \mathrm{Dom}\, \varrho\end{array}}{\mathrm{Enc}_{\{\varrho\}}(r) = j} \tag{153}$$

$$\frac{j \in \mathrm{JVal}}{\mathrm{Enc}_{\texttt{Value}}(j) = j} \tag{154}$$

Comments:

(140) The integer itself is the JSON number. Whether a host can hold every integer of IntRange exactly is not a matter for the definition (Appendix E).

(141) The sign of a zero is not kept: -0 encodes as the JSON number 0 (Section 8.1).

(142) JSON has no number for NaN or for an infinity. Such a float is encoded as `null`, which a `Float` port does not decode, while a `Maybe Float` port decodes it as `Nothing`.

(146) A value of type `Bytes` is sent as its byte sequence, which is not a JSON value: not a string, and not an array of numbers. Here each o_i on the left is an Int value and on the right the same integer.

(149) A `Just` has no encoding of its own. Hence the encoding of a `Maybe` value cannot tell `Nothing` from a `Just` whose argument encodes as `null`: in `Maybe (Maybe Int)`, both `Nothing` and `Just Nothing` encode as `null`, and so do `Nothing` and `Just ()` in `Maybe ()`.

(151) An array is encoded as the list of its elements. `Array.toList` is the basic value of Section D.4.9, bound to no identifier of the basis.

(153) The fields become the keys of an object, which has no order, so the order of fields is no part of what is sent.

(154) A value of type `Json.Encode.Value` is already a JSON value and is sent as it is. The side condition $j \in \mathrm{JVal}$ holds of every value of that type, so no byte sequence is ever sent through `Json.Encode.Value`.

Decoding

$\boxed{\mathrm{Dec}_\tau(j) = v}$

$$\frac{j \in \mathrm{IntRange}}{\mathrm{Dec}_{\texttt{Int}}(j) = j} \tag{155}$$

$$\frac{j \in \mathrm{JNum}}{\mathrm{Dec}_{\texttt{Float}}(j) = \mathrm{float}(j)} \tag{156}$$

$$\frac{}{\mathrm{Dec}_{\texttt{Bool}}(\texttt{true}) = \texttt{True}} \tag{157}$$

$$\frac{}{\mathrm{Dec}_{\texttt{Bool}}(\texttt{false}) = \texttt{False}} \tag{158}$$

$$\frac{j \in \mathrm{Str}}{\mathrm{Dec}_{\texttt{String}}(j) = j} \tag{159}$$

$$\frac{}{\mathrm{Dec}_{\texttt{Bytes}}(\mathrm{bytes}(o_1 \cdots o_k)) = \mathrm{binary}\,[o_1, \ldots, o_k]} \tag{160}$$

$$\frac{}{\mathrm{Dec}_{()}(\texttt{null}) = ()} \tag{161}$$

$$\frac{}{\mathrm{Dec}_{\tau\ \texttt{Maybe}}(\texttt{null}) = \texttt{Nothing}} \tag{162}$$

$$\frac{j \neq \texttt{null} \qquad \mathrm{Dec}_{\tau}(j) = v}{\mathrm{Dec}_{\tau\,\texttt{Maybe}}(j) = \texttt{Just}\ v} \tag{163}$$

$$\frac{\mathrm{Dec}_{\tau}(j_1) = v_1 \quad \cdots \quad \mathrm{Dec}_{\tau}(j_n) = v_n}{\mathrm{Dec}_{\tau\,\texttt{List}}([j_1, \ldots, j_n]) = [v_1, \ldots, v_n]} \tag{164}$$

$$\frac{\mathrm{Dec}_{\tau\,\texttt{List}}(j) = v \qquad \vdash \texttt{Array.fromList}\ v \Rightarrow v'}{\mathrm{Dec}_{\tau\,\texttt{Array}}(j) = v'} \tag{165}$$

$$\frac{\begin{array}{c} n \geq k \qquad k = 2\langle + 1\rangle \\ \mathrm{Dec}_{\tau_1}(j_1) = v_1 \qquad \mathrm{Dec}_{\tau_2}(j_2) = v_2 \qquad \langle \mathrm{Dec}_{\tau_3}(j_3) = v_3 \rangle \end{array}}{\mathrm{Dec}_{(\tau_1, \tau_2\langle, \tau_3\rangle)\,\texttt{Tuple}_k}([j_1, \ldots, j_n]) = (v_1, v_2\langle, v_3\rangle)} \tag{166}$$

$$\frac{\begin{array}{c} j \in \mathrm{Str} \xrightarrow{\mathrm{fin}} \mathrm{BVal} \qquad \mathrm{Dom}\,\varrho \subseteq \mathrm{Dom}\,j \qquad \mathrm{Dom}\,r = \mathrm{Dom}\,\varrho \\ \mathrm{Dec}_{\varrho(lab)}(j(lab)) = r(lab) \text{ for each } lab \in \mathrm{Dom}\,\varrho \end{array}}{\mathrm{Dec}_{\{\varrho\}}(j) = r} \tag{167}$$

$$\frac{j \in \mathrm{JVal}}{\mathrm{Dec}_{\texttt{Value}}(j) = j} \tag{168}$$

Comments:

(155) A JSON number is an Int exactly when it is an integer within IntRange: `1.0` and `-0` decode as `1` and `0`, while `1.5`, and an integer outside IntRange, do not decode. Since IntRange is a parameter of the implementation, so is the set of numbers a port of type `Int` accepts (Appendix E).

(156) Every JSON number decodes, integers among them, as the nearest floating-point number. A number too large for a finite floating-point number decodes as the infinity of its sign, by the rounding of floating-point constants. An incoming -0 is the number 0, and decodes as the float 0, not -0. No boundary value decodes as NaN.

(160) A byte sequence, and nothing else, decodes at `Bytes`: $\mathrm{Dec}_{\texttt{Bytes}}(j)$ is undefined for every JSON value j, so a string or an array of numbers sent to a `Bytes` port is rejected at the boundary.

(161) At an incoming port, the unit type accepts `null` and nothing else. The flags of a program are decoded otherwise when their type is the unit type (Chapter 9).

(163) The side condition $j \neq$ `null` makes `null` always decode as `Nothing`, at once, by rule (162): in `Maybe (Maybe Int)` it decodes as the outer `Nothing`, and `Just Nothing` is never the result of decoding. A value

other than `null` that does not decode at τ does not decode at `Maybe` τ either.

(166) An array longer than the tuple is accepted, and its further elements are ignored; a shorter array does not decode.

(167) An object with further keys is accepted, and they are ignored; an object that lacks the key of a field does not decode. Which missing field an implementation reports, and how, is outside the definition.

(168) Every JSON value decodes at `Json.Encode.Value`. A boundary value in which a byte sequence occurs is not a JSON value, and does not decode there: it is rejected.

There is no rule for any other pair of a type and a boundary value: a string does not decode as an Int, nor a number as a Bool, nor an object as a tuple, nor a byte sequence as a String or a list. The rules give the following properties.

> For every τ with $\mathrm{Port}(\tau)$, Enc_τ and Dec_τ are partial functions: each value has at most one encoding, and each boundary value at most one decoding, at τ. Every value that elaboration admits as a payload of type τ has an encoding at τ. If $\mathrm{Enc}_\tau(v) = j$, and v has no component that is NaN, an infinity or a negative zero, and no component `Just` v' whose argument encodes as `null`, then $\mathrm{Dec}_\tau(j) = v''$, where v'' is v with each array replaced by the value of `Array.fromList` applied to the list of its elements.

The value of a port is a basic value of arity 1 that the dynamic semantics of modules binds to the port name (Chapter 7). For an outgoing port p of payload type τ, that is the basic value $\mathrm{outport}(p, \tau)$, and for an incoming port the basic value $\mathrm{inport}(p, \tau)$. The class BasVal of Chapter 6 includes these values, defined here, for every port name and every payload type. Their meaning is

$$\begin{aligned} \mathrm{APPLY}(\mathrm{outport}(p,\tau), v) &= \{\mathrm{leaf}(\mathrm{out}\, p, j) \mid \mathrm{Enc}_\tau(v) = j\}, \\ \mathrm{APPLY}(\mathrm{inport}(p,\tau), v) &= \{\mathrm{leaf}(\mathrm{in}(p,\tau), v)\}. \end{aligned}$$

By the property above, the first set has exactly one member for every value an outgoing port is applied to: a value is encoded when the command is made. A value sent to an incoming port is decoded when it arrives (Section 8.8).

Figure 25 collects what can and cannot be sent over a port, for each form of type, in agreement with the predicate Port of Section 5.4. A type that is not a payload type makes a port declaration fail to elaborate; no such port exists at run time.

A type alias stands for the type it expands to. `Bool` and `Maybe` are custom types, and the only custom types a port admits.

Elm type	Port?	Sent as (Enc)	Accepted (Dec)
`Int`	yes	the number	a number that is an integer in IntRange
`Float`	yes	the number; `null` for NaN and the infinities	any number, rounded
`Bool`	yes	`true` or `false`	`true` or `false` only
`String`	yes	the string	a string only
`Bytes`	yes	a byte sequence	a byte sequence only
`()`	yes	`null`	`null` only (flags: Chapter 9)
`Maybe` *ty*	if *ty* is	`null` for `Nothing`; *v* as *ty* for `Just` *v*	`null` as `Nothing`; anything else as *ty*, under `Just`
`List` *ty*	if *ty* is	an array	an array whose elements decode as *ty*
`Array` *ty*	if *ty* is	an array	an array whose elements decode as *ty*
(*ty*$_1$, *ty*$_2$), (*ty*$_1$, *ty*$_2$, *ty*$_3$)	if each *ty*$_i$ is	an array of 2 or 3 elements	an array at least as long; further elements ignored
closed record { *lab*$_1$: *ty*$_1$, ... }	if each *ty*$_i$ is	an object with exactly the fields as keys	an object with at least those keys; further keys ignored
`Json.Encode.Value`	yes	itself	any JSON value; no byte sequence
`Char`	no		
a function type	no		
a type variable	no		
an extensible record	no		
any other custom type (`Result`, `Order`, `Dict`, `Set`, and those of the program)	no		
any other abstract type (`Cmd`, `Sub`, `Task`, `Process.Id`, `Program`)	no		

Figure 25: What Can and Cannot Be Sent over a Port

8.8 Messages

A leaf command gives messages from the inputs its manager receives: a task leaf from the success value of the process that performs its task, an incoming port leaf from the boundary values that arrive at its port. The rules below define the message given, through the functions of the maps around the leaf.

Messages $\boxed{\vdash c \triangleleft x \Rightarrow v}$

$$\frac{}{\vdash \mathrm{leaf}(\mathrm{task}, tk) \triangleleft v \Rightarrow v} \tag{169}$$

$$\frac{\mathrm{Dec}_\tau(j) = v' \qquad \vdash v\, v' \Rightarrow v''}{\vdash \mathrm{leaf}(\mathrm{in}(p, \tau), v) \triangleleft (p, j) \Rightarrow v''} \tag{170}$$

$$\frac{\vdash cm/sb \triangleleft x \Rightarrow v' \qquad \vdash v\, v' \Rightarrow v''}{\vdash \mathrm{map}(v, cm/sb) \triangleleft x \Rightarrow v''} \tag{171}$$

Comments:

(169) The input v is the value with which the process performing tk succeeds. The task of a leaf made by `Task.perform` or `Task.attempt` cannot fail. For `Task.perform` it is the task given, mapped, whose error type is `Never`, of which there is no value. For `Task.attempt` it is the task given with `Task.onError` applied, which turns a failure into a success whose value is the message made from `Err`. Either way the task that `Perform` carries has the error type `Never` (Section C.2). So every such process that ends gives a message.

(170) The port name of the input must be that of the manager, so a boundary value arriving at one port gives no message through a leaf of another. When $\mathrm{Dec}_\tau(j)$ is undefined, no rule applies, and the leaf gives no message.

(171) The innermost function is applied first. There is no rule for a leaf of an outgoing port: such a leaf gives no message, and the functions of the maps around it are never applied. The messages that the leaves of the managers of the platform give are defined by the platform.

A boundary value j that arrives at an incoming port p of payload type τ is *accepted* if $\mathrm{Dec}_\tau(j)$ is defined, and otherwise *rejected*. A rejected value is refused at the boundary: the program never sees it, and it is unaffected. When j is accepted and sb is the current subscription value of the program, every member c of $\mathrm{Flat}(sb)$ for which $\vdash c \triangleleft (p, j) \Rightarrow v$ gives the message v, once for each time it occurs in $\mathrm{Flat}(sb)$. Whether a value is accepted depends on its port alone, not on the current subscriptions; an accepted value that no current subscription receives gives no message. When the messages are handled, and in which order, is defined in Chapter 9.

Chapter 9

Programs

This chapter defines what a program is and what it means. A program is a main module together with the modules it imports. It is elaborated and evaluated module by module, in a basis that pairs a static basis (Chapter 5) with a dynamic basis (Chapter 7); elaboration finds the type of `main` in the main module and the ports of the program, and evaluation finds the value of `main`. That value is a program value (Chapter 8), and the program runs on the platform (Chapter 1), which supplies the primitives and managers beyond those of tasks and ports. The running of a program is defined as a labelled transition system, whose configurations hold the current model, the messages awaiting `update`, the processes, the current subscriptions and the values awaiting delivery on outgoing ports. The sentences of the chapter infer that a program elaborates to a type and a port environment, that it evaluates to a value, that a program value started with flags begins in a configuration, and that a configuration moves to another, with or without an event. The meaning of a program is the set of sequences of events at its boundary that it can produce.

9.1 Syntax of Programs

The phrase class Program of programs is defined as follows:

$$\textit{program} \quad ::= \quad \textit{module}\ \langle \textit{program} \rangle$$

A program is thus a non-empty sequence of modules; its last module is its *main module*. A module *module imports* a module *modid* if an import of *module*, written or default (Figure 11), names *modid*. Programs are subject to the following restrictions, in addition to the restrictions of Section 3.7 on each module.

- No two modules of a program have the same module identifier, and no module of a program has the identifier of a module of the initial basis (Appendix C).

- Each module of a program imports only modules of the initial basis and modules that stand before it in the program.
- Every module of a program other than the main module is imported, directly or through other modules, by the main module.
- The main module has a top-level value declaration that binds `main`. That declaration has the form `main` ⟨: *ty*⟩ `=` *exp*, binding `main` alone, and `main` does not occur free in *exp*, so that `main` does not refer to itself, directly or through a delayed occurrence.

Thus the modules of a program are the main module and every module it imports directly or indirectly; the import relation among them has no cycle; and every module stands after the modules it imports. A program as written is the finite set of these modules. Any sequence of that set that meets the restrictions is a program in the sense of the grammar; the main module comes last in every such sequence, and all such sequences have the same meaning (Section 9.3).

9.2 Elaboration of Programs

Until now, the name Basis and the metavariable B have stood for a static basis in Chapter 5 and for a dynamic basis in Chapter 7; those chapters never needed both at once. In this chapter the class Basis of Chapter 5 is called StaticBasis, with metavariable B_{STAT}, and that of Chapter 7 is called DynamicBasis, with metavariable B_{DYN}; a basis is a pair of the two:

$$B \text{ or } (B_{\text{STAT}}, B_{\text{DYN}}) \in \text{Basis} = \text{StaticBasis} \times \text{DynamicBasis}.$$

The *initial basis* B_0 is the pair of the initial static basis of Appendix C and the initial dynamic basis of Appendix D. Modification of static bases is component by component, the set component by union, as for contexts (Section 4.3):

$$(T, ME, PE) + (T', ME', PE') = (T \cup T',\ ME + ME',\ PE + PE');$$

a dynamic basis is a module environment, modified as a finite map.

The rules of this section and the next allow sentences of the forms

$$B_{\text{STAT}} \vdash_{\text{STAT}} \mathit{program} \Rightarrow (\tau, PE), \qquad B_{\text{DYN}} \vdash_{\text{DYN}} \mathit{program} \Rightarrow v,$$
$$B \vdash \mathit{program} \Rightarrow (\tau, PE, v)$$

to be inferred, where B_{STAT} is a static basis, B_{DYN} a dynamic basis, B a basis, τ a type, PE a port environment (Section 5.1) and v a value. We write $\vdash_{\text{STAT}}$ for the elaboration of programs, $\vdash_{\text{DYN}}$ for their evaluation, and $\vdash$ alone for the sentence that combines the two. The first sentence is pronounced "*program*

elaborates to τ and PE in B_{STAT}": τ is the type of `main` in the main module, and PE holds the ports of the program. The second is pronounced "*program* evaluates to v in B_{DYN}": v is the value of `main`. The third is pronounced "*program* elaborates to τ and PE, and evaluates to v, in B". Premises also use the sentences about modules of Chapters 5 and 7, in their forms there; other hypotheses are side conditions.

The type of `main` is the instance of its type scheme that takes `Float` for each bound type variable of class number. It is the type of the default copy of the declaration of `main`, the copy that number specialisation makes of every top-level declaration, taking `Float` for each of its variables of class number (Section 6.2). For a type scheme $\sigma = \forall \xi^{(k)}.\tau$, let $\alpha_1, \ldots, \alpha_l$ be those of the bound variables $\xi_1, \ldots, \xi_k$ that are type variables of class number; then

$$\mathrm{Dflt}\,\sigma = \tau\{\texttt{Float}, \ldots, \texttt{Float}/\alpha_1, \ldots, \alpha_l\}.$$

The substitution respects classes, since $\texttt{Float} \in \mathrm{Adm}_{\mathrm{number}}$. In the rules below, `Program` is the type name of `Platform.Program`, of arity 3, and `Node` that of `VirtualDom.Node`, of arity 1, of which `Html` is an alias (Appendix C); $\mathrm{Manager}_{\mathrm{plat}}$ is the set of the names of the managers of the platform (Section 8.1). In each rule, the premise $B_{\mathrm{STAT}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{STAT}}, E$ is inferred by a principal derivation for the module (Section 5.6), so that E, and the scheme σ of `main` in it, are principal.

Elaboration of Programs $\boxed{B_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{program} \Rightarrow (\tau, PE)}$

$$\frac{\begin{array}{c} B_{\mathrm{STAT}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{STAT}}, E \\ \mathrm{Dom}\,(PE \text{ of } B'_{\mathrm{STAT}}) \cap \big(\mathrm{Dom}\,(PE \text{ of } B_{\mathrm{STAT}}) \cup \mathrm{Manager}_{\mathrm{plat}}\big) = \emptyset \\ B_{\mathrm{STAT}} + B'_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{program} \Rightarrow (\tau, PE) \end{array}}{B_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{module}\ \mathit{program} \Rightarrow (\tau, PE)} \tag{172}$$

$$\frac{\begin{array}{c} B_{\mathrm{STAT}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{STAT}}, E \\ \mathrm{Dom}\,(PE \text{ of } B'_{\mathrm{STAT}}) \cap \big(\mathrm{Dom}\,(PE \text{ of } B_{\mathrm{STAT}}) \cup \mathrm{Manager}_{\mathrm{plat}}\big) = \emptyset \\ E(\texttt{main}) = (\sigma, \mathrm{v}) \qquad \sigma = \forall \xi^{(k)}.(\tau_1, \tau_2, \tau_3)\texttt{Program} \qquad \mathrm{Port}(\tau_1) \end{array}}{B_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{module} \Rightarrow \big(\mathrm{Dflt}\,\sigma,\ PE \text{ of } (B_{\mathrm{STAT}} + B'_{\mathrm{STAT}})\big)} \tag{173}$$

$$\frac{\begin{array}{c} B_{\mathrm{STAT}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{STAT}}, E \\ \mathrm{Dom}\,(PE \text{ of } B'_{\mathrm{STAT}}) \cap \big(\mathrm{Dom}\,(PE \text{ of } B_{\mathrm{STAT}}) \cup \mathrm{Manager}_{\mathrm{plat}}\big) = \emptyset \\ E(\texttt{main}) = (\sigma, \mathrm{v}) \qquad \mathrm{Dflt}\,\sigma = \tau\,\texttt{Node} \end{array}}{B_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{module} \Rightarrow \big(\tau\,\texttt{Node},\ PE \text{ of } (B_{\mathrm{STAT}} + B'_{\mathrm{STAT}})\big)} \tag{174}$$

Comments:

(172) Each module is elaborated in the static basis that the modules before it have built, so it finds there the interface of every module it imports, and the type names it generates are new (rule (46)). The side condition makes port names global to the program: no two ports of a program share a name, even in different modules, and no port has the name of a manager of the platform; a program with such a clash does not elaborate.

(173) Rules (173) and (174) cover the one production, since `main` may have either of two forms of type. Only the `main` of the main module is constrained; a `main` in another module is an ordinary value. The scheme of `main` is taken from E, the environment of all the declarations of the module, not from its interface, so `main` need not be exposed. A bound variable of class number in the scheme of `main` is `Float` in its type: with `init = \() -> (0, Cmd.none)` and no other constraint on the model, the model type is `Float`, and the value of `main` is that of its default copy, at `Float` (rule (176)). The flags type τ_1 is a port type, and is taken from the scheme before the default, so it contains no type variable. Thus `main : Program (Int -> Int) Model Msg` and `main : Program flags Model Msg`, with `flags` a type variable, do not elaborate, and nor does an unannotated `main` whose `init` is `\n -> (n + 1, Cmd.none)`, whose flags type is a variable of class number. The unit type is a port type, and as flags it is treated specially (Section 9.4). Since `Bytes` is a port type, flags may be, or contain, values of type `Bytes`. The model type τ_2 and the message type τ_3 are unconstrained. Some implementations admit further types of flags, which are not port types (Appendix E).

(174) A `main` of type `Html msg` or `Svg msg` has a type τ `Node`, since both are aliases. No function type is of either form, so a `main` with parameters does not elaborate.

Since the derivations for the modules of a program are principal, a flags type that the program leaves undetermined is a type variable in the scheme of `main`, which is not a port type: an unannotated `main = Platform.worker { init = \_ -> (0, Cmd.none), ... }`, whose `init` ignores its flags, does not elaborate, although a derivation that took the flags to be () would satisfy the rule.

There is no rule for a program one of whose modules does not elaborate. Such a program has no meaning, and no part of it is evaluated (Section 9.3).

9.3 Evaluation of Programs

The modules of a program are evaluated in the order in which they are elaborated, each in the dynamic basis that the modules before it have built. The syntax evaluated is the reduced syntax of Section 7.1, which records the direction and payload type that elaboration finds for each port, after number specialisation (Section 6.2), which makes the default copy of every top-level declaration besides the copies its uses need; when the declaration quantifies no variable of class number, the default copy is the declaration itself. In rule (176), $VE(\texttt{main})$ is the value bound by the default copy of the declaration of `main`, whose type is $\mathrm{Dflt}\,\sigma$ for the scheme σ of `main`, the type found by rule (173) or (174).

Evaluation of Programs $\boxed{B_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} \mathit{program} \Rightarrow v}$

$$\frac{B_{\mathrm{DYN}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{DYN}}, VE \qquad B_{\mathrm{DYN}} + B'_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} \mathit{program} \Rightarrow v}{B_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} \mathit{module}\ \mathit{program} \Rightarrow v} \tag{175}$$

$$\frac{B_{\mathrm{DYN}} \vdash \mathit{module} \Rightarrow B'_{\mathrm{DYN}}, VE \qquad v = VE(\texttt{main})}{B_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} \mathit{module} \Rightarrow v} \tag{176}$$

Comments:

(175) A module is evaluated once, after every module it imports, and its top-level values are evaluated then, whether or not they are used: every top-level declaration has its default copy, even when nothing uses it. A `main` in a module other than the main module is evaluated in the same way, once, as an ordinary value.

(176) The value of `main` is taken from VE, the value environment of all the top-level declarations of the module (rule (113)), not from the basis B'_{DYN}, which holds only what the module exposes; so `main` need not be exposed. No module imports the main module, and B'_{DYN} is not used. If a top-level value of any module has no value (it reaches `Debug.todo`, or forces a delayed reference to itself), no sentence about the program can be inferred, and the program has no meaning.

Programs $\boxed{B \vdash \mathit{program} \Rightarrow (\tau, PE, v)}$

$$\frac{B_{\mathrm{STAT}} \text{ of } B \vdash_{\mathrm{STAT}} \mathit{program} \Rightarrow (\tau, PE) \qquad B_{\mathrm{DYN}} \text{ of } B \vdash_{\mathrm{DYN}} \mathit{program} \Rightarrow v}{B \vdash \mathit{program} \Rightarrow (\tau, PE, v)} \tag{177}$$

Comments:

(177) A program is elaborated as a whole before any part of it is evaluated: if one module fails to elaborate, the static premise fails, and nothing is evaluated, not even the modules before it. Elaboration may choose the new type names of a module freely (rule (46)), but neither PE, whose types are built from type names of the initial basis alone, nor v depends on the choice.

A program *program has a meaning* when $B_0 \vdash \mathit{program} \Rightarrow (\tau, PE, v)$ for some τ, PE and v. Since a program is a set of modules, the following property, which may easily be demonstrated, is needed for this to make sense.

> Let *program* and *program*$'$ be sequences of the same modules, both meeting the restrictions of Section 9.1. Then the sentence $B_0 \vdash \mathit{program} \Rightarrow (\tau, PE, v)$ can be inferred if and only if the sentence $B_0 \vdash \mathit{program}' \Rightarrow (\tau', PE, v)$ can, where τ' is τ with the type names that the modules generate renamed.

9.4 Execution

Let $B_0 \vdash \mathit{program} \Rightarrow (\tau, PE, v)$. When τ is $\tau'\,\texttt{Node}$, the program is a view, whose meaning is given in Section 9.6. Otherwise $\tau = (\tau_1, \tau_2, \tau_3)\texttt{Program}$, where τ_1 is the *flags type*, and v is a program value $\mathrm{prog}\, r$ (Section 8.5). This section defines how a program value runs. Throughout, PE is the port environment of the program, and the record r is fixed: the rules apply its fields `init`, `update` and `subscriptions`, written $r(\texttt{init})$ and so on.

A program runs on a platform. Besides the primitives $\mathrm{Prim}_{\mathrm{plat}}$, their outcomes and the managers $\mathrm{Manager}_{\mathrm{plat}}$ (Chapter 8), the platform provides a set $\mathrm{Event}_{\mathrm{plat}}$ of *platform events*, and its *platform steps*, defined in Section 9.4.3. All of these are parameters of the platform; the definition fixes the manager of tasks, the ports, and the program loop.

9.4.1 Configurations and Events

For a set A, $A^* = \bigcup_{n \geq 0} A^n$ is the set of finite sequences of members of A. The empty sequence is nil, and $x \cdot y$ is the concatenation of the sequences x and y; a member of A is also a sequence of length one, and $x_1 \cdots x_n$ is $x_1 \cdot \ldots \cdot x_n$. A finite multiset of members of A is a finite map from A to the positive integers, giving the multiplicity of each member; multisets are written as in Section 8.2. The objects of execution are shown in Figure 26.

A configuration $K = (s, v, M, P, sb, Q, H)$ has the following components.

- s is the present time, in milliseconds since the program started.
- v is the *model*, the value that the latest `update`, or `init`, produced.
- M is the *message queue*, the messages awaiting `update`.

$$
\begin{aligned}
s \text{ or } d \in \text{Time} &= \{x \in \mathbb{R} \mid x \geq 0\} \\
K \in \text{Config} &= \text{Time} \times \text{Val} \times \text{Val}^* \times \text{ProcMap} \\
&\quad \times \text{SubVal} \times \text{OutQueue} \times \text{Pending} \\
P \in \text{ProcMap} &= \text{ProcId} \xrightarrow{\text{fin}} \text{Proc} \\
\in \text{Proc} &= \big(\text{TaskVal} \times \text{Time} \times (\text{CmdVal} \cup \{\text{spawned}\})\big) \\
&\quad \cup \{\text{ended}\} \\
Q \in \text{OutQueue} &= \text{Var} \xrightarrow{\text{fin}} \text{BVal}^* \\
H \in \text{Pending} &= \text{CmdVal} \xrightarrow{\text{fin}} \{1, 2, \ldots\} \\
\ell \in \text{Event} &= (\{\text{in}\} \times \text{Var} \times \text{BVal}) \\
&\quad \cup (\{\text{out}\} \times \text{Var} \times \text{BVal}) \cup \text{Action} \\
&\quad \cup (\{\text{wait}\} \times \text{Time}) \cup \text{Event}_{\text{plat}}
\end{aligned}
$$

Figure 26: Semantic Objects of Execution

- P holds every process started so far. If $P(pid) = (tk, s', o)$, the process named by *pid* is *live*, is reducing *tk*, and took its latest step (or started) at time s'; its *origin* o is the leaf command whose task it performs, or spawned if it was started by spawn. If $P(pid) = \text{ended}$, the process has ended or been killed.
- sb is the current subscription value, the value of `subscriptions` applied to the model.
- $Q(p)$ is the sequence of boundary values awaiting delivery on the outgoing port p.
- H holds the leaf commands handed to the managers of the platform and not yet acted on.

The event $\text{in}(p, j)$ is the host's sending of j on the incoming port p; $\text{out}(p, j)$ is the program's delivery of j on the outgoing port p; an action a (Section 8.4) is a primitive performed by a process; $\text{wait}\, d$, with $d > 0$, is the passing of d milliseconds; and a platform event is whatever the platform makes observable of its managers.

For a manager m and a value v, the leaf commands (and leaf subscriptions) of m with argument v are

$$\text{Leaf}(m, v) = \{\text{map}(v_n, \cdots \text{map}(v_1, \text{leaf}(m, v)) \cdots) \mid n \geq 0\}.$$

The outgoing queues at the start of a program are

$$Q_{\text{init}} = \{p \mapsto \text{nil}\,;\ PE(p) = (\text{out}, \tau) \text{ for some } \tau\}.$$

For a type τ with $\text{Port}(\tau)$ and a boundary value j, the *flags value* of j at τ is

$$\text{Flags}_\tau(j) = \begin{cases} () & \text{if } \tau = (), \\ \text{Dec}_\tau(j) & \text{otherwise,} \end{cases}$$

which is undefined when $\mathrm{Dec}_\tau(j)$ is.

9.4.2 Dispatch and Start

The rules of this section and the next allow sentences of the forms

$$s, P, Q, H \vdash L \Rightarrow (P', Q', H'), \qquad PE, \tau_1, j \vdash \mathrm{prog}\, r \Rightarrow K, \qquad K \xrightarrow{\langle \ell \rangle} K'$$

to be inferred. In the first, L is a sequence of leaf commands, and the sentence is pronounced "dispatching L at time s turns P, Q, H into P', Q', H'". The second is pronounced "prog r, started with the flags j, begins in K". The third is pronounced "K moves to K'", and, when the event ℓ is present, "with ℓ"; a transition without an event is *silent*. Premises also use the sentences of Chapter 8 and the sentence $\vdash v_1\, v_2 \Rightarrow v$ of the application of values (Section 6.7); other hypotheses are side conditions. The convention for options is the same as for the Core static semantics. There are no states and no exception packets: the configuration is the whole of what changes, and every rule is given in full.

A command is *dispatched* by dispatching a sequence that holds its leaf commands, each as many times as Flat gives it, in any order. Which order is no part of the meaning, by the effect order convention of Section 8.5.

Dispatch $\boxed{s, P, Q, H \vdash L \Rightarrow (P', Q', H')}$

$$\frac{}{s, P, Q, H \vdash \mathrm{nil} \Rightarrow (P, Q, H)} \tag{178}$$

$$\frac{c \in \mathrm{Leaf}(\mathrm{task}, tk) \qquad pid \notin \mathrm{Dom}\, P \qquad s,\ P + \{pid \mapsto (tk, s, c)\},\ Q, H \vdash L \Rightarrow (P', Q', H')}{s, P, Q, H \vdash c \cdot L \Rightarrow (P', Q', H')} \tag{179}$$

$$\frac{c \in \mathrm{Leaf}(\mathrm{out}\, p, j) \qquad s, P,\ Q + \{p \mapsto Q(p) \cdot j\},\ H \vdash L \Rightarrow (P', Q', H')}{s, P, Q, H \vdash c \cdot L \Rightarrow (P', Q', H')} \tag{180}$$

$$\frac{c \in \mathrm{Leaf}(m, v) \qquad m \in \mathrm{Manager}_{\mathrm{plat}} \qquad s, P, Q,\ H \uplus \{\!|\, c \,|\!\} \vdash L \Rightarrow (P', Q', H')}{s, P, Q, H \vdash c \cdot L \Rightarrow (P', Q', H')} \tag{181}$$

Comments:

(179) Each task leaf starts a process of its own, with an identifier that names no process started before, live or ended; a leaf that occurs twice in a command starts two processes, and performs its task twice. The process keeps the leaf as its origin, since the message it gives, when its task succeeds, comes from the leaf (rule (188)).

(180) The boundary value was computed when the command was made (Section 8.7); dispatch only appends it to the queue of its port. Since $Q(p)$ is defined only for the outgoing ports of the program, p is one of them. The values of one command for one port join the queue in the order of L, which is unspecified: with `Cmd.batch [send 1, send 2]`, either value may be delivered first.

(181) What the managers of the platform do with the leaves handed to them is given by the platform steps (Section 9.4.3). No rule dispatches a leaf of any other manager: a leaf of an incoming port is a subscription, never a command, and there are no effect modules in Standard Elm.

Starting a Program

$\boxed{PE, \tau_1, j \vdash \text{prog}\, r \Rightarrow K}$

$$\frac{\begin{array}{c}\text{Flags}_{\tau_1}(j) = v_1 \qquad \vdash r(\texttt{init})\, v_1 \Rightarrow (v, cm) \\ \text{Flat}(cm) = \{\!|\, c_1, \ldots, c_n \,|\!\} \qquad 0,\ \{\},\ Q_{\text{init}},\ \{\} \vdash c_1 \cdots c_n \Rightarrow (P, Q, H) \\ \vdash r(\texttt{subscriptions})\, v \Rightarrow sb\end{array}}{PE, \tau_1, j \vdash \text{prog}\, r \Rightarrow (0, v, \text{nil}, P, sb, Q, H)} \tag{182}$$

Comments:

(182) The host starts a program with a boundary value j as its flags; when the host gives no flags, j is `null`. With flags of the unit type every boundary value is accepted, whereas a port of the unit type accepts only `null`. When $\text{Flags}_{\tau_1}(j)$ is undefined, as for $j = 1.5$ with flags type `Int`, no rule applies, there is no configuration in which the program begins, and the program does not start; how an implementation reports this is outside the definition (Appendix E). The commands of `init` are dispatched, and the subscriptions computed from the initial model, before any message is handled. The empty map $\{\}$ serves here as the empty process map and the empty multiset.

9.4.3 Transitions

A *platform step* $K \xrightarrow{\langle \ell \rangle}_{\text{plat}} K'$ is given by the platform. Its event ℓ, if present, is a platform event. A platform step changes nothing but the components M and H of K: it removes leaf commands from H, and it appends to M messages that, as the platform defines, are given by the leaf commands it removes or by the members of $\text{Flat}(sb)$ whose manager is in $\text{Manager}_{\text{plat}}$ (Section 8.8). Whether a platform step exists may depend on the present time s.

Transitions $\boxed{K \xrightarrow{\langle \ell \rangle} K'}$

$$\frac{\begin{array}{c}\vdash r(\texttt{update})\, v_1 \Rightarrow v_2 \qquad \vdash v_2\, v \Rightarrow (v', cm) \\ \mathrm{Flat}(cm) = \{\!|\, c_1, \ldots, c_n \,|\!\} \qquad s, P, Q, H \vdash c_1 \cdots c_n \Rightarrow (P', Q', H') \\ \vdash r(\texttt{subscriptions})\, v' \Rightarrow sb'\end{array}}{(s, v, v_1 \cdot M, P, sb, Q, H) \to (s, v', M, P', sb', Q', H')} \tag{183}$$

$$\frac{\begin{array}{c}P(pid) = (tk, s', o) \qquad tk \xrightarrow{\langle a \rangle} tk' \\ \langle a = \pi(v_1 \cdots v_k) \Downarrow tk'' \qquad \pi \in \mathrm{Prim}_{\mathrm{plat}} \rangle\end{array}}{(s, v, M, P, sb, Q, H) \xrightarrow{\langle a \rangle} (s, v, M, P + \{pid \mapsto (tk', s, o)\}, sb, Q, H)} \tag{184}$$

$$\frac{\begin{array}{c}P(pid) = (tk, s', o) \qquad tk \xrightarrow{a} tk' \qquad a = \mathrm{sleep}(v') \Downarrow tk'' \\ \text{if } v' \in \mathrm{Flt}_{\mathrm{fin}} \text{ then } s \geq s' + \max(0, v')\end{array}}{(s, v, M, P, sb, Q, H) \xrightarrow{a} (s, v, M, P + \{pid \mapsto (tk', s, o)\}, sb, Q, H)} \tag{185}$$

$$\frac{\begin{array}{c}P(pid) = (tk, s', o) \qquad tk \xrightarrow{a} tk' \qquad a = \mathrm{spawn}(tk_0) \Downarrow \mathrm{succeed}\, pid' \\ pid' \notin \mathrm{Dom}\, P \qquad P' = P + \{pid \mapsto (tk', s, o),\ pid' \mapsto (tk_0, s, \mathrm{spawned})\}\end{array}}{(s, v, M, P, sb, Q, H) \xrightarrow{a} (s, v, M, P', sb, Q, H)} \tag{186}$$

$$\frac{\begin{array}{c}P(pid) = (tk, s', o) \qquad tk \xrightarrow{a} tk' \qquad a = \mathrm{kill}(pid') \Downarrow tk'' \\ P' = (P + \{pid' \mapsto \mathrm{ended}\}) + \{pid \mapsto (tk', s, o)\}\end{array}}{(s, v, M, P, sb, Q, H) \xrightarrow{a} (s, v, M, P', sb, Q, H)} \tag{187}$$

$$\frac{P(pid) = (\mathrm{succeed}\, v', s', c) \qquad \vdash c \triangleleft v' \Rightarrow v''}{(s, v, M, P, sb, Q, H) \to (s, v, M \cdot v'', P + \{pid \mapsto \mathrm{ended}\}, sb, Q, H)} \tag{188}$$

$$\frac{\begin{array}{c}P(pid) = (tk, s', o) \\ tk = \mathrm{fail}\, v' \text{ or } (tk = \mathrm{succeed}\, v' \text{ and } o = \mathrm{spawned})\end{array}}{(s, v, M, P, sb, Q, H) \to (s, v, M, P + \{pid \mapsto \mathrm{ended}\}, sb, Q, H)} \tag{189}$$

$$\frac{\begin{array}{c}PE(p) = (\mathrm{in}, \tau) \qquad \mathrm{Dec}_\tau(j) = v' \qquad \mathrm{Flat}(sb) = \{\!|\, c_1, \ldots, c_n \,|\!\} \\ \text{for } 1 \leq i \leq n : \vdash c_i \triangleleft (p, j) \Rightarrow M_i, \\ \text{or } M_i = \mathrm{nil} \text{ and } \vdash c_i \triangleleft (p, j) \Rightarrow v'' \text{ for no } v''\end{array}}{(s, v, M, P, sb, Q, H) \xrightarrow{\mathrm{in}(p,j)} (s, v, M \cdot M_1 \cdots M_n, P, sb, Q, H)} \tag{190}$$

$$\frac{PE(p) = (\text{in}, \tau) \qquad \text{Dec}_\tau(j) \text{ is undefined}}{K \xrightarrow{\text{in}(p,j)} K} \tag{191}$$

$$\frac{Q(p) = j \cdot J}{(s, v, M, P, sb, Q, H) \xrightarrow{\text{out}(p,j)} (s, v, M, P, sb, Q + \{p \mapsto J\}, H)} \tag{192}$$

$$\frac{d > 0}{(s, v, M, P, sb, Q, H) \xrightarrow{\text{wait } d} (s + d, v, M, P, sb, Q, H)} \tag{193}$$

$$\frac{K \xrightarrow{\langle \ell \rangle}_{\text{plat}} K'}{K \xrightarrow{\langle \ell \rangle} K'} \tag{194}$$

Comments:

(183) This is the only rule that takes a message from the queue or changes the model. The message at the head of the queue is applied, with the model, to `update`; the commands of the result are dispatched, and the subscriptions recomputed from the new model, whether or not it has changed, all in the one step. If the application of `update` has no result (it reaches `Debug.todo`), the rule never applies again, and no further message is handled, though processes, ports and the platform go on taking steps.

(184) A process takes one reduction of its task at a time, and the steps of different processes, and every other transition, may come between any two of them. A process may thus be suspended at any andThen; a program may rely on nothing more. Rules (184) to (187) record the present time as the time of the process's latest step. An action of a primitive of the platform acts on the world as the platform defines.

(185) The process reached $\text{prim}(\text{sleep}, v')$ by its step at time s', so the action takes place no sooner than $\max(0, v')$ milliseconds later; for an infinite or NaN delay, no condition is imposed. The definition sets no time by which the action must take place.

(186) The new process names a process never started before, and its origin is spawned: its outcome goes to no other process, and it gives no message (rule (189)).

(187) The killed process takes no further step, and if it was performing the task of a leaf it gives no message. Killing a process that has ended changes nothing. When a process kills itself, the second modification overrides the first, so it is not ended and goes on with tk'.

(188) A process whose origin is a leaf command ends, when its task succeeds, by adding the message of its leaf (rule (169)) to the end of the queue. When the functions of the maps around the leaf give no result, the process never ends.

(189) A process whose task fails ends, whatever its origin, and nothing else happens: there is no error, and no message.

(190) An incoming value is an event of its own, so it is never handled in the middle of another step; its messages join the end of the queue, and are handled by later updates. Each leaf subscription on p gives one message, through the functions of the maps around it; a leaf on another port, or of another manager, gives none, and so does a leaf whose functions give no result. The order of the messages of one value follows the enumeration of Flat(sb), and is unspecified. A value accepted while no subscription is current gives no message.

(191) A boundary value that does not decode at the type of the port is rejected: the program never sees it, and is unchanged. Thus with `port recv : (String -> msg) -> Sub msg`, the value 1337 is rejected. For every incoming port and every boundary value one of rules (190) and (191) applies, so the host may send at any time. A name that is not an incoming port of the program has no transition: sending on it is outside the definition.

(192) The values of each port are delivered in the order in which they joined its queue, so a value sent by one update is delivered before a value that a later update sends on the same port. Delivery happens after the step that queued the value, at a time the definition does not fix, and in no fixed order across ports. The event records only that the program has delivered j; whether the host has a subscriber, and what it does with j, are outside the definition.

(193) Time passes only in this rule, and every other transition takes no time. Since this rule applies to every configuration, the definition requires no step to happen by any time: only the lower bound of rule (185), and what the platform imposes on its own steps, relate the steps of a program to time.

(194) The managers of the platform, such as those of `Time` or `Http`, act through these steps, within the limits stated above. A platform with no managers has no platform steps.

The rules give the following guarantees, and no others.

Messages are handled one at a time, by rule (183), in the order in which they joined the message queue. Each update applies `update` to the model that the previous update produced, or to the model of `init` for the first. The commands of an update are dispatched in the step of that update, so the effects of an earlier update are dispatched before those of a later one. Apart from these, and from the order of the values of each outgoing port, the steps of distinct processes, the arrival of incoming values and platform steps, the delivery of outgoing values, and the update steps interleave in every way the rules allow.

9.5 Observable Behaviour

Let $B_0 \vdash \mathit{program} \Rightarrow (\tau, PE, \mathrm{prog}\, r)$ with $\tau = (\tau_1, \tau_2, \tau_3)\texttt{Program}$, and let j be a boundary value. An *execution* of *program* with flags j is a finite sequence of transitions

$$K_0 \xrightarrow{\langle \ell_1 \rangle} K_1 \xrightarrow{\langle \ell_2 \rangle} \cdots \xrightarrow{\langle \ell_n \rangle} K_n \qquad (n \geq 0),$$

where $PE, \tau_1, j \vdash \mathrm{prog}\, r \Rightarrow K_0$. The *observable* events are those of the forms $\mathrm{in}(p, j')$, $\mathrm{out}(p, j')$ and $\mathrm{wait}\, d$, the platform events, and the actions $\pi(v_1 \cdots v_k) \Downarrow \mathit{tk}$ with $\pi \in \mathrm{Prim}_{\mathrm{plat}}$. The *trace* of an execution is the sequence of its observable events, in order; silent transitions and the actions of sleep, spawn and kill, whose whole effect lies within the configuration, are omitted. The set of traces

$$\mathrm{Tr}(\mathit{program}, j)$$

is the set of the traces of the executions of *program* with flags j, and the *meaning* of *program* is the function that maps each j to $\mathrm{Tr}(\mathit{program}, j)$. When $\mathrm{Tr}(\mathit{program}, j)$ is empty, the program does not start with flags j; otherwise it holds the empty trace and every prefix of each of its traces.

An implementation *conforms* to the definition if, for every program that has a meaning and every boundary value j, every trace that the implementation can produce when it runs the program with flags j is a member of $\mathrm{Tr}(\mathit{program}, j)$, and the implementation does not start the program when $\mathrm{Tr}(\mathit{program}, j)$ is empty. Two conforming implementations may produce different traces of one program, as where the order of the commands of a batch, or the interleaving of processes, is left open, and both conform. Conformance concerns what is produced, not that anything is: the definition does not require a program to make progress.

9.6 Views and Other Programs

Let $B_0 \vdash \mathit{program} \Rightarrow (\tau'\,\texttt{Node}, PE, v)$. The program is a *view*: its meaning is the value v of type `Html`, which the host renders. It has no configuration

and no transitions: it handles no message, and has no traffic on its ports, whatever ports its modules declare. How the host renders v is outside the definition; an implementation that only evaluates the program conforms. Every top-level value of every module is evaluated, so a `Debug.log` in the definition of `main` is evaluated once.

A platform may provide functions other than `Platform.worker` that make program values, such as the functions `sandbox`, `element`, `document` and `application` of the module `Browser`. They are not part of the initial basis, and their records have further fields, such as `view` (Section 8.5). A program whose `main` is made by one of them runs as in Section 9.4, except that the platform defines how `init` is applied (for `application`, to the flags, a URL and a navigation key), and that among its platform steps are steps that render the value of `view` applied to the present model, as a platform event, after each update, and steps that append to the message queue the messages that the host sends from the rendered view. A platform that does not provide these functions, or does not run the programs they make, is recorded in Appendix E.

Appendix A

Derived Forms

Several derived forms are provided in the Core and in Modules; they are presented in Figures 27, 28, 29 and 30. Each derived form is given with its equivalent form, and each row of the tables is a rewriting rule

$$\text{Derived form} \implies \text{Equivalent form}$$

which may be applied to any subphrase, repeatedly, until the phrase is a phrase of the bare language of Chapters 2 and 3. Within a figure the rows are ordered, and the figures follow one another, so that every equivalent form uses only bare syntax and the derived forms of earlier rows. Two rows, the grouping of the definitions of a `let` and the grouping of the declarations of a module, are not local rewritings; their equivalent forms are defined in Sections A.1 and A.2. See Appendix B for the full grammar, including all the derived forms.

In equivalent forms, `Basics.True` and `Basics.False` are the constructors of the type `Bool`, `Basics.negate` is the negation function, and `::` is the list constructor. They are looked up as any other identifier of the module in which the derived form occurs. Because only an import of the module `Basics` provides the qualifier `Basics` (Section 3.7), `Basics.True`, `Basics.False` and `Basics.negate` always denote the bindings of the initial basis (Appendices C and D), whatever the module imports or declares; `::` is bound in the initial environment E_0 of those appendices, from which every module is elaborated and evaluated, and no program declares it. Being qualified, or an operator, none of them is a free occurrence of a variable in the sense below.

In the first row of Figure 27, exp_1 *op* exp_2 is an infixed application as grouped by the resolution of Section 2.7: exp_1 and exp_2 are the two operands of *op* after grouping, and *op* is neither `&&` nor `||`. Each operator of a chain is rewritten in this way, outermost first. The infixed forms of `&&` and `||` have rows of their own, which make them conditionals; the operator values `(&&)` and `(||)` are bare, and denote functions of the initial basis that take both arguments as values. In the row for negation, `-`*atexp* is a negation as defined

in Section 2.5; it is atomic, so that it may be the argument of an application. In the row for field access, the atomic expressions that may precede *.lab* are restricted in Section 2.10 (the restrictions on records); after any other atomic expression, and after a field accessor, a `.` begins a field accessor (Section 2.5), so $.lab_1 .lab_2$ is the application of $.lab_1$ to $.lab_2$. In the equivalent form of the field accessor *.lab*, the record pattern `{` *lab* `=` *lab* `}` binds the variable spelt as the label *lab*, and the body of the lambda is that variable. The pattern row is written in the bare syntax, *lab* `=` *var* with *var* spelt as *lab*, since the pattern row written as a label alone is a derived form of a later figure.

An annotation *var* `:` *ty* and the definition of *var* that follows it (Chapter 2) are written together, here as in the bare grammar, as one value binding with `:` *ty* after *var*. Thus a definition is a value binding without `and`, of the form $var\ \langle : ty \rangle\ atpat_1 \cdots atpat_n$ `=` exp ($n \geq 0$) or *atpat* `=` *exp*.

The syntactic restrictions of Chapters 2 and 3 apply to a program as written, before any derived form in it is replaced by its equivalent form (Section 2.10), and are not checked again on equivalent forms. In particular the variable bound in the equivalent form of *.lab* may have the name of a variable in scope, and a restriction that distinguishes a definition with parameters from one without refers to the definition as written.

An occurrence of a variable in a phrase is *free* if it is unqualified and is not within the scope of a binding of that variable, by a pattern or by a value binding, made inside the phrase. No equivalent form adds or removes a free occurrence of a variable, so the groupings of Sections A.1 and A.2 are the same whichever other derived forms have already been rewritten.

No equivalent form introduces a new identifier, so no table carries a note of fresh identifiers. The only variables bound by equivalent forms are the one bound by the equivalent form of *.lab* and the one bound by the equivalent form of a pattern row written as a label alone; each is spelt as its label.

In the equivalent form for lists, each `::` is an infixed application whose right operand is everything to its right:

$$exp_1 \texttt{ :: (}exp_2 \texttt{ :: } \cdots \texttt{ (}exp_n \texttt{ :: [ ])} \cdots \texttt{).}$$

In the equivalent form for list patterns, each `::` is the bare pattern *pat* `::` *pat*, the parentheses make the result an atomic pattern, and for $n = 1$ the equivalent form is $(pat_1$ `:: [ ])`. In the row for pattern rows, *var* is the variable spelt as *lab*: the pattern `{ h, n }` stands for `{ h = h, n = n }`. A program writes every pattern row in this derived form (Section 2.10); only the copies of number specialisation (Section 6.2) have pattern rows whose variable is spelt otherwise.

No derived form supplies the header of a module: every module is written with one (Chapter 3).

Expressions *exp*

Derived Form	Equivalent Form
exp_1 *op* exp_2	`(`*op*`)` `(`exp_1`)` `(`exp_2`)`
`if` exp_1 `then` exp_2 `else` exp_3	`case` exp_1 `of Basics.True ->` `(`exp_2`)` `; Basics.False ->` exp_3
`if` exp_1 `then` exp'_1 `else if` exp_2 `then` exp'_2 $\cdots$ `else if` exp_n `then` exp'_n `else` *exp*	`if` exp_1 `then` exp'_1 `else (` `if` exp_2 `then` exp'_2 `else (` $\cdots$ `if` exp_n `then` exp'_n `else` *exp* `)` $\cdots$ `)` $(n \geq 2)$
exp_1 `&&` exp_2	`if` exp_1 `then` exp_2 `else Basics.False`
exp_1 `||` exp_2	`if` exp_1 `then Basics.True else` exp_2
`\`$atpat_1 \cdots atpat_n$ `->` *exp*	`\`$atpat_1$ `->` $\cdots$ `\`$atpat_n$ `->` *exp* $(n \geq 2)$

Atomic Expressions *atexp*

Derived Form	Equivalent Form
`-`*atexp*	`(Basics.negate` *atexp*`)`
`[`exp_1`,` $\cdots$`,` exp_n`]`	`(`exp_1 `::` $\cdots$ `::` exp_n `::` `[ ])` $(n \geq 1)$
.lab	`(\{` *lab* `=` *lab* `}` `->` *lab*`)`
atexp .lab	`(.`*lab* *atexp*`)`

Figure 27: Derived forms of Expressions and Atomic Expressions

Atomic Patterns *atpat*

Derived Form	Equivalent Form
`[`pat_1`,` $\cdots$`,` pat_n`]`	`(`pat_1 `::` `(`pat_2 `::` $\cdots$ `(`pat_n `::` `[ ])` $\cdots$`))` $(n \geq 1)$

Pattern Rows *patrow*

Derived Form	Equivalent Form
lab ⟨`,` *patrow*⟩	*lab* `=` *var* ⟨`,` *patrow*⟩ (*var* spelt as *lab*)

Figure 28: Derived forms of Patterns and Pattern Rows

Value Bindings *valbind*

Derived Form	Equivalent Form
var ⟨`:` *ty*⟩ $atpat_1 \cdots atpat_n$ `=` *exp* ⟨`and` *valbind*⟩	*var* ⟨`:` *ty*⟩ `=` `\`$atpat_1 \cdots atpat_n$ `->` *exp* ⟨`and` *valbind*⟩ $(n \geq 1)$

Declarations *dec*

Derived Form	Equivalent Form
`let` $valbind_1$ $\cdots$ $valbind_n$ `in` *exp*	`let` dec_1 `in` $\cdots$ `let` dec_k `in` *exp* (see Section A.1) $(n \geq 2)$

Figure 29: Derived forms of Value Bindings and Declarations

Top-level Declarations *topdec*

Derived Form	Equivalent Form

$topdec_1 \ \cdots \ topdec_n$	`type` *datbind* `;`
	`type alias` $typbind'_1$ `;` $\cdots$ `type alias` $typbind'_l$ `;`
	$topdec'_1$ `;` $\cdots$ `;` $topdec'_k$
	(see Section A.2) $\quad (n \geq 2)$

Figure 30: Derived forms of Top-level Declarations

A.1 Grouping the Definitions of a Let

A `let` expression as written has definitions $valbind_1, \ldots, valbind_n$ $(n \geq 2)$, each a value binding without `and`. The bare language has `let` *dec* `in` *exp* only for a single group of bindings joined by the abstract `and`, so the definitions are grouped and ordered as follows.

The variables *bound by* a definition, and its *right-hand side*, are given by the following table, in which $m \geq 1$ and a definition of the first two forms may carry an annotation `:` *ty* after *var*:

Definition	Bound	Right-hand side
var `=` *exp*	*var*	*exp*
var $atpat_1 \cdots atpat_m$ `=` *exp*	*var*	`\` $atpat_1 \cdots atpat_m$ `->` *exp*
atpat `=` *exp*	those of *atpat*	*exp*

The variables bound by a pattern are its variable patterns, the variables of its pattern rows (Section 2.10), and the variables following `as` in it. A definition $valbind_i$ *refers to* a definition $valbind_j$ if a variable bound by $valbind_j$ occurs free in the right-hand side of $valbind_i$; $j = i$ is allowed. Two definitions are in the same *component* if each can be reached from the other by a chain of zero or more steps of "refers to". The components partition the definitions. A component is *recursive* if it has more than one member, or if its one member refers to itself. The members of the recursive components are exactly the definitions that lie on a cycle of "refers to", and the restriction on recursion in Section 2.10 requires each of them to be a definition with parameters; in particular a definition *atpat* `=` *exp* is never in a recursive component.

Number the components $1, \ldots, k$ so that whenever a member of component a refers to a member of component $b \neq a$, then $b < a$. Such a numbering exists, because a chain of "refers to" that returns to its start never leaves a component. For each a, let $valbind_{i_1}, \ldots, valbind_{i_p}$, with $i_1 < \cdots < i_p$, be the members of component a, and let

$$dec_a \ = \ valbind_{i_1} \ \texttt{and} \ \cdots \ \texttt{and} \ valbind_{i_p}.$$

Then the equivalent form is given by

$$\begin{aligned} &\texttt{let} \ valbind_1 \ \cdots \ valbind_n \ \texttt{in} \ exp \\ &\quad \Longrightarrow \ \texttt{let} \ dec_1 \ \texttt{in} \ \texttt{let} \ dec_2 \ \texttt{in} \ \cdots \ \texttt{let} \ dec_k \ \texttt{in} \ exp. \end{aligned}$$

An annotation is part of the definition it annotates, and so goes with that definition into dec_a.

When two components are independent (neither can be reached from the other) the numbering is not unique, and which admissible numbering is chosen does not matter: independent components bind distinct variables and neither refers to the other, the bindings of a component add no free type variable to the context (Clos generalises every type variable not free in the context, Section 4.8), and evaluation has neither state nor exceptions, so each component elaborates and evaluates to the same result whichever independent components precede it.

A.2 Grouping the Declarations of a Module

The declarations of a module body as written stand in any order. Each declaration $topdec_i$ is one of the following, where $m \geq 0$:

`type` $datbind_i$	a custom type, declaring one type;
`type alias` $typbind_i$	an alias, declaring one alias;
`port` var `:` ty	a port;
var $\langle$`:` $ty\rangle$ $atpat_1 \cdots atpat_m$ `=` exp	a definition.

A module body has no definitions of the form $atpat$ `=` exp (Chapter 3). A module body as written is always rewritten by this row, even when its declarations already stand in an order the row could produce. The equivalent form

$$\begin{array}{l}\texttt{type}\ datbind\ \texttt{;}\\ \texttt{type alias}\ typbind'_1\ \texttt{;}\ \cdots\ \texttt{type alias}\ typbind'_l\ \texttt{;}\\ topdec'_1\ \texttt{;}\ \cdots\ \texttt{;}\ topdec'_k\end{array}$$

is built as follows, where `;` is the abstract separator of top-level declarations in the bare grammar (Chapter 3); a part that is omitted below is omitted with its separator.

1. *Custom types.* If `type` $datbind_{i_1}, \ldots,$ `type` $datbind_{i_p}$, with $i_1 < \cdots < i_p$, are the custom types of the body, then all of them form one group,

$$datbind \;=\; datbind_{i_1}\ \texttt{and}\ \cdots\ \texttt{and}\ datbind_{i_p},$$

   whose members may refer to one another. If $p = 0$, the declaration `type` $datbind$ is omitted.

2. *Aliases.* An alias *mentions* another alias of the body if the type constructor that the other binds occurs, unqualified, in its body. The aliases $typbind'_1, \ldots, typbind'_l$ are those of the body, ordered so that each comes after every alias it mentions. Such an order exists, because the restrictions of Chapter 3 forbid a chain of mentions that returns to its start. If $l = 0$, there are no alias declarations.

3. *Ports and definitions.* A port `port` *var* `:` *ty* binds *var* and has no right-hand side, so it refers to nothing; a definition binds its *var* and has the right-hand side of Section A.1. Over the ports and definitions of the body, "refers to", components and their numbering $1, \ldots, k$ are defined as in Section A.1. A port is alone in its component. For each a, $topdec'_a$ is the port if component a is a port, and otherwise is the value group

$$valbind_{i_1} \texttt{ and } \cdots \texttt{ and } valbind_{i_q},$$

where $valbind_{i_1}, \ldots, valbind_{i_q}$, with $i_1 < \cdots < i_q$, are the members of component a, each with its annotation.

The type constructors bound by *datbind* and by the aliases are in scope throughout these type declarations: a constructor binding in *datbind* may mention an alias of the body, and an alias may mention a custom type of *datbind*. Chapter 5 elaborates the group *datbind* together with the aliases that follow it.

The values that the type declarations bind (the constructors of *datbind* and the constructors of record aliases) are thus in scope in every port and definition. As for a `let`, which order is chosen among aliases neither of which mentions the other, and which admissible numbering is chosen for the components, does not matter: an alias denotes the same type function whichever unrelated aliases precede it, and a component of ports and definitions elaborates and evaluates to the same result whichever independent components precede it, for the reasons given in Section A.1.

Appendix B

Full Grammar

The syntax of programs is given in Chapter 9. The syntax of long identifiers is that of Chapter 2.

The full grammar of the Core and of Modules consists of the grammar of Chapters 2 and 3, in which each derived form of Appendix A is added as an alternative of the phrase class it belongs to. The grammatical rules are shown in Figures 31, 32, 33, 34 and 35. An alternative whose label carries the mark † is a derived form, whose equivalent form is given in Appendix A; every other alternative is a phrase of the bare language, and has the same label as in Chapter 2 or 3. No subclass of Exp is introduced: which expressions may be the operands of an operator chain is stated in Section 2.7.

The grammatical conventions are those of Chapter 2, namely:

- The brackets $\langle\ \rangle$ enclose optional phrases.
- Alternative forms for each phrase class are in order of decreasing precedence. L (resp. R) means left (resp. right) association.
- The infixed application, conjunction and disjunction alternatives are grouped by the fixities of Figure 3, as Section 2.7 describes, and not by their order in Figure 31.
- Which item `-` is a negation symbol, and which `.` begins a field selection or a field accessor, is decided by lexical analysis (Section 2.5).
- The precedence of types and of patterns, and the extent of lambda expressions, case expressions and `let` expressions, are as stated with the grammar of the Core (Figure 5 and the conventions before it); the extent of each phrase is decided by the layout rule (Sections 2.6 and 3.4).
- The separator `;` between match rules and between top-level declarations, and the connective `and` between the members of a value binding or of a custom-type binding, are abstract syntax. Neither is an item of

source text: match rules, the definitions of a `let` and the declarations of a module are separated by layout, and `and` is not a reserved word. Likewise an annotated binding *var* `:` *ty* `=` *exp* is written in source text as the annotation *var* `:` *ty* followed by a definition of *var*. The derived alternatives `let` $valbind_1 \cdots valbind_n$ `in` *exp* and $topdec_1 \cdots topdec_n$ are the forms in which several definitions and declarations are written; in them each $valbind_i$ is a definition, that is, a value binding without `and`, and each $topdec_i$ is a single declaration, as listed in Section A.2.

The syntactic restrictions are those of Sections 2.10 and 3.7. They apply to phrases of the full grammar, that is, to a program as written, before any derived form is replaced by its equivalent form.

In the infixed application alternative, *op* is neither `&&` nor `||`. The alternative `-` *atexp* is atomic, so that a negated expression may be the argument of an application.

Type expressions have no derived forms; Figure 33 is Figure 7 unchanged. Modules, headers, imports and exposing lists have no derived forms either: every module has a header, and Figure 34 is Figure 9 unchanged.

A top-level value declaration as written is a variable binding or a function binding without `and`; the function binding alternative of Figure 31 applies to it.

atexp	::=	*scon*	special constant
		longvar	value variable
		longcon	value constructor
		`(` *op* `)`	operator value
		`()`	unit
		`(` exp_1 `,` exp_2 $\langle$`,` $exp_3\rangle$ `)`	tuple
		`[ ]`	empty list
		`[`exp_1`, ` $\cdots$ `, `exp_n`]`	list ($n \geq 1$)†
		`{` $\langle$*exprow*$\rangle$ `}`	record
		`{` *var* `|` *exprow* `}`	record update
		`.`*lab*	field accessor†
		atexp`.`*lab*	field selection†
		`-`*atexp*	negation†
		`let` *dec* `in` *exp*	local declaration
		`let` $valbind_1 \cdots valbind_n$	definitions†
		`in` *exp*	($n \geq 2$)
		`(` *exp* `)`	parentheses
exprow	::=	*lab* `=` *exp* $\langle$`,` *exprow*$\rangle$	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *op* exp_2	infixed application†
		exp_1 `&&` exp_2	conjunction†
		exp_1 `||` exp_2	disjunction†
		`\` *atpat* `->` *exp*	function
		`\`$atpat_1 \cdots atpat_n$ `->` *exp*	function ($n \geq 2$)†
		`if` exp_1 `then` exp_2 `else` exp_3	conditional†
		`if` exp_1 `then` exp'_1	conditional chain†
		`else if` exp_2 `then` exp'_2 $\cdots$	($n \geq 2$)
		`else if` exp_n `then` exp'_n	
		`else` *exp*	
		`case` *exp* `of` *match*	case analysis
match	::=	*mrule* $\langle$`;` *match*$\rangle$	match
mrule	::=	*pat* `->` *exp*	match rule
dec	::=	*valbind*	value declaration
valbind	::=	*var* $\langle$`:` *ty*$\rangle$ `=` *exp* $\langle$`and` *valbind*$\rangle$	variable binding
		var $\langle$`:` *ty*$\rangle$ $atpat_1 \cdots atpat_n$	function binding†
		`=` *exp* $\langle$`and` *valbind*$\rangle$	($n \geq 1$)
		atpat `=` *exp*	destructuring binding

† derived form, Appendix A.

Figure 31: Full Grammar: Expressions, Matches, Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		var	variable
		longcon	constructor
		`()`	unit
		`(` pat_1 `,` pat_2 $\langle$`,` $pat_3\rangle$ `)`	tuple
		`[ ]`	empty list
		`[`pat_1`, ` $\cdots$`, ` pat_n`]`	list ($n \geq 1$)†
		`{` $\langle patrow\rangle$ `}`	record
		`(` *pat* `)`	parentheses
patrow	::=	*lab* `=` *var* $\langle$`,` *patrow*$\rangle$	pattern row
		lab $\langle$`,` *patrow*$\rangle$	label as variable†
pat	::=	*atpat*	atomic
		longcon $atpat_1$ $\cdots$ $atpat_n$	constructed pattern ($n \geq 1$)
		pat_1 `::` pat_2	list construction (R)
		pat `as` *var*	layered

† derived form, Appendix A.

Figure 32: Full Grammar: Patterns

ty	::=	*tyvar*	type variable
		`{` $\langle tyrow\rangle$ `}`	record type expression
		`{` *tyvar* `|` *tyrow* `}`	extensible record type expression
		`()`	unit type expression
		`(` ty_1 `,` ty_2 $\langle$`,` $ty_3\rangle$ `)`	tuple type expression
		longtycon ty_1 $\cdots$ ty_n	type construction ($n \geq 0$)
		ty `->` ty'	function type expression (R)
		`(` *ty* `)`	parentheses
tyrow	::=	*lab* `:` *ty* $\langle$`,` *tyrow*$\rangle$	type-expression row

Figure 33: Full Grammar: Type Expressions

module	::=	*header imports topdec*	module
header	::=	`module` *modid* `exposing` *explist*	plain
		`port module` *modid* `exposing` *explist*	port module
imports	::=	`import` *modid* ⟨`as` *modid′*⟩ ⟨`exposing` *explist*⟩ *imports*	import
			empty
explist	::=	`(..)`	everything
		`(` $item_1$ `,` $\cdots$ `,` $item_n$ `)`	explicit ($n \geq 1$)
item	::=	*var*	value
		`(`*op*`)`	operator
		tycon	type
		tycon `(..)`	open type

Figure 34: Full Grammar: Modules, Headers, Imports and Exposing Lists

topdec	::=	*valbind*	value
		`type` *datbind*	custom type
		`type alias` *typbind*	type alias
		`port` *var* `:` *ty*	port
		$topdec_1$ `;` $topdec_2$	sequential
		$topdec_1 \cdots topdec_n$	module body ($n \geq 2$)†
datbind	::=	*tycon tyvarseq* `=` *conbind* ⟨`and` *datbind*⟩	custom-type binding
conbind	::=	*con* $ty_1 \cdots ty_n$ ⟨`|` *conbind*⟩	constructor binding ($n \geq 0$)
typbind	::=	*tycon tyvarseq* `=` *ty*	type-alias binding
tyvarseq	::=	$tyvar_1 \cdots tyvar_k$	type parameters ($k \geq 0$)

† derived form, Appendix A.

Figure 35: Full Grammar: Top-level Declarations and Bindings

Appendix C

The Initial Static Basis

In this appendix and the next we define a minimal initial basis: the type names, modules, types and values that the definition itself mentions, together with all of the module `Basics`. Libraries may provide richer bases, extending the module environment with further modules and the interfaces below with further values; such libraries are not described here. Components of the initial basis are indicated by the subscript 0. The initial static basis is

$$B_0 = T_0, ME_0, PE_0, \qquad \text{where } PE_0 = \{\},$$

T_0 is given in Section C.1 and ME_0 in Sections C.2 to C.5. The basis declares no port.

In the type schemes of this appendix a type variable is written as the type variable of the syntax that names it (Section 4.1), so that its spelling gives its class: *number* has class number, *comparable* has class comparable, *appendable* has class appendable, and every other variable written here (a, b, c, d, e, r, x, y, *msg*, *flags*, *model*) has class any. No variable of class compappend occurs in the initial basis. A type is written as in Chapter 4, with type names after their arguments: the type that a program writes `Maybe a` is a `Maybe`, and `Task x a` is (x, a)`Task`.

C.1 Type Names

The set of type names of the initial basis is

$$\begin{aligned} T_0 = \{&\texttt{Int}, \texttt{Float}, \texttt{Char}, \texttt{String}, (), \texttt{List}, \texttt{Tuple}_2, \texttt{Tuple}_3, \\ &\texttt{Bool}, \texttt{Order}, \texttt{Never}, \texttt{Maybe}, \texttt{Result}, \texttt{Array}, \texttt{Bytes}, \texttt{Value}, \\ &\texttt{Cmd}, \texttt{Sub}, \texttt{Task}, \texttt{ProcessId}, \texttt{Program}, \texttt{Node}\}. \end{aligned}$$

The members of T_0 are type names, not type constructors; we have spelt each as a type constructor bound to it in ME_0, or, for () and $\texttt{Tuple}_k$, after the syntax of the types that denote them. Their arities, and the type constructors

bound to them, are shown in Figure 36. The first eight are the type names of Figure 13, which the rules of Chapter 4 mention directly. The type names (), $\texttt{Tuple}_2$ and $\texttt{Tuple}_3$ are bound to no type constructor: the type expressions `()` and `(` ty_1`,` $ty_2\langle$`,` $ty_3\rangle$`)` denote them. Which of the type names are admissible for each class is given by Figure 15: a type built from any type name of T_0 other than `Int`, `Float`, `Char`, `String`, `List`, $\texttt{Tuple}_2$ and $\texttt{Tuple}_3$ is admissible for no class but any.

t	arity	bound to
`Int`	0	`Basics.Int`
`Float`	0	`Basics.Float`
`Char`	0	`Char.Char`
`String`	0	`String.String`
()	0	(none)
`List`	1	`List` (Section C.2)
$\texttt{Tuple}_2$	2	(none)
$\texttt{Tuple}_3$	3	(none)
`Bool`	0	`Basics.Bool`
`Order`	0	`Basics.Order`
`Never`	0	`Basics.Never`
`Maybe`	1	`Maybe.Maybe`
`Result`	2	`Result.Result`
`Array`	1	`Array.Array`
`Bytes`	0	`Bytes.Bytes`
`Value`	0	`Json.Encode.Value`, `Json.Decode.Value`
`Cmd`	1	`Platform.Cmd.Cmd`
`Sub`	1	`Platform.Sub.Sub`
`Task`	2	`Platform.Task`, `Task.Task`
`ProcessId`	0	`Platform.ProcessId`, `Process.Id`
`Program`	3	`Platform.Program`
`Node`	1	`VirtualDom.Node`, `Html.Html`

Figure 36: Type Names of the Initial Basis, T_0

The type names `Bool`, `Order`, `Maybe` and `Result` are those of custom types whose constructors ME_0 exposes; `Never` is that of a custom type that `Basics` exposes closed (Section 5.1): its one constructor `JustOneMore` is in its type structure but not in the value environment of the interface, so that no phrase of a program has a value of type `Never`. The others are *abstract*: ME_0 binds them with no constructors, and their values are given by the dynamic basis (Appendix D) and Chapter 8.

C.2 The Module Environment

The module environment ME_0 has the domain

$$\{\texttt{Basics}, \texttt{List}, \texttt{Maybe}, \texttt{Result}, \texttt{String}, \texttt{Char}, \texttt{Tuple}, \texttt{Array}, \texttt{Bytes}, \texttt{Debug}, \texttt{Platform}, \texttt{Platform.Cmd}, \texttt{Platform.Sub}, \texttt{Task}, \texttt{Process}, \texttt{Json.Encode}, \texttt{Json.Decode}, \texttt{VirtualDom}, \texttt{Html}\}.$$

The interface $ME_0(\mathit{modid})$ of each module is an environment $(\{\}, TE, VE)$ whose qualified environment is empty; its type environment and value environment are shown in Figures 37 to 44. In Figures 41 to 44 the entries of each interface are headed by the name of its module; the bindings of the type environment come first and, below a rule, those of the value environment. An interface with no binding shown in a component has the empty map there; the interfaces of `List` and `Tuple` are $(\{\}, \{\}, \{\})$. As in Chapter 4, every variable has status v and every constructor status c, and a constructor of a custom type is bound in the value environment of the type's structure and, unless the type is closed, in the value environment of the interface. The type names occurring in ME_0 are those of T_0:

$$\operatorname{tynames} ME_0 \subseteq T_0.$$

Every module of ME_0 belongs to the library `elm/core`, except five: `Bytes` belongs to `elm/bytes`, `Json.Encode` and `Json.Decode` to `elm/json`, `VirtualDom` to `elm/virtual-dom`, and `Html` to `elm/html`. These five are in ME_0 only for their types, which the definition of ports (Section 5.4) and of the type of `main` (Chapter 9) mention.

Before any import is read, every module has in scope, unqualified, the type constructor `List` and the two list constructor names `[]` and `::` of Section 4.2, which no identifier spells. They form the environment

$$E_0 = (\{\}, TE_0, VE_0), \qquad TE_0 = \{\texttt{List} \mapsto (\texttt{List}, VE_0)\},$$

where

$$VE_0 = \{\,\texttt{[]} \mapsto (\forall a.\, a\,\texttt{List}, \mathrm{c}),\ \texttt{::} \mapsto (\forall a.\, a \to a\,\texttt{List} \to a\,\texttt{List}, \mathrm{c})\}.$$

The module `List` declares neither the type constructor `List` nor any constructor, so that `List.List` is not a long type constructor of the initial basis.

The imports of every module begin with the default imports of Figure 11, which are elaborated (Chapter 5) against ME_0. Every module they name is in $\operatorname{Dom} ME_0$, and every type and value they expose is in that module's interface; an operator they expose is one that the module declares, and binds nothing. Thus the default imports put in scope, unqualified, every type, constructor and value of `Basics`; the types `Maybe`, `Result`, `String`, `Char`,

`Program`, `Cmd` and `Sub`; and the constructors of `Maybe` and `Result`. They make every value of `Debug`, `Platform`, `Maybe` and `Result` available qualified, and those of `Platform.Cmd` and `Platform.Sub` under the qualifiers `Cmd` and `Sub`. The modules `Array`, `Bytes`, `Task`, `Process`, `Json.Encode`, `Json.Decode`, `VirtualDom` and `Html` are in scope only where a module imports them.

Operators. No program declares an operator, so the operators are those of the initial basis, the same in every program; their fixities are those of Figure 3. Each is declared, by `Basics` or `List`, to stand for an identifier. The *static operator table*

$$\mathrm{OT}_{\mathrm{STAT}} : \mathrm{Op} \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme} \times \mathrm{IdStatus}$$

maps each operator of Figure 3 to the type scheme and status of the identifier it stands for, and is given in Figure 40. The meaning of an operator is $\mathrm{OT}_{\mathrm{STAT}}(op)$, whatever the context (Section 4.3): it is looked up in no environment and reached through no interface, so a program's own top-level `add` does not capture `+`. The operators of library modules outside the initial basis are outside the definition. The operator `::` stands for the list constructor `::` of VE_0, with status c, and is declared by `List`; every other operator is declared by `Basics`, as the figure records, so that the set of operators that `List` declares is $\{\texttt{::}\}$ and that of `Basics` is $\mathrm{Dom}\,\mathrm{OT}_{\mathrm{STAT}} \setminus \{\texttt{::}\}$ (no other module of ME_0 declares an operator). Every operator other than `::` stands for a variable of `Basics` (`add`, `eq`, `apR` and the others of the figure) that `Basics` declares but does not expose. These variables are *hidden*: they are bound in no interface of ME_0, so that no program can write `Basics.add` or `add` for `(+)`.

Hidden identifiers. Besides the variables that the operators stand for, the modules of ME_0 declare the following identifiers, which no interface binds and no program can name: the constructor `JustOneMore` of the closed type `Never` of `Basics`; and, in `Task`, the function `command` and the constructor `Perform` of the custom type `MyCmd`, which `Task` does not expose. In elm/core 1.0.5 that type is declared as `type MyCmd msg = Perform (Task Never msg)`, so the task that `Perform` carries cannot fail. These hidden identifiers are mentioned only by the texts of values of the basis written in Elm, and their values are given in Section D.3.

Library functions. The modules of ME_0 declare, in their libraries, many values that are absent from the figures, among them every function of `List`, `String`, `Tuple`, `Array`, `Maybe` and `Result`, every function of `Char` but `toCode` and `fromCode`, the functions of `Bytes`, and the encoders, decoders and views of `elm/json` and `elm/html`. Like the modules not in $\mathrm{Dom}\,ME_0$ (`Dict`, `Set`, `Bitwise`, `Bytes.Encode`, `Bytes.Decode` and the rest), they are library, not language, and lie outside the definition. The module `Tuple` is in ME_0 only because a default import names it; its interface is empty. The type `Router` and the values `sendToApp` and `sendToSelf` of `Platform`, which serve managers, are omitted with them, effect modules being kernel-only.

C.3 Basics

The interface $ME_0(\texttt{Basics})$ contains every type and value that `Basics` exposes. Its type environment is shown in Figure 37, and its value environment in Figures 38 and 39. The equality operators have a type scheme whose variable has class any, so that equality elaborates at every type.

$tycon \mapsto (\theta,\ \{con_1 \mapsto (\sigma_1, is_1), \cdots, con_n \mapsto (\sigma_n, is_n)\}) \quad (n \geq 0)$
$\texttt{Int} \mapsto (\texttt{Int},\ \{\})$
$\texttt{Float} \mapsto (\texttt{Float},\ \{\})$
$\texttt{Bool} \mapsto (\texttt{Bool},\ \{\texttt{True} \mapsto (\texttt{Bool}, \mathrm{c}),\ \texttt{False} \mapsto (\texttt{Bool}, \mathrm{c})\})$
$\texttt{Order} \mapsto (\texttt{Order},\ \{\texttt{LT} \mapsto (\texttt{Order}, \mathrm{c}),\ \texttt{EQ} \mapsto (\texttt{Order}, \mathrm{c}),$ $\texttt{GT} \mapsto (\texttt{Order}, \mathrm{c})\})$
$\texttt{Never} \mapsto (\texttt{Never},\ \{\texttt{JustOneMore} \mapsto (\texttt{Never} \to \texttt{Never}, \mathrm{c})\})$

Figure 37: Static TE of $ME_0(\texttt{Basics})$

var or $con \mapsto (\sigma, is)$
$\texttt{True}, \texttt{False} \mapsto (\texttt{Bool}, \mathrm{c})$
$\texttt{LT}, \texttt{EQ}, \texttt{GT} \mapsto (\texttt{Order}, \mathrm{c})$
$\texttt{toFloat} \mapsto (\texttt{Int} \to \texttt{Float}, \mathrm{v})$
$\texttt{round}, \texttt{floor} \mapsto (\texttt{Float} \to \texttt{Int}, \mathrm{v})$
$\texttt{ceiling}, \texttt{truncate} \mapsto (\texttt{Float} \to \texttt{Int}, \mathrm{v})$
$\texttt{max}, \texttt{min} \mapsto (\forall comparable.\ comparable \to comparable \to comparable, \mathrm{v})$
$\texttt{compare} \mapsto (\forall comparable.\ comparable \to comparable \to \texttt{Order}, \mathrm{v})$
$\texttt{not} \mapsto (\texttt{Bool} \to \texttt{Bool}, \mathrm{v})$
$\texttt{xor} \mapsto (\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \mathrm{v})$
$\texttt{modBy}, \texttt{remainderBy} \mapsto (\texttt{Int} \to \texttt{Int} \to \texttt{Int}, \mathrm{v})$
$\texttt{negate}, \texttt{abs} \mapsto (\forall number.\ number \to number, \mathrm{v})$
$\texttt{clamp} \mapsto (\forall number.\ number \to number \to number \to number, \mathrm{v})$

Figure 38: Static VE of $ME_0(\texttt{Basics})$: Constructors, Conversions, Comparison, Booleans and Integer Functions

The operators of Figure 40 are listed in decreasing order of precedence. The infixed forms of `&&` and `||` are derived forms (Appendix A), which elaborate as conditional expressions; the type schemes of `and` and `or` are those of the operator values `(&&)` and `(||)`.

$var \mapsto (\sigma, is)$
$\texttt{sqrt} \mapsto (\texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{logBase} \mapsto (\texttt{Float} \to \texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{e}, \texttt{pi} \mapsto (\texttt{Float}, \text{v})$
$\texttt{degrees}, \texttt{radians} \mapsto (\texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{turns} \mapsto (\texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{cos}, \texttt{sin}, \texttt{tan} \mapsto (\texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{acos}, \texttt{asin}, \texttt{atan} \mapsto (\texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{atan2} \mapsto (\texttt{Float} \to \texttt{Float} \to \texttt{Float}, \text{v})$
$\texttt{toPolar} \mapsto ((\texttt{Float}, \texttt{Float})\texttt{Tuple}_2 \to (\texttt{Float}, \texttt{Float})\texttt{Tuple}_2, \text{v})$
$\texttt{fromPolar} \mapsto ((\texttt{Float}, \texttt{Float})\texttt{Tuple}_2 \to (\texttt{Float}, \texttt{Float})\texttt{Tuple}_2, \text{v})$
$\texttt{isNaN}, \texttt{isInfinite} \mapsto (\texttt{Float} \to \texttt{Bool}, \text{v})$
$\texttt{identity} \mapsto (\forall a.\ a \to a, \text{v})$
$\texttt{always} \mapsto (\forall a\ b.\ a \to b \to a, \text{v})$
$\texttt{never} \mapsto (\forall a.\ \texttt{Never} \to a, \text{v})$

Figure 39: Static VE of $ME_0(\texttt{Basics})$: Floating-point Functions and Function Helpers

op	stands for $\mapsto \text{OT}_{\text{STAT}}(op)$
`<<`	$\texttt{composeL} \mapsto (\forall a\ b\ c.\ (b \to c) \to (a \to b) \to a \to c, \text{v})$
`>>`	$\texttt{composeR} \mapsto (\forall a\ b\ c.\ (a \to b) \to (b \to c) \to a \to c, \text{v})$
`^`	$\texttt{pow} \mapsto (\forall number.\ number \to number \to number, \text{v})$
`*`	$\texttt{mul} \mapsto (\forall number.\ number \to number \to number, \text{v})$
`/`	$\texttt{fdiv} \mapsto (\texttt{Float} \to \texttt{Float} \to \texttt{Float}, \text{v})$
`//`	$\texttt{idiv} \mapsto (\texttt{Int} \to \texttt{Int} \to \texttt{Int}, \text{v})$
`+`, `-`	$\texttt{add}, \texttt{sub} \mapsto (\forall number.\ number \to number \to number, \text{v})$
`++`	$\texttt{append} \mapsto (\forall appendable.\ appendable \to appendable \to appendable, \text{v})$
`::`*	$\texttt{::} \mapsto (\forall a.\ a \to a\,\texttt{List} \to a\,\texttt{List}, \text{c})$
`==`, `/=`	$\texttt{eq}, \texttt{neq} \mapsto (\forall a.\ a \to a \to \texttt{Bool}, \text{v})$
`<`, `>`	$\texttt{lt}, \texttt{gt} \mapsto (\forall comparable.\ comparable \to comparable \to \texttt{Bool}, \text{v})$
`<=`, `>=`	$\texttt{le}, \texttt{ge} \mapsto (\forall comparable.\ comparable \to comparable \to \texttt{Bool}, \text{v})$
`&&`, `||`	$\texttt{and}, \texttt{or} \mapsto (\texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}, \text{v})$
`|>`	$\texttt{apR} \mapsto (\forall a\ b.\ a \to (a \to b) \to b, \text{v})$
`<|`	$\texttt{apL} \mapsto (\forall a\ b.\ (a \to b) \to a \to b, \text{v})$

* Declared by `List` to stand for the list constructor `::` of VE_0. Every other operator is declared by `Basics` to stand for the hidden variable shown.

Figure 40: The Static Operator Table, OT_{STAT}

C.4 Further Data Types

The interfaces of `Maybe`, `Result`, `String`, `Char`, `Array` and `Bytes` are shown in Figure 41. The interfaces of `List` and `Tuple` bind no type constructor and no value; the list type and the list constructors are those of E_0 (Section C.2), and the meaning of the operator `::` that `List` declares is $\mathrm{OT}_{\mathrm{STAT}}(\texttt{::})$ (Figure 40).

`Maybe`
`Maybe` $\mapsto$ (`Maybe`, {`Just` $\mapsto$ $(\forall a.\ a \to a\,\texttt{Maybe}, \mathrm{c})$, `Nothing` $\mapsto$ $(\forall a.\ a\,\texttt{Maybe}, \mathrm{c})$})
`Just` $\mapsto$ $(\forall a.\ a \to a\,\texttt{Maybe}, \mathrm{c})$ `Nothing` $\mapsto$ $(\forall a.\ a\,\texttt{Maybe}, \mathrm{c})$
`Result`
`Result` $\mapsto$ (`Result`, {`Ok` $\mapsto$ $(\forall x\ a.\ a \to (x, a)\texttt{Result}, \mathrm{c})$, `Err` $\mapsto$ $(\forall x\ a.\ x \to (x, a)\texttt{Result}, \mathrm{c})$})
`Ok` $\mapsto$ $(\forall x\ a.\ a \to (x, a)\texttt{Result}, \mathrm{c})$ `Err` $\mapsto$ $(\forall x\ a.\ x \to (x, a)\texttt{Result}, \mathrm{c})$
`String`
`String` $\mapsto$ (`String`, {})
`Char`
`Char` $\mapsto$ (`Char`, {})
`toCode` $\mapsto$ (`Char` $\to$ `Int`, v) `fromCode` $\mapsto$ (`Int` $\to$ `Char`, v)
`Array`
`Array` $\mapsto$ (`Array`, {})
`Bytes`
`Bytes` $\mapsto$ (`Bytes`, {})

Figure 41: Static Interfaces of `Maybe`, `Result`, `String`, `Char`, `Array` and `Bytes`

C.5 Debugging, Programs and Effects

The interfaces of the remaining modules are shown in Figures 42, 43 and 44. The meanings of the values of `Debug` are given in Appendix D. Since the result type of `Debug.todo` is a bound variable of its type scheme, an application of `Debug.todo` elaborates to every type. Chapter 8 builds task, command and subscription values from the values of `Platform.Cmd`, `Platform.Sub`, `Task` and `Process`, and `Platform.worker` makes a program (Chapter 9). The function `command` of `Task` and the constructor `Perform` of its custom type `MyCmd` are hidden (Section C.2): they are absent from the interface of `Task`, and are reached only through the texts of `perform` and `attempt`, whose environment (Section D.3) binds them. The values of ports are not in ME_0: a port is declared by a module of a program (Section 5.4). The type

constructors `Task.Task` and `Process.Id` are type aliases in their libraries; the type functions they are bound to are the type names `Task` and `ProcessId` themselves (Section 4.5), and likewise `Json.Decode.Value` is bound to `Value` and `Html.Html` to `Node`.

`Debug`
`log` $\mapsto (\forall a.\, \texttt{String} \to a \to a, \mathrm{v})$ `todo` $\mapsto (\forall a.\, \texttt{String} \to a, \mathrm{v})$ `toString` $\mapsto (\forall a.\, a \to \texttt{String}, \mathrm{v})$
`Platform`
`Program` $\mapsto (\texttt{Program},\ \{\})$ `Task` $\mapsto (\texttt{Task},\ \{\})$ `ProcessId` $\mapsto (\texttt{ProcessId},\ \{\})$
`worker` $\mapsto (\forall \mathit{flags}\ \mathit{model}\ \mathit{msg}.$ $\{\texttt{init} \mapsto \mathit{flags} \to (\mathit{model}, \mathit{msg}\ \texttt{Cmd})\texttt{Tuple}_2,$ $\texttt{update} \mapsto \mathit{msg} \to \mathit{model} \to (\mathit{model}, \mathit{msg}\ \texttt{Cmd})\texttt{Tuple}_2,$ $\texttt{subscriptions} \mapsto \mathit{model} \to \mathit{msg}\ \texttt{Sub}\}$ $\to (\mathit{flags}, \mathit{model}, \mathit{msg})\texttt{Program},\ \mathrm{v})$
`Platform.Cmd`
`Cmd` $\mapsto (\texttt{Cmd},\ \{\})$
`none` $\mapsto (\forall \mathit{msg}.\, \mathit{msg}\ \texttt{Cmd}, \mathrm{v})$ `batch` $\mapsto (\forall \mathit{msg}.\, (\mathit{msg}\ \texttt{Cmd})\texttt{List} \to \mathit{msg}\ \texttt{Cmd}, \mathrm{v})$ `map` $\mapsto (\forall a\ \mathit{msg}.\, (a \to \mathit{msg}) \to a\ \texttt{Cmd} \to \mathit{msg}\ \texttt{Cmd}, \mathrm{v})$
`Platform.Sub`
`Sub` $\mapsto (\texttt{Sub},\ \{\})$
`none` $\mapsto (\forall \mathit{msg}.\, \mathit{msg}\ \texttt{Sub}, \mathrm{v})$ `batch` $\mapsto (\forall \mathit{msg}.\, (\mathit{msg}\ \texttt{Sub})\texttt{List} \to \mathit{msg}\ \texttt{Sub}, \mathrm{v})$ `map` $\mapsto (\forall a\ \mathit{msg}.\, (a \to \mathit{msg}) \to a\ \texttt{Sub} \to \mathit{msg}\ \texttt{Sub}, \mathrm{v})$

Figure 42: Static Interfaces of `Debug` and the `Platform` Modules

Task
Task ↦ (Task, {})
succeed ↦ $(\forall x\ a.\ a \to (x, a)\texttt{Task}, \text{v})$
fail ↦ $(\forall x\ a.\ x \to (x, a)\texttt{Task}, \text{v})$
map ↦ $(\forall x\ a\ b.\ (a \to b) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task}, \text{v})$
map2 ↦ $(\forall x\ a\ b\ r.\ (a \to b \to r) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task} \to (x, r)\texttt{Task}, \text{v})$
map3 ↦ $(\forall x\ a\ b\ c\ r.\ (a \to b \to c \to r) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task} \to (x, c)\texttt{Task} \to (x, r)\texttt{Task}, \text{v})$
map4 ↦ $(\forall x\ a\ b\ c\ d\ r.\ (a \to b \to c \to d \to r) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task} \to (x, c)\texttt{Task} \to (x, d)\texttt{Task} \to (x, r)\texttt{Task}, \text{v})$
map5 ↦ $(\forall x\ a\ b\ c\ d\ e\ r.\ (a \to b \to c \to d \to e \to r) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task} \to (x, c)\texttt{Task} \to (x, d)\texttt{Task} \to (x, e)\texttt{Task} \to (x, r)\texttt{Task}, \text{v})$
sequence ↦ $(\forall x\ a.\ ((x, a)\texttt{Task})\texttt{List} \to (x, a\,\texttt{List})\texttt{Task}, \text{v})$
andThen ↦ $(\forall x\ a\ b.\ (a \to (x, b)\texttt{Task}) \to (x, a)\texttt{Task} \to (x, b)\texttt{Task}, \text{v})$
onError ↦ $(\forall x\ y\ a.\ (x \to (y, a)\texttt{Task}) \to (x, a)\texttt{Task} \to (y, a)\texttt{Task}, \text{v})$
mapError ↦ $(\forall x\ y\ a.\ (x \to y) \to (x, a)\texttt{Task} \to (y, a)\texttt{Task}, \text{v})$
perform ↦ $(\forall a\ msg.\ (a \to msg) \to (\texttt{Never}, a)\texttt{Task} \to msg\,\texttt{Cmd}, \text{v})$
attempt ↦ $(\forall x\ a\ msg.\ ((x, a)\texttt{Result} \to msg) \to (x, a)\texttt{Task} \to msg\,\texttt{Cmd}, \text{v})$

Figure 43: Static Interface of `Task`

Process
Id ↦ (ProcessId, {})
spawn ↦ $(\forall x\ y\ a.\ (x, a)\texttt{Task} \to (y, \texttt{ProcessId})\texttt{Task}, \text{v})$ sleep ↦ $(\forall x.\ \texttt{Float} \to (x, ())\texttt{Task}, \text{v})$ kill ↦ $(\forall x.\ \texttt{ProcessId} \to (x, ())\texttt{Task}, \text{v})$
Json.Encode and Json.Decode
Value ↦ (Value, {})
VirtualDom
Node ↦ (Node, {})
Html
Html ↦ (Node, {})

Figure 44: Static Interfaces of `Process`, `Json` and View Modules

Appendix D

The Initial Dynamic Basis

We indicate components of the initial basis by the subscript 0, as in Appendix C; in this appendix ME_0, E_0 and VE_0 denote dynamic objects. The initial dynamic basis is the module environment

$$B_0 = ME_0,$$

whose domain is the domain of the static ME_0 (Section C.2). For each *modid* in that domain, $ME_0(modid) = (RA, CE, VE)$ is a dynamic interface (Section 7.2), given in Section D.2. Every module is evaluated starting from the environment

$$E_0 = (\{\}, VE_0), \qquad VE_0 = \{\texttt{[]} \mapsto \texttt{[]},\ \texttt{::} \mapsto \texttt{::}\},$$

to which its imports add (Chapter 7), as statically it starts from the environment E_0 of Section C.2. E_0 is an environment of Chapter 6, with an empty qualified environment; it binds the two list constructor names to the constructor values they name (Section 6.3). Every identifier of the initial static basis is bound in the initial dynamic basis, and the reverse; the hidden identifiers of Section C.2 are bound only in the environments of the texts of Section D.3.

The values of the basis are constructor values, special values, closures given by the text of the library `elm/core` 1.0.5, and basic values, whose meaning is the function APPLY of Section D.4. The basic mathematical operations are defined there in full, on the special values defined in Section D.1; every one of them has a result for every argument, a defined value being given even where no mathematically correct one exists. Three things are left to the implementation and recorded in Appendix E: the range of Int with its normalisation, the accuracy of the elementary functions on floats, and the string that `Debug.toString` gives.

D.1 Special Values

The class of special values (Section 6.3) is the disjoint union

$$\mathrm{SVal} = \mathrm{IntRange} \cup \mathrm{Flt} \cup \mathrm{Chr} \cup \mathrm{Str}$$

of the *Int values*, the *floats*, the *characters* and the *strings*, defined below. The union is disjoint: the Int value 3 and the float 3 are different special values. We write $\mathbb{Z}$ for the set of integers and $\mathbb{R}$ for the set of real numbers.

D.1.1 Int Values

The set $\mathrm{IntRange} \subseteq \mathbb{Z}$ of Int values and the *normalisation* $\mathrm{norm} : \mathbb{Z} \to \mathrm{IntRange}$ are parameters of the definition. Every implementation chooses them so that:

- IntRange has a least member $\min \mathrm{IntRange}$ and a greatest member $\max \mathrm{IntRange}$, and contains every integer between them;
- $\min \mathrm{IntRange} \leq -2^{31}$ and $2^{31} - 1 \leq \max \mathrm{IntRange}$;
- if $n \in \mathrm{IntRange}$ and $n \neq \min \mathrm{IntRange}$, then $-n \in \mathrm{IntRange}$;
- $\mathrm{norm}(n) = n$ for every $n \in \mathrm{IntRange}$.

The choices of the implementations are given in Section E.1. Int values are integers, so Int has one zero.

D.1.2 Floats

The *finite non-zero floats* are the real numbers $m \cdot 2^q$ and $-m \cdot 2^q$, where m and q are integers with either

$$2^{52} \leq m < 2^{53} \text{ and } -1074 \leq q \leq 971, \quad \text{or} \quad 1 \leq m < 2^{52} \text{ and } q = -1074.$$

Each has exactly one such representation, and m is its *significand*. They are the finite non-zero numbers of the binary64 format of the standard IEEE 754-2019. The set of floats is

$$\mathrm{Flt} = \{\text{finite non-zero floats}\} \cup \{+0, -0, +\infty, -\infty, \mathrm{NaN}\},$$

and $\mathrm{Flt}_{\mathrm{fin}} = \mathrm{Flt} \setminus \{+\infty, -\infty, \mathrm{NaN}\}$ is the set of *finite* floats. The two zeros are different floats, and so are the two infinities. There is one NaN. In this appendix x, y and z range over Flt.

In a formula, a float other than NaN stands for the number it denotes on the extended real line, both zeros denoting 0; thus $x < y$ and $x = y$ compare numbers, and $+0 = -0$ holds as a comparison of numbers. That x *is* a particular float, such as -0, is said in words. For $x \neq \mathrm{NaN}$, the *sign* $\mathrm{sgn}(x)$

is $+1$ when x is $+0$, $+\infty$ or positive, and -1 when x is -0, $-\infty$ or negative. For $s \in \{+1, -1\}$, $s \cdot 0$ and $s \cdot \infty$ are the zero and the infinity of sign s, and for a float $z \neq \mathrm{NaN}$, $s \cdot z$ is z when $s = +1$ and z with its sign reversed when $s = -1$.

Real numbers are rounded to floats by $\mathrm{fl} : \mathbb{R} \to \mathrm{Flt}$, rounding to nearest with ties to even as IEEE 754-2019 defines it:

- $\mathrm{fl}(r) = +\infty$ if $r \geq 2^{1024} - 2^{970}$, and $\mathrm{fl}(r) = -\infty$ if $r \leq -(2^{1024} - 2^{970})$;
- otherwise, if $r \neq 0$, $\mathrm{fl}(r)$ is the finite float nearest to r and, of two equally near, the one whose significand is even (a zero counting as even); when that float is a zero, it is the zero of the sign of r;
- $\mathrm{fl}(0) = +0$.

Every Int value n with $|n| \leq 2^{53}$ is a float exactly: $\mathrm{fl}(n) = n$.

D.1.3 Characters and Strings

The set Chr of characters is a parameter of the definition (Section E.2), with an injective function

$$\mathrm{code} : \mathrm{Chr} \to \mathrm{IntRange},$$

the *code* of a character, which the basic value `Char.toCode` gives (Figure 53). For every Unicode scalar value h there is a character with the code h, the character with code point h: it is the character that the escape `\u{`h`}` denotes (Figure 1), and the one that a character constant holding that character itself denotes. A further parameter is a function

$$\mathit{other} : \mathbb{Z} \to \mathrm{Chr},$$

the character that `Char.fromCode` gives for an integer that is not a Unicode scalar value. Nothing is required of *other*: its result may be a character that is no Unicode scalar value, such as a lone surrogate, but it is a character, and so has a code in IntRange distinct from that of every other character. Nothing else is required of characters: the initial basis has no operation on characters other than equality, order, `Char.toCode` and `Char.fromCode`. A string is a finite sequence $ch_1 \cdots ch_k$ of characters ($k \geq 0$), its *length* being k; Str is the set of strings, and s ranges over Str. A character constant denotes a character and a string constant a string (Chapter 2).

D.2 The Value Environments

Every interface of ME_0 has $RA = \emptyset$, since the basis has no record alias. The constructor environments are

$$\begin{aligned}
CE \text{ of } ME_0(\texttt{Basics}) &= \{\texttt{Bool} \mapsto \{\texttt{True} \mapsto \texttt{True},\ \texttt{False} \mapsto \texttt{False}\},\\
&\quad\ \texttt{Order} \mapsto \{\texttt{LT} \mapsto \texttt{LT},\ \texttt{EQ} \mapsto \texttt{EQ},\ \texttt{GT} \mapsto \texttt{GT}\},\\
&\quad\ \texttt{Never} \mapsto \{\}\},\\
CE \text{ of } ME_0(\texttt{Maybe}) &= \{\texttt{Maybe} \mapsto \{\texttt{Just} \mapsto \texttt{Just},\ \texttt{Nothing} \mapsto \texttt{Nothing}\}\},\\
CE \text{ of } ME_0(\texttt{Result}) &= \{\texttt{Result} \mapsto \{\texttt{Ok} \mapsto \texttt{Ok},\ \texttt{Err} \mapsto \texttt{Err}\}\},
\end{aligned}$$

and CE of $ME_0(modid) = \{\}$ for every other $modid$. Thus CE maps every custom type that a module of the basis exposes to the constructors it exposes, the closed type `Never` to the empty map; the abstract types are in the domain of no CE. The value environments of the interfaces of ME_0 are shown in Figures 45 to 47. In the figures, a constructor is bound to the constructor value it names, whose arity is shown. An identifier shown with an arity is bound to the basic value of the same name, which has that arity; we write the basic values of `Basics` unqualified (`add`) and those of other modules qualified (`Debug.log`), and the basic values of Figures 45 to 47 are all distinct. An identifier marked "Elm" is bound to the value of its elm/core text (Section D.3); for `abs`, to that of its default copy, at `Float`, the copy at `Int` being bound to a new variable (Section D.3). The identifiers `e` and `pi` are bound to floats.

Each operator of the initial basis stands for a hidden variable of `Basics` or, for `::`, the list constructor (Figure 40). The *dynamic operator table*

$$\mathrm{OT}_{\mathrm{DYN}} : \mathrm{Op} \xrightarrow{\mathrm{fin}} \mathrm{Val}$$

maps each operator of Figure 3 to the value of the identifier it stands for, and is given in Figure 46. The value of an operator is $\mathrm{OT}_{\mathrm{DYN}}(op)$ in every environment: it is looked up in no environment and reached through no interface.

The interfaces of the modules `List`, `Tuple`, `String`, `Array`, `Bytes`, `Json.Encode`, `Json.Decode`, `VirtualDom` and `Html` are $(\emptyset, \{\}, \{\})$: these modules are in ME_0 for their types, which have no dynamic counterpart (the values of `Array` and `Bytes` being given in Sections D.4.9 and D.4.10). The interfaces of `Char`, `Platform`, `Platform.Cmd`, `Platform.Sub`, `Task`, `Process` and `Debug` are $(\emptyset, \{\}, VE)$, with VE as in Figure 47. These interfaces have the property of Section 7.2. The basic values `Platform.Cmd.batch` and `Platform.Cmd.map`, and their counterparts in `Platform.Sub`, are written `Cmd.batch`, `Cmd.map`, `Sub.batch` and `Sub.map` in Chapter 8, after the qualifiers of the default imports.

Besides those of the figures, the class BasVal contains the following basic values, which are bound to no identifier of the basis: the basic value `Elm.`

var or *con* ↦ *v*		arity
`True`, `False`	↦ `True`, `False`	0
`LT`, `EQ`, `GT`	↦ `LT`, `EQ`, `GT`	0
`toFloat`, `round`, `floor`, `ceiling`, `truncate`	↦ basic value	1
`compare`	↦ basic value	2
`min`, `max`	↦ basic value	2
`clamp`	↦ basic value	3
`not`	↦ basic value	1
`xor`	↦ basic value	2
`modBy`, `remainderBy`	↦ basic value	2
`negate`	↦ basic value	1
`sqrt`	↦ basic value	1
`cos`, `sin`, `tan`, `acos`, `asin`, `atan`	↦ basic value	1
`atan2`	↦ basic value	2
`isNaN`, `isInfinite`	↦ basic value	1
`e`	↦ fl(e)	
`pi`	↦ fl(π)	
`abs`	↦ Elm	
`logBase`, `degrees`, `radians`, `turns`	↦ Elm	
`fromPolar`, `toPolar`	↦ Elm	
`identity`, `always`, `never`	↦ Elm	

Figure 45: Dynamic *VE* of ME_0(`Basics`)

op	stands for	↦ $\mathrm{OT}_{\mathrm{DYN}}(op)$	arity
`<<`, `>>`	`composeL`, `composeR`	↦ Elm	
`^`	`pow`	↦ basic value	2
`*`, `/`, `//`	`mul`, `fdiv`, `idiv`	↦ basic value	2
`+`, `-`	`add`, `sub`	↦ basic value	2
`++`	`append`	↦ basic value	2
`::`	`::`	↦ `::`	2
`==`, `/=`	`eq`, `neq`	↦ basic value	2
`<`, `>`, `<=`, `>=`	`lt`, `gt`, `le`, `ge`	↦ basic value	2
`&&`, `||`	`and`, `or`	↦ basic value	2
`|>`, `<|`	`apR`, `apL`	↦ Elm	

Figure 46: The Dynamic Operator Table, $\mathrm{OT}_{\mathrm{DYN}}$

var or *con* ↦ *v*	arity
`Maybe`	
`Just` ↦ `Just`	1
`Nothing` ↦ `Nothing`	0
`Result`	
`Ok`, `Err` ↦ `Ok`, `Err`	1
`Char`	
`toCode`, `fromCode` ↦ basic value	1
`Debug`	
`log` ↦ basic value	2
`todo`, `toString` ↦ basic value	1
`Platform`	
`worker` ↦ basic value	1
`Platform.Cmd` and `Platform.Sub`	
`none` ↦ Elm	
`batch` ↦ basic value	1
`map` ↦ basic value	2
`Task`	
`succeed`, `fail` ↦ basic value	1
`andThen`, `onError` ↦ basic value	2
`map`, `map2`, `map3`, `map4`, `map5` ↦ Elm	
`sequence`, `mapError`, `perform`, `attempt` ↦ Elm	
`Process`	
`spawn`, `sleep`, `kill` ↦ basic value	1

Figure 47: Dynamic *VE* of the Other Modules of ME_0

`Kernel.Basics.log` of arity 1 (Section D.3); the basic value `Task.command` of arity 1, to which the environment of the texts of `Task` binds the hidden variable `command` (Section D.3), and whose APPLY is given in Section 8.6; the basic values `Array.fromList` and `Array.toList` of arity 1, which Chapter 8 applies to encode and decode arrays at ports (Section D.4.9); and the values of ports, outport(p, τ) and inport(p, τ), of arity 1 (Section 8.7).

D.3 Values Written in Elm

A value of the basis whose definition in elm/core 1.0.5 is written in Elm is defined by that text, except `min`, `max`, `clamp` and `negate`, which are basic values (below); the other basic values are those whose definition is a reference to the kernel. The texts of the values of `Basics` marked "Elm" in Figures 45 and 46 are shown in Figure 48, as they stand in the file `Basics.elm` of elm/core 1.0.5.

```
abs n =
  if lt n 0 then -n else n

logBase base number =
  fdiv
    (Elm.Kernel.Basics.log number)
    (Elm.Kernel.Basics.log base)

radians angleInRadians =
  angleInRadians

degrees angleInDegrees =
  fdiv (mul angleInDegrees pi) 180

turns angleInTurns =
  mul (mul 2 pi) angleInTurns
```

```
fromPolar (radius, theta) =
  ( mul radius (cos theta)
  , mul radius (sin theta)
  )

toPolar ( x, y ) =
  ( sqrt (add (mul x x) (mul y y))
  , atan2 y x
  )

composeL g f x =
  g (f x)

composeR f g x =
  g (f x)

apR x f =
  f x

apL f x =
  f x

identity x =
  x

always a _ =
  a

never (JustOneMore nvr) =
  never nvr
```

Figure 48: The Values of `Basics` Written in Elm

Each text of Figure 48 is a declaration *dec* that, by the derived forms of

Appendix A, has the form *var* `=` \ *atpat* `->` *exp*. Let

$$E_{\mathrm{K}} = E_0 + (QE_{\mathrm{K}}, VE_{\mathrm{K}}), \quad \text{where}$$
$$QE_{\mathrm{K}} = \{\texttt{Elm.Kernel.Basics} \mapsto (\{\}, \{\texttt{log} \mapsto \texttt{Elm.Kernel.Basics.log}\}),$$
$$\texttt{Basics} \mapsto (\{\}, VE_{\mathrm{K}})\},$$

and VE_{K} binds every identifier of `Basics` that Figures 45 and 46 do not mark "Elm" to the value shown there, the hidden variables of the operators included, and binds the hidden constructor `JustOneMore` of the closed type `Never` to the constructor value `JustOneMore` of arity 1. The qualifier `Basics` is bound in E_{K} to the environment of the identifiers of `Basics` that VE_{K} binds, so that the long identifiers `Basics.True`, `Basics.False` and `Basics.negate` that the derived forms of Appendix A introduce into these texts (by `if` and by prefix minus) denote the values `True`, `False` and `negate` of Figure 45; the texts are part of the definition, not of a program, so the rule that a module cannot qualify its own names does not apply to them. No text refers to a value that another text defines. The value bound to *var* in ME_0(`Basics`) is $VE(var)$, where

$$E_{\mathrm{K}} \vdash dec \Rightarrow VE$$

by rule (90), or, for `never`, whose declaration is recursive, by rule (91). In each case VE is unique: it binds *var* to a closure. In the text of `abs`, `-n` is the derived form `Basics.negate n`; the text of `negate` itself is `-n`, which by that derived form would apply `negate` to its own argument, so `negate` is a basic value. `min` and `max` are basic values at every comparable type: at floats their APPLY is the minimum and maximum of IEEE 754-2019, which their elm/core texts do not give, and at every other comparable type it is the result of those texts (Section D.4.6). `clamp` is a basic value likewise: at floats its APPLY is built on the minimum and maximum of IEEE 754-2019, which its elm/core text does not give, and at Int values it is the result of that text (Sections D.4.2 and D.4.6). `Debug.log`, `Debug.todo` and `Debug.toString` are kernel references in elm/core, and are basic values.

The values of `Platform.Cmd`, `Platform.Sub` and `Task` marked "Elm" in Figure 47 are defined in the same way by their texts in the files `Platform/Cmd.elm`, `Platform/Sub.elm` and `Task.elm` of elm/core 1.0.5. The text of a value of the module *modid* is evaluated in the environment

$$E_{modid} = E_0 + (QE_{modid}, VE_{modid}),$$

where QE_{modid} binds each qualifier that the texts of *modid* use to an environment binding the identifiers they write under it, to their values in ME_0 or, for a further function of elm/core, to the value of its text (below), and VE_{modid} binds the identifiers of the interfaces of *modid* and of the modules it imports unqualified (among them the constructors `Ok` and `Err`) to their values in ME_0. The values marked "Elm" are taken in an order in which each text

refers only to values already defined: `Task.map` before `Task.perform`, and `Task.map2` before `Task.sequence`. For *modid* = `Task`, $VE_{\texttt{Task}}$ also binds the two hidden identifiers of `Task` (Section C.2): the variable `command` to the basic value `Task.command`, whose APPLY is that of Figure 24, and the constructor `Perform` of the custom type `MyCmd` to the constructor value `Perform`, of arity 1. Neither is in ME_0(`Task`). The texts of `Task.perform` and `Task.attempt` apply `command` to a value `Perform` *tk*, which APPLY makes a leaf command of the task manager. Where such a text refers to a further function of elm/core (`Task.sequence` applies `List.foldr`), that function is defined by its text likewise, and the kernel function `Elm.Kernel.List.cons` is the list constructor `::`. Chapter 8 states what these texts give (Section 8.6).

The texts are specialised over their `number` variables with the rest of the program (Section 6.2): the copy of `abs` used at `Float` compares with the float 0, and the constants `180` and `2` in the texts of `degrees` and `turns`, whose type is `Float`, denote floats. Of the values of the basis written in Elm, only `abs` has a type scheme that quantifies a variable of class number (Figure 38); the identifier `abs` of Figure 45 is bound to its default copy, at `Float`. Besides this, the value environment of ME_0(`Basics`) binds the new variable of the copy of the declaration of `abs` at `Int` that the specialisation makes, to the value of that copy, evaluated in E_{K} as the default copy is. An occurrence of `abs` at `Int` thus finds its copy: qualified, through the qualifier `Basics`; unqualified, through the default import `import Basics exposing (..)`, whose exposing list gives the whole of that value environment; and through an exposing item `abs` of an import, which stands for the variable and the new variable of its copy (Section 7.1). So `abs -1.5` is the float 1.5, and `abs (3 // -1)` the Int value 3.

Thus the order in which a basis function written in Elm applies the functions it is given, for instance `List.foldr` from the last element of its list, is fixed by its text. Consequences of the texts of Figure 48 that the definitions of Section D.4 make visible are noted there.

D.4 The Function APPLY

For a basic value b of arity n and values $v_1, \ldots, v_n$, APPLY$(b, v_1 \cdots v_n)$ is a finite set of values (Section 6.5); in this section we separate the arguments by commas, writing APPLY$(b, v_1, \ldots, v_n)$. The set has exactly one member except where stated: it has two members when an equality test reaches two functions (Section D.4.5), and none for `Debug.todo` (Section D.4.8). In this section m and n range over IntRange, x, y and z over Flt, and s and s' over Str. The definitions cover every argument that the evaluation of a program that elaborates supplies to b.

Overloading. The basic values `add`, `sub`, `mul`, `pow`, `negate` and `clamp` act on Int values and on floats; `compare`, `lt`, `gt`, `le`, `ge`, `min` and `max` on all

comparable values; `append` on strings and on lists; `eq` and `neq` on all values. APPLY dispatches on the special values it is given: each case is defined below for each kind of argument. Before evaluation a program is specialised over its `number` variables, so that every integer constant has the type `Int` or `Float`, and an integer constant of type `Float` denotes the nearest float (Section 6.2). Hence both arguments of a number operation are Int values or both are floats, the three arguments of `clamp` are all Int values or all floats, every argument of a basic value of floats is a float, and an Int value never meets a float in equality or order.

D.4.1 Int Arithmetic

For real r, $\lfloor r \rfloor$ is the greatest integer at most r, $\lceil r \rceil$ the least integer at least r, and $\mathrm{trunc}(r)$ is $\lfloor r \rfloor$ when $r \geq 0$ and $\lceil r \rceil$ when $r < 0$. On Int values, APPLY is given in Figure 49. An exact result is normalised into IntRange; where an operation has no mathematically correct result, the result is 0.

$$
\begin{aligned}
\mathrm{APPLY}(\texttt{add}, m, n) &= \{\mathrm{norm}(m+n)\} \\
\mathrm{APPLY}(\texttt{sub}, m, n) &= \{\mathrm{norm}(m-n)\} \\
\mathrm{APPLY}(\texttt{mul}, m, n) &= \{\mathrm{norm}(m \cdot n)\} \\
\mathrm{APPLY}(\texttt{negate}, n) &= \{\mathrm{norm}(-n)\} \\
\mathrm{APPLY}(\texttt{idiv}, m, n) &= \{\mathrm{norm}(\mathrm{trunc}(m/n))\} && \text{if } n \neq 0 \\
&= \{0\} && \text{if } n = 0 \\
\mathrm{APPLY}(\texttt{modBy}, n, m) &= \{m - n \cdot \lfloor m/n \rfloor\} && \text{if } n \neq 0 \\
&= \{0\} && \text{if } n = 0 \\
\mathrm{APPLY}(\texttt{remainderBy}, n, m) &= \{m - n \cdot \mathrm{trunc}(m/n)\} && \text{if } n \neq 0 \\
&= \{0\} && \text{if } n = 0 \\
\mathrm{APPLY}(\texttt{pow}, m, n) &= \{\mathrm{norm}(m^n)\} && \text{if } n \geq 0 \\
&= \{1\} && \text{if } n < 0 \text{ and } m = 1 \\
&= \{(-1)^n\} && \text{if } n < 0 \text{ and } m = -1 \\
&= \{0\} && \text{if } n < 0 \text{ and } m \notin \{1, -1\}
\end{aligned}
$$

Figure 49: APPLY on Int Values

In the figure, $m^0 = 1$ for every m, including 0. The first argument of `modBy` and `remainderBy` is the divisor: `modBy 4 -5` is 3 and `remainderBy 4 -5` is -1. The following property, which may easily be demonstrated, shows that `modBy` and `remainderBy` need no normalisation.

> For $n \neq 0$, $m - n \cdot \lfloor m/n \rfloor$ is 0 or has the sign of n, $m - n \cdot \mathrm{trunc}(m/n)$ is 0 or has the sign of m, and both are less than $|n|$ in absolute value; hence both are members of IntRange.

Thus `3 // 0`, `modBy 0 3` and `remainderBy 0 3` are all 0, and, at `Int`, `1 ^ -1` is 1, `(-1) ^ -3` is -1 and `2 ^ -1` is 0. The quotient `(min IntRange)`

`// -1` is norm($-\min$ IntRange). By the text of `abs` (Figure 48), `abs n` is norm($|n|$).

D.4.2 Floating-Point Arithmetic

On floats, `add`, `sub`, `mul`, `fdiv`, `negate` and `sqrt` are the operations of IEEE 754-2019 on binary64 numbers, rounding to nearest with ties to even; Figure 50 gives them case by case, the first applicable line giving the result.

$$
\begin{aligned}
\text{APPLY}(\texttt{negate}, x) &= \{\text{NaN}\} && \text{if } x \text{ is NaN}\\
&= \{(-1)\cdot x\} && \text{otherwise}\\
\text{APPLY}(\texttt{add}, x, y) &= \{\text{NaN}\} && \text{if } x \text{ or } y \text{ is NaN,}\\
& && \text{or } \{x, y\} = \{+\infty, -\infty\}\\
&= \{x\} && \text{if } x \in \{+\infty, -\infty\}\\
&= \{y\} && \text{if } y \in \{+\infty, -\infty\}\\
&= \{-0\} && \text{if } x \text{ and } y \text{ are both } -0\\
&= \{\text{fl}(x+y)\} && \text{otherwise}\\
\text{APPLY}(\texttt{sub}, x, y) &= \text{APPLY}(\texttt{add}, x, z) && \text{where APPLY}(\texttt{negate}, y) = \{z\}\\
\text{APPLY}(\texttt{mul}, x, y) &= \{\text{NaN}\} && \text{if } x \text{ or } y \text{ is NaN,}\\
& && \text{or one is a zero and one an infinity}\\
&= \{\text{sgn}(x)\text{sgn}(y)\cdot\infty\} && \text{if } x \text{ or } y \text{ is an infinity}\\
&= \{\text{sgn}(x)\text{sgn}(y)\cdot 0\} && \text{if } x \text{ or } y \text{ is a zero}\\
&= \{\text{fl}(x\cdot y)\} && \text{otherwise}\\
\text{APPLY}(\texttt{fdiv}, x, y) &= \{\text{NaN}\} && \text{if } x \text{ or } y \text{ is NaN,}\\
& && \text{or both are zeros, or both infinities}\\
&= \{\text{sgn}(x)\text{sgn}(y)\cdot\infty\} && \text{if } x \text{ is an infinity, or } y \text{ a zero}\\
&= \{\text{sgn}(x)\text{sgn}(y)\cdot 0\} && \text{if } x \text{ is a zero, or } y \text{ an infinity}\\
&= \{\text{fl}(x/y)\} && \text{otherwise}\\
\text{APPLY}(\texttt{sqrt}, x) &= \{\text{NaN}\} && \text{if } x \text{ is NaN, or } x < 0\\
&= \{x\} && \text{if } x \in \{+0, -0, +\infty\}\\
&= \{\text{fl}(\sqrt{x})\} && \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\text{APPLY}(\texttt{isNaN}, x) &= \{\texttt{True}\} && \text{if } x \text{ is NaN}\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{isInfinite}, x) &= \{\texttt{True}\} && \text{if } x \in \{+\infty, -\infty\}\\
&= \{\texttt{False}\} && \text{otherwise}
\end{aligned}
$$

Figure 50: APPLY on Floats: Arithmetic

Thus `1 / 0` is $+\infty$, `1 / -0` is $-\infty$, `0 / 0` is NaN, and `x - x` is $+0$ for every finite `x`. In the case $x < 0$ of `sqrt`, x is not a zero, so `sqrt -0` is -0. By the text of `abs`, `abs -0` is -0 and `abs` of NaN is NaN, since `lt` is false for both (Section D.4.6).

At floats, `min` and `max` are the operations *minimum* and *maximum* of

IEEE 754-2019, the first applicable line giving the result:

$$\begin{aligned}
\text{APPLY}(\texttt{min}, x, y) &= \{\text{NaN}\} && \text{if } x \text{ or } y \text{ is NaN}\\
&= \{-0\} && \text{if } x \text{ and } y \text{ are zeros and one of them is } -0\\
&= \{x\} && \text{if } x \leq y\\
&= \{y\} && \text{otherwise}\\
\text{APPLY}(\texttt{max}, x, y) &= \{\text{NaN}\} && \text{if } x \text{ or } y \text{ is NaN}\\
&= \{+0\} && \text{if } x \text{ and } y \text{ are zeros and one of them is } +0\\
&= \{x\} && \text{if } x \geq y\\
&= \{y\} && \text{otherwise.}
\end{aligned}$$

These lines apply when both arguments are floats; at the other comparable values APPLY takes the cases of Section D.4.6. The case is selected by the arguments, as for `lt`: the type schemes of `min` and `max` quantify a variable of class comparable, not number, so number specialisation makes no copy of them (Section 6.2). The result does not depend on the order of the arguments: with `nan` for `0/0`, `max nan 1` and `max 1 nan` are both NaN, `max 0 -0` and `max -0 0` are $+0$, and `min -0 0` and `min 0 -0` are -0.

At floats, `clamp` is built on these operations:

$$\begin{aligned}
\text{APPLY}(\texttt{clamp}, x, y, z) = \text{APPLY}(\texttt{min}, z', y)\\
\text{where APPLY}(\texttt{max}, z, x) = \{z'\},
\end{aligned}$$

when x, y and z are floats; the first argument is the lower bound, the second the upper bound, and `clamp lo hi x` is `min (max x lo) hi`. At Int values APPLY takes the cases of Section D.4.6; the case is selected by the arguments, as for `add`. When `lo` is greater than `hi`, the result is `hi` unless an argument is NaN: `clamp 10 0 5` is $+0$. A NaN in any argument gives NaN: `clamp 0 1 nan`, `clamp nan 1 0.5` and `clamp 0 nan 0.5` are all NaN. Zeros are ordered as by `max` and `min`: `clamp 0 1 -0` is $+0$, the maximum of -0 and $+0$ being $+0$, and `clamp -1 -0 0` is -0.

D.4.3 Powers and Elementary Functions

The power at floats, the trigonometric functions, and the logarithm `Elm.Kernel.Basics.log` on which `logBase` is built, denote the real functions x^y, sin, cos, tan, arcsin, arccos, arctan, the natural logarithm ln, and the two-argument arctangent, rounded to a float, with the special values that IEEE 754-2019 (Section 9.2) recommends. The two-argument arctangent of a point $(x, y) \neq (0, 0)$ is the angle θ with $-\pi < \theta \leq \pi$ such that $(x, y) = \rho \cdot (\cos\theta, \sin\theta)$ for some $\rho > 0$; the first argument of `atan2` is y. The power is given in Figure 51 and the other functions in Figure 52; in each, the first applicable line gives the result. An integer k is *odd* when $k = 2j + 1$ for an integer j.

The first two lines apply even when the other argument is NaN. In the last line of Figure 51, x and y are finite and x is not a zero; when $x < 0$, y is

$$
\begin{array}{lll}
\text{APPLY}(\texttt{pow}, x, y) = \{1\} & & \text{if } y \text{ is a zero} \\
= \{1\} & & \text{if } x \text{ is } 1 \\
= \{\text{NaN}\} & & \text{if } x \text{ or } y \text{ is NaN} \\
= \{\text{sgn}(x) \cdot \infty\} & & \text{if } x \text{ is a zero and } y < 0 \text{ an odd integer} \\
= \{+\infty\} & & \text{if } x \text{ is a zero and } y < 0 \\
= \{x\} & & \text{if } x \text{ is a zero and } y > 0 \text{ an odd integer} \\
= \{+0\} & & \text{if } x \text{ is a zero and } y > 0 \\
= \{1\} & & \text{if } x = -1 \text{ and } y \in \{+\infty, -\infty\} \\
= \{+0\} & & \text{if } |x| < 1 \text{ and } y = +\infty \\
= \{+0\} & & \text{if } |x| > 1 \text{ and } y = -\infty \\
= \{+\infty\} & & \text{if } |x| > 1 \text{ and } y = +\infty \\
= \{+\infty\} & & \text{if } |x| < 1 \text{ and } y = -\infty \\
= \{+0\} & & \text{if } x = +\infty \text{ and } y < 0 \\
= \{+\infty\} & & \text{if } x = +\infty \text{ and } y > 0 \\
= \{-0\} & & \text{if } x = -\infty \text{ and } y < 0 \text{ an odd integer} \\
= \{-\infty\} & & \text{if } x = -\infty \text{ and } y > 0 \text{ an odd integer} \\
= \{+0\} & & \text{if } x = -\infty \text{ and } y < 0 \\
= \{+\infty\} & & \text{if } x = -\infty \text{ and } y > 0 \\
= \{\text{NaN}\} & & \text{if } x < 0 \text{ and } y \text{ is not an integer} \\
= \{\text{fl}(x^y)\} & & \text{otherwise}
\end{array}
$$

Figure 51: APPLY on Floats: Power

an integer and $x^y = (-1)^y|x|^y$. Thus `1 ^ (1/0)` is 1, and `(-8) ^ (1/3)` is NaN.

In the first part of Figure 52, b ranges over `sin`, `cos`, `tan`, `asin`, `acos`, `atan` and `Elm.Kernel.Basics.log`, and f is the real function of b: sin, cos, tan, arcsin, arccos, arctan or ln; on the arguments of the line $\{\text{fl}(f(x))\}$, $f(x)$ is a real number. In the last line for `atan2`, θ is the two-argument arctangent of (x, y). The values `e` and `pi` are fl(e) and fl(π), and `logBase`, `degrees`, `radians`, `turns`, `fromPolar` and `toPolar` are given by their texts (Figure 48), so that `logBase 10 1000` is the quotient of two rounded logarithms, rounded again.

The lines $\{\text{fl}(x^y)\}$, $\{\text{fl}(f(x))\}$ and $\{\text{fl}(\theta)\}$ of Figures 51 and 52 give the correctly rounded result. On those arguments, and only there, the accuracy with which an implementation computes the result is implementation-dependent, and an implementation may give a float near the real result in its place (Section E.3). The operations of Figure 50 allow no such latitude.

$$
\begin{aligned}
\mathrm{APPLY}(b, \mathrm{NaN}) &= \{\mathrm{NaN}\} && \text{for each } b \text{ of the figure}\\
\mathrm{APPLY}(b, x) &= \{x\} && \text{if } x \text{ is a zero, for } b \in \{\texttt{sin}, \texttt{tan}, \texttt{asin}, \texttt{atan}\}\\
\mathrm{APPLY}(b, x) &= \{\mathrm{NaN}\} && \text{if } x \in \{+\infty, -\infty\},\ \text{for } b \in \{\texttt{sin}, \texttt{cos}, \texttt{tan}\}\\
\mathrm{APPLY}(b, x) &= \{\mathrm{NaN}\} && \text{if } |x| > 1,\ \text{for } b \in \{\texttt{asin}, \texttt{acos}\}\\
\mathrm{APPLY}(\texttt{atan}, x) &= \{\mathrm{sgn}(x) \cdot \mathrm{fl}(\pi/2)\} && \text{if } x \in \{+\infty, -\infty\}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{APPLY}(\texttt{Elm.Kernel.Basics.log}, x) &= \{-\infty\} && \text{if } x \text{ is a zero}\\
&= \{\mathrm{NaN}\} && \text{if } x < 0\\
&= \{+\infty\} && \text{if } x = +\infty\\
\mathrm{APPLY}(b, x) &= \{\mathrm{fl}(f(x))\} && \text{otherwise, for each } b
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{APPLY}(\texttt{atan2}, y, x) &= \{\mathrm{NaN}\} && \text{if } x \text{ or } y \text{ is NaN}\\
&= \{y\} && \text{if } y \text{ is a zero, and } x \text{ is } +0 \text{ or } x > 0\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(\pi)\} && \text{if } y \text{ is a zero, and } x \text{ is } -0 \text{ or } x < 0\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(\pi/2)\} && \text{if } x \text{ is a zero}\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(\pi/2)\} && \text{if } y \text{ is an infinity and } x \text{ is finite}\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(3\pi/4)\} && \text{if } y \text{ is an infinity and } x = -\infty\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(\pi/4)\} && \text{if } y \text{ is an infinity and } x = +\infty\\
&= \{\mathrm{sgn}(y) \cdot \mathrm{fl}(\pi)\} && \text{if } x = -\infty\\
&= \{\mathrm{sgn}(y) \cdot 0\} && \text{if } x = +\infty\\
&= \{\mathrm{fl}(\theta)\} && \text{otherwise}
\end{aligned}
$$

Figure 52: APPLY on Floats: Elementary Functions

D.4.4 Conversions

An Int value becomes a float by rounding, and a float becomes an Int value by *saturation* into IntRange:

$$
\mathrm{sat}(k) = \begin{cases} k & \text{if } k \in \mathrm{IntRange} \\ \max \mathrm{IntRange} & \text{if } k = +\infty \text{ or } k > \max \mathrm{IntRange} \\ \min \mathrm{IntRange} & \text{if } k = -\infty \text{ or } k < \min \mathrm{IntRange} \end{cases}
$$

for $k \in \mathbb{Z} \cup \{+\infty, -\infty\}$. The conversions are given in Figure 53.

In the last two lines b is any of `round`, `floor`, `ceiling` and `truncate`. The sum $x + \frac{1}{2}$ is exact, not rounded to a float, so `round` rounds a half upward: `round 2.5` is 3 and `round -2.5` is -2. A float zero of either sign converts to the one Int zero, and `toFloat 0` is $+0$, so `1 / toFloat (negate 0)` is $+\infty$.

D.4.5 Equality

The *function values* are the closures, the basic values with their partial applications, and the constructor values awaiting arguments:

$$\mathrm{Fun} = \mathrm{Closure} \cup \mathrm{BasApp} \cup \{\mathit{con}\ v_1 \cdots v_k \in \mathrm{ConVal} \mid k < \mathrm{arity}(\mathit{con})\}.$$

Equality is structural on all values, and is defined by a set $\mathrm{Eq}(v, v') \subseteq \{\texttt{True}, \texttt{False}\}$ in Figure 54, by recursion on the structure of v and v' (values

$$
\begin{array}{rll}
\mathrm{APPLY}(\texttt{toFloat}, n) = \{\mathrm{fl}(n)\} & & \\
\mathrm{APPLY}(\texttt{Char.toCode}, ch) = \{\mathrm{code}(ch)\} & & \\
\mathrm{APPLY}(\texttt{Char.fromCode}, n) = \{ch\} & & \text{if } n \text{ is a Unicode scalar value} \\
 & & \text{and } \mathrm{code}(ch) = n \\
 = \{other(n)\} & & \text{otherwise} \\
\mathrm{APPLY}(\texttt{round}, x) = \{\mathrm{sat}(\lfloor x + \tfrac{1}{2} \rfloor)\} & & \text{if } x \in \mathrm{Flt}_{\mathrm{fin}} \\
\mathrm{APPLY}(\texttt{floor}, x) = \{\mathrm{sat}(\lfloor x \rfloor)\} & & \text{if } x \in \mathrm{Flt}_{\mathrm{fin}} \\
\mathrm{APPLY}(\texttt{ceiling}, x) = \{\mathrm{sat}(\lceil x \rceil)\} & & \text{if } x \in \mathrm{Flt}_{\mathrm{fin}} \\
\mathrm{APPLY}(\texttt{truncate}, x) = \{\mathrm{sat}(\mathrm{trunc}(x))\} & & \text{if } x \in \mathrm{Flt}_{\mathrm{fin}} \\
\mathrm{APPLY}(b, x) = \{\mathrm{sat}(x)\} & & \text{if } x \in \{+\infty, -\infty\} \\
\mathrm{APPLY}(b, \mathrm{NaN}) = \{0\} & &
\end{array}
$$

Figure 53: APPLY on Conversions

are finite, and the recursion does not enter a function value). Then

$$
\begin{array}{rcl}
\mathrm{APPLY}(\texttt{eq}, v, v') & = & \mathrm{Eq}(v, v'), \\
\mathrm{APPLY}(\texttt{neq}, v, v') & = & \{\texttt{True} \mid \texttt{False} \in \mathrm{Eq}(v, v')\} \\
 & & \cup \{\texttt{False} \mid \texttt{True} \in \mathrm{Eq}(v, v')\}.
\end{array}
$$

The clauses are tried in order. Lists, `Maybe` and `Result` values, `Bool` and `Order` values, arrays and the values of type `Bytes` are constructor values, so they are compared by the fourth clause: two lists are equal when they have the same length and equal elements. A record type gives all its records the same domain, so the condition $\mathrm{Dom}\, r = \mathrm{Dom}\, r'$ always holds in a program that elaborates. The following property may easily be demonstrated.

> $\mathrm{Eq}(v, v')$ is non-empty. It is $\{\texttt{True}, \texttt{False}\}$ only if, at some depth, corresponding components of v and v' are both function values, and it is $\{\texttt{False}\}$ whenever some pair of corresponding components that are not function values is unequal.

Equality is thus deterministic except when the comparison reaches two function values, where it may give either `True` or `False`: `(\x -> x) == (\x -> x)` may evaluate to either. This is the only non-determinism in the dynamic semantics of the Core (rule (84)). Because equality looks only at structure, the identity of values plays no part. NaN is unequal to every float, itself included, so a value with a NaN component is not equal to itself: if `t` is `(0/0, 1)`, then `t == t` is `False`. The two zeros of the floats are equal. Implementations whose equality consults identity are recorded in Section E.4.

D.4.6 Order

The *comparable values* are the Int values, the floats, the characters, the strings, and the lists and tuples whose elements or components are comparable

1. If $v, v' \in \mathrm{Fun}$, then $\mathrm{Eq}(v, v') = \{\texttt{True}, \texttt{False}\}$.
2. If v and v' are floats: $\mathrm{Eq}(v, v') = \{\texttt{True}\}$ if neither is NaN and $v = v'$ (so that $+0$ equals -0), and $\{\texttt{False}\}$ otherwise.
3. If v and v' are both Int values, both characters, both strings, both the unit value, both process identifiers, or both boundary values (Chapter 8) that are `null`, `true`, `false`, numbers, strings or byte sequences: $\mathrm{Eq}(v, v') = \{\texttt{True}\}$ if v and v' are the same, and $\{\texttt{False}\}$ otherwise.
4. If v and v' have the same form, with components $v_1, \ldots, v_k$ and $v'_1, \ldots, v'_k$ ($k \geq 0$), as listed below, then $\texttt{True} \in \mathrm{Eq}(v, v')$ if $\texttt{True} \in \mathrm{Eq}(v_i, v'_i)$ for every i, and $\texttt{False} \in \mathrm{Eq}(v, v')$ if $\texttt{False} \in \mathrm{Eq}(v_i, v'_i)$ for some i. The forms are:
   - tuples with the same number of components;
   - constructor values $con\ v_1 \cdots v_k$ and $con\ v'_1 \cdots v'_k$ with the same constructor name and $k = \mathrm{arity}(con)$;
   - records r and r' with $\mathrm{Dom}\, r = \mathrm{Dom}\, r'$, the components being $r(lab)$ and $r'(lab)$ for each $lab \in \mathrm{Dom}\, r$;
   - arrays of boundary values (JSON arrays among them) with the same number of elements, and objects of boundary values with the same set of keys, the components being the values at each key;
   - effect values (Chapter 8) of the same form, with the same primitives, managers and port names, the components being the values, boundary values, tasks, commands and subscriptions they contain.
5. Otherwise $\mathrm{Eq}(v, v') = \{\texttt{False}\}$.

Figure 54: Equality of Values

values; they are the values of the types admissible for the class comparable (Section 4.4). For comparable values v and v' of one type, $\mathrm{cmp}(v, v') \in \{\mathtt{LT}, \mathtt{EQ}, \mathtt{GT}\}$ is defined in Figure 55. Then

$$\mathrm{APPLY}(\mathtt{compare}, v, v') = \{\mathrm{cmp}(v, v')\}.$$

1. Int values m, n: $\mathrm{cmp}(m, n)$ is `LT` if $m < n$, `EQ` if $m = n$, and `GT` if $m > n$.
2. Floats x, y: $\mathrm{cmp}(x, y)$ is `EQ` if both are NaN, `GT` if only x is NaN, `LT` if only y is NaN, and otherwise `LT`, `EQ` or `GT` as $x < y$, $x = y$ or $x > y$.
3. Characters: $\mathrm{cmp}(ch, ch') = \mathrm{cmp}(\mathrm{code}(ch), \mathrm{code}(ch'))$.
4. Strings $s = ch_1 \cdots ch_k$ and $s' = ch'_1 \cdots ch'_l$: if $ch_i \neq ch'_i$ for some $i \leq \min(k, l)$, and i is the least such, then $\mathrm{cmp}(s, s') = \mathrm{cmp}(ch_i, ch'_i)$; otherwise $\mathrm{cmp}(s, s') = \mathrm{cmp}(k, l)$.
5. Lists: $\mathrm{cmp}(\mathtt{[]}, \mathtt{[]}) = \mathtt{EQ}$, $\mathrm{cmp}(\mathtt{[]}, v_1 \mathtt{::} v_2) = \mathtt{LT}$, $\mathrm{cmp}(v_1 \mathtt{::} v_2, \mathtt{[]}) = \mathtt{GT}$, and $\mathrm{cmp}(v_1 \mathtt{::} v_2, v'_1 \mathtt{::} v'_2)$ is $\mathrm{cmp}(v_1, v'_1)$ if that is not `EQ`, and $\mathrm{cmp}(v_2, v'_2)$ otherwise.
6. Tuples $(v_1, \ldots, v_k)$ and $(v'_1, \ldots, v'_k)$: $\mathrm{cmp}(v_i, v'_i)$ for the least i with $\mathrm{cmp}(v_i, v'_i) \neq \mathtt{EQ}$, and `EQ` if there is none.

Figure 55: Order of Comparable Values

> For comparable values v, v' and v'' of one type, $\mathrm{cmp}(v, v') = \mathtt{LT}$ if and only if $\mathrm{cmp}(v', v) = \mathtt{GT}$, $\mathrm{cmp}(v, v) = \mathtt{EQ}$, and if $\mathrm{cmp}(v, v') \neq \mathtt{GT}$ and $\mathrm{cmp}(v', v'') \neq \mathtt{GT}$ then $\mathrm{cmp}(v, v'') \neq \mathtt{GT}$.

Thus `compare` is a total order on each comparable type, up to the values it makes `EQ`, and sorting by it has a defined result. It is `EQ` for the two zeros and for two NaNs, and puts NaN above every other float; it therefore disagrees with equality at NaN, which is unequal to itself. Characters are ordered by code, and strings by their sequences of characters, so the order of one-character strings is that of their characters.

The order operators are the IEEE 754 predicates at floats and are derived

from cmp at every other comparable type:

$$\begin{aligned}
\text{APPLY}(\texttt{lt}, x, y) &= \{\texttt{True}\} && \text{if neither is NaN and } x < y\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{le}, x, y) &= \{\texttt{True}\} && \text{if neither is NaN and } x \le y\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{gt}, x, y) &= \text{APPLY}(\texttt{lt}, y, x)\\
\text{APPLY}(\texttt{ge}, x, y) &= \text{APPLY}(\texttt{le}, y, x)
\end{aligned}$$

$$\begin{aligned}
\text{APPLY}(\texttt{lt}, v, v') &= \{\texttt{True}\} && \text{if } \text{cmp}(v, v') = \texttt{LT}\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{le}, v, v') &= \{\texttt{True}\} && \text{if } \text{cmp}(v, v') \neq \texttt{GT}\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{gt}, v, v') &= \text{APPLY}(\texttt{lt}, v', v)\\
\text{APPLY}(\texttt{ge}, v, v') &= \text{APPLY}(\texttt{le}, v', v)
\end{aligned}$$

where in the first six lines x and y are floats, and in the last six v and v' are comparable values of any other kind. Hence `nan < 1`, `nan > 1` and `nan >= nan` are `False`, where `nan` is `0/0`, while `[ nan ] > [ 1 ]` is `True`, the list being compared by cmp.

At comparable values other than floats, `min` and `max` give the results of their elm/core texts, `min x y = if lt x y then x else y` and `max x y = if gt x y then x else y`:

$$\begin{aligned}
\text{APPLY}(\texttt{min}, v, v') &= \{v\} && \text{if } \text{cmp}(v, v') = \texttt{LT}\\
&= \{v'\} && \text{otherwise}\\
\text{APPLY}(\texttt{max}, v, v') &= \{v\} && \text{if } \text{cmp}(v, v') = \texttt{GT}\\
&= \{v'\} && \text{otherwise,}
\end{aligned}$$

where v and v' are comparable values that are not floats; the cases at floats are those of Section D.4.2. Thus `max [ nan ] [ 1 ]` and `max [ 1 ] [ nan ]` are both `[ nan ]`, the lists being compared by cmp, which puts NaN above every other float.

At Int values, `clamp` gives the result of its elm/core text,

```
clamp low high number =
  if lt number low then low
  else if gt number high then high else number,
```

the first applicable line giving the result:

$$\begin{aligned}
\text{APPLY}(\texttt{clamp}, v, v', v'') &= \{v\} && \text{if } \text{APPLY}(\texttt{lt}, v'', v) = \{\texttt{True}\}\\
&= \{v'\} && \text{if } \text{APPLY}(\texttt{gt}, v'', v') = \{\texttt{True}\}\\
&= \{v''\} && \text{otherwise,}
\end{aligned}$$

where v, v' and v'' are not floats, and are therefore Int values, the type scheme of `clamp` quantifying a variable of class number; the case at floats is that of Section D.4.2. When `lo` is greater than `hi`, `clamp lo hi x` at Int values is `lo` if `x < lo`, else `hi` if `x > hi`, else `x`: `clamp 10 0 5` is 10, where at floats it is +0.

D.4.7 Booleans, Appending and Functions

The Boolean values are the constructor values `True` and `False`.

$$\begin{aligned}
\text{APPLY}(\texttt{not}, v) &= \{\texttt{False}\} && \text{if } v = \texttt{True}\\
&= \{\texttt{True}\} && \text{otherwise}\\
\text{APPLY}(\texttt{and}, v, v') &= \{\texttt{True}\} && \text{if } v = v' = \texttt{True}\\
&= \{\texttt{False}\} && \text{otherwise}\\
\text{APPLY}(\texttt{or}, v, v') &= \{\texttt{False}\} && \text{if } v = v' = \texttt{False}\\
&= \{\texttt{True}\} && \text{otherwise}\\
\text{APPLY}(\texttt{xor}, v, v') &= \{\texttt{True}\} && \text{if } v \neq v'\\
&= \{\texttt{False}\} && \text{otherwise}
\end{aligned}$$

The basic values `and` and `or` receive both arguments as values, so the operator values `(&&)` and `(||)` are strict; only the infixed forms `e1 && e2` and `e1 || e2`, which are derived forms, evaluate their second operand conditionally.

The basic value `append` concatenates strings and lists; for strings, $s\,s'$ is the sequence of the characters of s followed by those of s':

$$\begin{aligned}
\text{APPLY}(\texttt{append}, s, s') &= \{s\,s'\}\\
\text{APPLY}(\texttt{append}, \texttt{[]}, v) &= \{v\}\\
\text{APPLY}(\texttt{append}, v_1 \texttt{ :: } v_2, v) &= \{v_1 \texttt{ :: } v'\}
\end{aligned}$$

where, in the last line, $\text{APPLY}(\texttt{append}, v_2, v) = \{v'\}$. The composition and application operators and `identity`, `always` and `never` are given by their texts (Figure 48). No value has the type `Never`, so `never` is never applied.

D.4.8 Debugging

$$\begin{aligned}
\text{APPLY}(\texttt{Debug.log}, s, v) &= \{v\}\\
\text{APPLY}(\texttt{Debug.todo}, s) &= \emptyset\\
\text{APPLY}(\texttt{Debug.toString}, v) &= \{s_v\}
\end{aligned}$$

where s_v is the string that the implementation gives for v. Thus `Debug.log` returns its second argument; any output it makes lies outside the definition. `Debug.todo` has no result, so an application of it has no value (Section 6.5). The string of `Debug.toString` is determined by its argument, in a way that is implementation-dependent (Section E.6).

D.4.9 Arrays

We take the class Con to contain, besides `[]` and `::`, a third constructor name array that no identifier spells, with arity(array) = 1. The values of the type `Array` are the constructor values array v with v a list: the array of the elements of v, in order. The two basic values that Chapter 8 applies to arrays are

$$\begin{aligned}
\text{APPLY}(\texttt{Array.fromList}, v) &= \{\text{array } v\},\\
\text{APPLY}(\texttt{Array.toList}, \text{array } v) &= \{v\}.
\end{aligned}$$

Two arrays are equal when their lists are, by the fourth clause of Figure 54. The other functions of the library module `Array` lie outside the definition (Appendix C).

D.4.10 Bytes

We take the class Con to contain also a fourth constructor name binary that no identifier spells, with arity(binary) = 1. The values of the type `Bytes` are the constructor values binary v with v a list of Int values each from 0 to 255; the value binary $[o_1, \ldots, o_k]$ is the finite sequence of the bytes $o_1, \ldots, o_k$, in order. Two values of type `Bytes` are equal exactly when their sequences are, by the fourth clause of Figure 54: equality on them is structural, like that on lists. The values of `Bytes` meet the definition only at ports and in flags, where Chapter 8 relates them to the byte sequences of the boundary (Section 8.7); the functions of the library `elm/bytes` lie outside the definition (Appendix C).

D.4.11 Effect Values

For the basic values of `Task`, `Process`, `Platform.Cmd`, `Platform.Sub` and `Platform`, and for `Task.command`, APPLY and the arities are given in Figure 24 (Section 8.6); for the values of ports, outport(p, τ) and inport(p, τ), in Section 8.7.

Appendix E

Implementation-Dependent Details

Where implementations of Standard Elm legitimately differ, the body of the definition states only what every implementation must satisfy, using an abstract parameter where one is needed. This appendix gives, for each such parameter or unspecified point, what the body requires of it and what two implementations choose. The implementations are Elm 0.19.1 with the elm/core 1.0.5 kernel on JavaScript, written *Elm/JS*, and eco 0.2.0 producing native code, written *eco*; eco's output for JavaScript behaves as Elm/JS does, except for `Bytes` at the boundary (Section E.8.5). Each section states the requirement, with a reference to the body, and then gives a table with one row per implementation: its choice, and the consequences for programs. An implementation *conforms* on a point when its choice meets the requirement. The requirements concern the programs to which the definition gives a value. Where a program elaborates and the definition gives it a value, a conforming implementation's result is one of its values; where the definition gives it none (the evaluation of its modules reaches `Debug.todo`, or does not end), the program has no meaning, and the definition requires nothing of an implementation, which may stop, run for ever, or run the program to some result. So Elm/JS, for which `(&&) False (Debug.todo "x")` is `False` where the definition gives no value, conforms on a program whose modules evaluate it. A program that has a meaning is held to its traces (Section 9.5). Where a step of it has no result, as when an application of `update` reaches `Debug.todo`, no further message is handled (rule (183)), and an implementation that handles further messages does not conform. Section E.10 collects every known point on which an implementation does not conform, with the ruling that settles it. The facts recorded here describe the versions named; they are not part of the definition.

E.1 The Int Range and Normalisation

The body requires the following of the type `Int` (Chapter 2, Section 6.3 and Section D.4).

- IntRange is a set of integers containing every integer n with $-2^{31} \leq n \leq 2^{31} - 1$; Int has one zero.
- norm maps every integer into IntRange and is the identity on IntRange. Int addition, subtraction, multiplication, negation, absolute value and power with a non-negative exponent give norm of the exact result, and an integer constant spelling n denotes norm(n).
- For $b \neq 0$, `a // b` is norm of the truncation of a/b toward zero; `x // 0`, `modBy 0 x` and `remainderBy 0 x` are 0.
- For $e < 0$, `b ^ e` at Int is 1 when $b = 1$, 1 or -1 by the parity of e when $b = -1$, and 0 otherwise.
- `round`, `floor`, `ceiling` and `truncate` give the integer they denote when it lies in IntRange, the least or greatest member of IntRange when it lies below or above it (an infinity included), and 0 for NaN.

A value of type `Int` is always a member of IntRange: never a fraction, NaN, an infinity or a negative zero.

E.1.1 The Range and Normalisation

Impl.	Choice	Consequences for programs
Elm/JS	An Int is an IEEE 754 double. `+`, `-` and `*` give the exact result rounded to the nearest double: exact while its magnitude is at most 2^{53}, rounded above that, an infinity beyond the largest double. An integer constant is reduced modulo 2^{64} into $[-2^{63}, 2^{63})$ as two's complement.	No IntRange and norm describe Elm/JS: Int results above 2^{53} in magnitude are inexact, an overflow gives an infinity, and other operations below leave the integers. Programs whose Int results stay within $\pm 2^{53}$ compute exactly.
eco	IntRange is $[-2^{63}, 2^{63} - 1]$, and norm(n) is the m in IntRange with $m \equiv n \pmod{2^{64}}$ (64-bit two's complement wrap). Integer constants likewise: `0xFFFFFFFFFFFFFFFF` denotes -1.	Conforms. Overflow wraps silently: the greatest Int plus 1 is the least; `abs` and `negate` of the least Int give it back.

E.1.2 Integer Division and Remainders

Impl.	Choice	Consequences for programs
Elm/JS	`a // b` is the quotient a/b converted to a 32-bit integer (truncation, then reduction modulo 2^{32} into $[-2^{31}, 2^{31})$; NaN and the infinities give 0), so `x // 0` is 0. `modBy 0 x` stops the program with the message "Cannot perform mod 0. Division by zero error." `remainderBy 0 x` is NaN, held as an Int.	Quotients outside $\pm 2^{31}$ wrap: `5000000000 // 1` is `705032704`. `modBy 0` ends the program; NaN from `remainderBy 0` spreads through later arithmetic (`remainderBy 0 7 + 1` is NaN).
eco	Truncated division on 64 bits; `x // 0`, `modBy 0 x` and `remainderBy 0 x` are 0; the least Int `// -1` is the least Int.	Conforms.

E.1.3 Powers with a Negative Exponent

Impl.	Choice	Consequences for programs
Elm/JS	The real power, held as an Int: at Int, `2 ^ -1` is `0.5` and `0 ^ -1` is an infinity.	Does not conform: the result is not an integer.
eco	0 for every base, `1` and `-1` included. For $e \geq 0$, the power reduced modulo 2^{64}, as norm.	Does not conform for the bases 1 and -1: at Int, `1 ^ -1` is 0, not 1.

E.1.4 Conversion from Float

Impl.	Choice	Consequences for programs
Elm/JS	`round`, `floor` and `ceiling` of NaN or an infinity give NaN or that infinity, held as an Int. `truncate` converts to a 32-bit integer (NaN and the infinities give 0).	Does not conform: `round (0/0)` is NaN, and `truncate 3e10` is `-64771072`.
eco	Saturating conversion to 64 bits: NaN gives 0; values above or below IntRange, infinities included, give its greatest or least member.	Conforms: `truncate 3e10` is `30000000000`, and `round (1/0)` and `truncate 1e20` are `9223372036854775807`.

E.1.5 Negative Zero

Impl.	Choice	Consequences for programs
Elm/JS	`negate 0`, `0 * -1`, `modBy 4 -4`, `remainderBy 4 -4`, `round -0.4` and `ceiling -0.5` give a negative zero held as an Int.	Does not conform. The sign shows after conversion: `1 / toFloat (negate 0)` is `-Infinity`. Equality does not see it.
eco	Int has one zero.	Conforms: `1 / toFloat (negate 0)` is `Infinity`.

In Elm/JS a value of type `Int` may thus be a fraction, NaN, an infinity or a negative zero, none of which is a member of any IntRange; Elm/JS does not conform at the extremes of Int. A program whose exact Int results, quotients and conversions from Float all lie between -2^{31} and $2^{31} - 1$, which applies neither `modBy 0`, `remainderBy 0` nor Int `^` with a negative exponent, and which leaves no variable of class `number` to the default of Float (Section E.6.3), has the same meaning in every implementation listed here, apart from the sign of a zero seen through `toFloat`.

E.2 Characters and Strings

The body requires the following (Chapter 2, Section 6.3 and Section D.4).

- `Char` is an abstract class of characters with an injective function `toCode` into the Int range, under which the character with code point h has the code h, for every Unicode scalar value h; a string is a finite sequence of characters. `Char.fromCode n` is the character with code point n when n is a Unicode scalar value, and otherwise a character of the implementation's choosing, which has a code like every other.

- The escape `\u{`h`}` denotes a Unicode scalar value: surrogate code points are excluded.

- Chars are ordered by `toCode`; strings are ordered lexicographically by their sequences of characters.

- The letters of identifiers are classified by the general categories of a version of the Unicode standard chosen by the implementation.

- Case mapping is outside the initial basis and has no requirement.

E.2.1 What a Character Is

Impl.	Choice	Consequences for programs
Elm/JS	A Char is a JavaScript string of one code point, one or two UTF-16 code units; case mapping can make it longer. `toCode` gives its first code point, and NaN for a lone high surrogate. `fromCode n` gives U+FFFD for $n < 0$ or $n > 10FFFF_{16}$, and passes surrogates through. A string is a sequence of UTF-16 code units. A surrogate escape is accepted.	`toCode` is not injective on the Chars Elm/JS can make: the two-character result of `Char.toUpper 'ß'` has the code of `'S'`. A string holding a lone surrogate makes `String.toList` run until memory is exhausted.
eco	A Char is one code point in 0 to $10FFFF_{16}$; a surrogate code point is a Char of its own. `toCode` is the code point; `fromCode` as Elm/JS. A string is a sequence of UTF-16 code units: a valid surrogate pair is one character, a lone surrogate is one character. A surrogate escape is accepted.	`toCode` is injective. `String.fromList (String.toList s)` is `s` for every string.

In both, `String.length` and string indices count UTF-16 code units, so a character above U+FFFF counts twice: the length of a three-character string whose middle character is U+1F600 is 4.

E.2.2 The Order of Characters and Strings

Impl.	Choice	Consequences for programs
Elm/JS	Chars and strings both by UTF-16 code units.	The two orders agree with each other but not with `toCode`: a character above U+FFFF sorts below the characters U+E000 to U+FFFF. Does not conform.
eco	Chars by code point; strings by UTF-16 code units.	Char order conforms; string order does not, for strings comparing a character above U+FFFF with one in U+E000 to U+FFFF. Char order and the order of one-character strings then disagree.

E.2.3 The Unicode Version of Identifiers

Impl.	Choice	Consequences for programs
Elm/JS	The Unicode tables of the Haskell system that built the compiler.	A letter assigned in a later Unicode version than one implementation knows is not a letter there: an identifier using it is accepted by one implementation and rejected by the other.
eco	Unicode 14.0.0.	

E.2.4 Case Mapping

Impl.	Choice	Consequences for programs
Elm/JS	Full case mapping of the host's Unicode version, with the final sigma rule; `toLocaleUpper` and `toLocaleLower` use the host's locale in a browser.	`Char.toUpper 'ß'` is the two-character `"SS"`, held as a Char. `String.toLower` lowers a word-final capital sigma to final sigma.
eco	Simple case mapping of Unicode 17.0, one code point to one; no final sigma rule; `toLocaleUpper` and `toLocaleLower` are `toUpper` and `toLower`.	`Char.toUpper 'ß'` is `'ß'`; case mapping keeps the length of a string.

The predicates of the module `Char` (`isUpper`, `isLower`, `isAlpha`, `isDigit`, `isOctDigit` and `isHexDigit`) are, in both, defined by their elm/core text over ASCII code ranges, and agree.

E.3 Floating-Point Operations

The body requires the following of the type `Float` (Section 6.3 and Section D.4).

- `compare` on Floats is a total order: -0 and $+0$ are equal, NaN equals NaN and is greater than every other Float.
- At Float, `<`, `<=`, `>` and `>=` are the IEEE predicates, false when an operand is NaN; on tuples and lists they are the lexicographic order derived from `compare`.
- At Float, `min` and `max` are the *minimum* and *maximum* of IEEE 754-2019: NaN when either argument is NaN, and -0 below $+0$, whatever the order of the arguments. At every other comparable type they give the results of their elm/core definitions.
- At Float, `clamp lo hi x` is `min (max x lo) hi`, with the `min` and `max` above: hi when $lo > hi$, NaN when any argument is NaN, and `clamp`

`0 1 -0` is +0. At Int it is given by its elm/core definition through `<` and `>`: *lo* when $x < lo$, else *hi* when $x > hi$, else x, so that `clamp 10 0 5` is 10.

- At Float, `+`, `-`, `*`, `/`, `sqrt` and `negate` are the operations of IEEE 754-2019, correctly rounded, with no latitude in accuracy (Figure 50).

- The power and the transcendental functions (the trigonometric functions, their inverses, `atan2` and the logarithm under `logBase`) denote the real function rounded to a Float, with special values as IEEE 754-2019 recommends; their accuracy is implementation-dependent.

In both implementations a Float is an IEEE 754 binary64 number, and `+`, `-`, `*`, `/` and `sqrt` are the IEEE operations rounding to nearest.

E.3.1 `compare` with NaN

Impl.	Choice	Consequences for programs
Elm/JS	`LT` if $a < b$, `EQ` if a and b are equal, else `GT`: any comparison with NaN is `GT`.	Does not conform: `compare 1 nan` and `compare nan 1` are both `GT`. Sorting, `Dict` and `Set` with NaN keys have no defined result.
eco	The total order of the body.	Conforms: `compare 1 nan` is `LT` and `compare nan nan` is `EQ`; NaNs sort last and are one key of a `Dict` or `Set`.

E.3.2 The Order Operators

Impl.	Choice	Consequences for programs
Elm/JS	The IEEE predicate when an operand is written as a constant; otherwise derived from `compare` above.	Does not conform: with `one = 1`, `one > nan` is `True` while `1 > nan` is `False`, and `nan >= nan` is `True` when neither operand is a constant.
eco	IEEE predicates at Float on every path; on tuples and lists, by `compare`.	Conforms: `nan < 1` is `False`, and `[ nan ] > [ 1 ]` is `True`.

E.3.3 `min`, `max` and `clamp`

Impl.	Choice	Consequences for programs
Elm/JS	The elm/core definitions, at every type, through the comparison of `compare` above, which makes any comparison with NaN `GT` and the two zeros `EQ`.	`min` and `max` do not conform at Float when an argument is NaN or both are zeros: the result depends on the order of the arguments. `min nan 1` and `max 1 nan` are `1`, where the body gives NaN. `clamp` conforms at Int; at Float it does not conform when $lo > hi$, when an argument is NaN, or with zeros of different signs: `clamp 10 0 5` is `10`, `clamp 0 1 nan` is `1`, `clamp nan 1 0.5` is `0.5` and `clamp 0 1 -0` is -0, where the body gives $+0$, NaN, NaN and $+0$.
eco	At Float, IEEE 754-2019 *minimum* and *maximum* (NaN in either argument gives NaN; $-0 < +0$), and `clamp lo hi x` is `min (max x lo) hi`. At every other type, the elm/core definitions.	`min`, `max` and `clamp` conform.

E.3.4 Special Values and Accuracy

Impl.	Choice	Consequences for programs
Elm/JS	The host's ECMAScript `Math` functions, whose accuracy the host chooses. `^` at Float follows ECMAScript: `1 ^ (1/0)` is NaN.	Does not conform for the special values of `^`: IEEE 754-2019 gives 1. `logBase 10 1000` is `2.9999999999999996`.
eco	`^` at Float follows C99 Annex F (`1 ^ (1/0)` is 1); `sin` and `cos` are those of the code generator, `tan x` is computed as `sin x / cos x`, and the inverse trigonometric functions are those of the C mathematics library.	Accuracy is that of the functions named. The special values of `^` conform, but no test of eco checks them.

E.4 Equality

The body requires that equality be structural, with IEEE equality at Float; the identity of values plays no part, so a value containing NaN is not equal to itself. When an equality test reaches two functions, its result is an unspecified Bool: APPLY gives the set of both Bools (Section 6.5, rule (84) and Section D.4).

E.4.1 Identity

Impl.	Choice	Consequences for programs
Elm/JS	Structural, but `True` at once when both operands are the same object. At Float, `x == 0` with `0` written as an Int constant tests whether `x` is zero or NaN.	Does not conform: `p == p` with `p = ( nan, 1 )` is `True`, and `nan == 0` is `True` (`nan /= 0` is `False`).
eco	Structural, but its code gives `True` at once when both operands are the same object, in the equality kernel and in the inline fast path; whether two occurrences are the same object depends on optimisation. IEEE at Float on every path.	Does not conform for values containing NaN: by the code, `p == p` as above is `True`, while a separately built `( nan, 1 )` is unequal to `p`. Two of eco's tests of known Elm bugs expect `False` for `p == p` and for other containers compared with themselves, as the definition gives, but eco's test harness does not run them.

E.4.2 Functions

Impl.	Choice	Consequences for programs
Elm/JS	`True` for the same object; otherwise the program stops with the message "Trying to use `(==)` on functions".	Does not conform: `(\x -> x) == (\x -> x + 0)` and `Just f == Just g` end the program.
eco	`True` if the two are the same object, else `False`, at any depth. A lambda that captures no variable is one object however often it is evaluated; a partial application is a new object each time.	Conforms: the result is one member of the set. `f == f` is `True`; `(\x -> x) == (\x -> x + 0)` is `False`.

E.5 Bitwise Operations and String Conversion

The functions of the module `Bitwise` are outside the initial basis (Chapter C), and so are the conversions `String.toInt`, `String.fromInt` and `String.fromFloat`; the body places no requirement on them. They are recorded here because they reveal the Int range.

E.5.1 Bitwise Operations

Impl.	Choice	Consequences for programs
Elm/JS	32 bits: operands converted to 32-bit integers, results signed 32-bit, shift counts taken modulo 32; `shiftRightZfBy` gives an unsigned 32-bit result.	`shiftLeftBy 32 1` is `1`; `shiftRightZfBy 0 -1` is `4294967295`.
eco	64 bits, signed; shift counts taken modulo 64; `shiftRightZfBy` gives its 64-bit result read as signed.	`shiftLeftBy 32 1` is `4294967296`, `shiftLeftBy 63 1` is the least Int, and `shiftRightZfBy 0 -1` is `-1`.

E.5.2 Conversion between Numbers and Strings

Impl.	Choice	Consequences for programs
Elm/JS	`String.toInt` reads an optional sign and decimal digits into a double, with no overflow check and no length limit; `"-0"` gives a negative zero. `String.fromInt` and `String.fromFloat` use the host's printing of numbers.	Long numerals give rounded results rather than `Nothing`. `String.fromInt` prints 10^{21} as `"1e+21"`, and an Int holding `0.5` as `"0.5"`.
eco	`String.toInt` gives `Nothing` when the numeral is outside IntRange or longer than 64 code units, and the exact value otherwise. `String.fromInt` prints the exact decimal. `String.fromFloat` prints the shortest digits that read back to the same Float: fixed notation for magnitudes from 10^{-6} up to 10^{21}, exponent notation otherwise, and `"0"` for -0.	`String.toInt` and `String.fromInt` are exact over IntRange.

E.6 Evaluation Order and Debug

The body requires the following (Section 6.7, Section 6.5 and Chapter 7).

- The order in which the parts of a phrase are evaluated carries no meaning; the groups of a `let` are evaluated in an order consistent with their dependencies, and the order among independent bindings is not specified.

- Each top-level value is evaluated once, when its module is initialised, in dependency order; an implementation may evaluate it later or repeatedly provided the results agree.

- `Debug.log l v` evaluates to v, and any output it makes is outside the definition; `Debug.todo s` has no value; `Debug.toString` gives an implementation-dependent String.

Since evaluation has no other effect, the choices below are observable only through the output of `Debug.log`, through which of several subexpressions without a value is reached first, and through resources.

E.6.1 The Order of Evaluation within a Phrase

Impl.	Choice	Consequences for programs
Elm/JS	`let` bindings by dependency groups, dependencies first. Record fields by field name; tuple and list components, and the function and arguments of an application, left to right.	Conforms.
eco	`let` bindings by dependency groups, dependencies first; named bindings before bindings `_ = e`, and not in source order among themselves; a run of bindings `_ = e` in source order. Record fields of type Int, Float and Char first, then the rest, each by field name. Tuple and list components and arguments in no fixed order. The callback of a basis function such as `List.map` may be applied in another order when it makes no use of `Debug`.	Conforms. The output of several `Debug.log` calls in one `let` may appear in a different order from Elm/JS.

E.6.2 Top-level Values

Impl.	Choice	Consequences for programs
Elm/JS	Each once, at start-up, before `main` runs, in dependency order.	Conforms. A top-level value whose evaluation reaches `Debug.todo` stops the program at start-up, used or not.
eco	On first use. A value of type Int, Float or Char is computed again at each use; a top-level value of function type evaluates its body again at each call; any other value is computed once.	Conforms. A `Debug.log` in a top-level value may print once per use instead of once.

In both, a `main` in a module other than the main module is evaluated once, and a top-level value needed while it is itself being evaluated, which has no value, ends the program: by exhausting the stack, or, in Elm/JS compiled for debugging, with a report of infinite recursion.

E.6.3 Number Specialisation

The body requires the following (Section 4.10 and Section 6.2).

- A type variable of class `number` that nothing fixes is `Float`. An implementation that fixes the types of numbers before the program runs meets this by closing each such variable to `Float`; nothing more need be evaluated.

- A declaration whose type scheme quantifies a variable of class `number` has its default copy, at `Float`, and a copy for each other type at which a phrase uses it, whether or not that phrase is evaluated. Each copy of a top-level declaration is evaluated when its module is, and each copy of a `let` declaration when the `let` is.

A copy that the program does not need, and that has no value, therefore leaves the program without one when it is evaluated with its module, and the definition then requires nothing of an implementation (the beginning of this appendix); evaluated in a step of a running program, such as an application of `update`, it leaves that step without a result, which a conforming implementation must respect.

Impl.	Choice	Consequences for programs
Elm/JS	One representation for Int and Float, which behaves as Float. A value is evaluated once, with that representation.	Conforms, except in a step of a running program where a copy that is not needed has no value: Elm/JS carries on (Section E.10). A computation with a `number` that nothing fixes gives the value of the default copy.
eco	Monomorphises: a definition is copied only for the types at which the program uses it, and a `number` variable that nothing fixes is closed to `Int`. A copy of a top-level value is evaluated on first use (Section E.6); the copies of a `let` declaration are bindings of the `let`, evaluated when it is.	Does not conform for a `number` that nothing fixes: `2 ^ -1 == 0` is `True`, where the definition gives `False`. Closing such variables to `Float` instead would conform. Conforms on copies.

On copies, the implementations differ only where a copy that the program does not need has no value. With `check h = if h == 0 then Debug.todo "h" else h` and `x = check (2 ^ -1)` used as `x + 0.5` and, in a branch that is not taken, as `x // 1`, the copy of `x` at Int exists and has no value, since `2 ^ -1` is 0 there:

`x` is	Definition	Elm/JS	eco
top-level	the copy at Int is evaluated with the module: no value, so no meaning	runs: `x` is 0.5 once, and `x + 0.5` is 1	runs: the copy at Int is never used, so never evaluated
bound by a `let` evaluated with the module	the copy at Int is evaluated with the `let`: no value, so no meaning	runs, as above	stops at the `Debug.todo`, as the definition's evaluation would
bound by a `let` in `update`	the copy at Int is evaluated with the `let`: the step has no result, and no further message is handled	handles further messages: does not conform	stops handling messages: conforms

In the first two rows the program has no meaning, and both implementations conform. Where only the copy at Float is used, no copy at Int is made, and the program has a value in the definition and in both implementations.

E.6.4 Debug

Impl.	Choice	Consequences for programs
Elm/JS	`Debug.log l v` writes `l:` followed by `Debug.toString v` to the host's console. `Debug.todo s` stops the program with "TODO in module … on line …" and `s`. A program using `Debug` is rejected when compiled with `--optimize`.	Conforms. A program using `Debug` cannot be compiled for production.
eco	`Debug.log l v` writes `l:` followed by `Debug.toString v` to the program's output when it is evaluated. `Debug.todo s` writes "Debug.todo: " and `s` to the error output and ends the process with status 1. `Debug` is accepted with `--optimize`, and its output kept.	Conforms.

`Debug.toString` gives the same text in both for the forms tested, such as `{ age = 3, name = "x" }`, `Just -1` and `0` for -0; for Floats it follows `String.fromFloat` (Section E.5).

E.7 Resource Limits

The body places no bound on the size of a phrase (Section 2.10) or on the resources evaluation uses; an implementation that stops for lack of resources is outside the definition (Section 6.7).

E.7.1 The Size of Phrases

Impl.	Choice	Consequences for programs
Elm/JS	No limit.	
eco	On every target, a program is rejected that has a function or lambda with more than 2047 parameters, a lambda whose parameters and captured variables together exceed 2047, a record type or record with more than 2047 fields, a constructor with more than 2040 fields, or a nullary constructor that is not among the first 1024 constructors of its type. For native code the closure limit is checked again after each polymorphic function is copied for every type at which it is used.	Programs beyond these limits are rejected. A closure within the limit in the source can exceed it in native code: 700 local polymorphic functions each used at three types inside one lambda make 2100 captured variables.

E.7.2 Stack and Memory

Impl.	Choice	Consequences for programs
Elm/JS	The host's stack and memory. Deep recursion that is not a self tail call stops with a `RangeError` the host can catch, at about ten thousand frames. A `Cmd.batch` nested 100000 deep exhausts the stack.	Recursion over a long list must be a self tail call, which runs in constant stack.
eco	The program runs on a stack of 64 MiB. Exhausting it ends the process with a segmentation fault; exhausting memory ends it with the message "eco: out of memory" and an abort. Deeply nested `Cmd.batch` values are handled without recursion.	Far deeper recursion succeeds than in Elm/JS, but its failure cannot be caught. Self tail calls run in constant stack.

E.8 Platform and Ports

The body defines tasks, commands, subscriptions and the encoding of values at ports in Chapter 8, and the execution of a program in Section 9.4. It requires only that messages be handled one at a time, each update seeing the model the previous one produced, and the effects of an earlier update being dispatched before those of a later one. It leaves the following unspecified or weakly specified.

- The order in which the commands of one batch are performed, and the relative order of effects handled by different managers.

- An incoming port value is an event, handled after the step in progress completes, never during it.
- Outgoing port values are delivered after the step that produced them, in order per port; what the host does with them is outside the definition.
- An incoming value that does not match the port's type is rejected at the boundary; the program is unaffected.
- Absent flags are the JSON value null.
- `Bytes` is a payload type in every position, in ports and in flags, and its values cross the boundary as byte sequences, which are not JSON values; how a host represents a byte sequence is outside the definition (Section 8.7).
- JSON numbers are finite decimal rationals. A Float that is NaN or an infinity is encoded as `null`, and -0 as the number 0; an incoming number decodes at `Int` exactly when it is an integer in IntRange, an incoming zero being the one Int zero, and at `Float` as the nearest float, $+0$ for a zero and an infinity for a number beyond the finite floats (Section 8.7).
- JSON objects are unordered, and error messages are unspecified.
- A spawned process that fails ends, and nothing else happens; killing a finished process has no effect; the task `Process.sleep d` waits at least $\max(0, d)$ milliseconds, and for non-finite d the delay is unspecified.
- The set of managers, beyond the task manager and ports, is a parameter of the platform.

E.8.1 The Order of Commands

Impl.	Choice	Consequences for programs
Elm/JS	A manager receives the commands of a batch in reverse order: `Cmd.batch [ perform a, perform b ]` runs `b` first, and `Cmd.batch [ out 1, out 2 ]` delivers `2` then `1`. Several subscriptions to one incoming port receive a value last first. Managers are served in the order the program declares them and its ports.	Conforms. Programs that depend on these orders, or on the order of port declarations, behave differently in eco.
eco	The same reverse order within a batch and among subscriptions to one port. Managers are served in no fixed order.	Conforms.

E.8.2 Incoming Port Values

Impl.	Choice	Consequences for programs
Elm/JS	The host's `send` decodes the value and runs `update` for each current subscription before it returns. A value that does not decode makes `send` throw "Trying to send an unexpected type of value through port" to the host; the program is unaffected.	Does not conform for delivery: a value sent from inside an outgoing subscriber is handled during the step in progress, and a reply can be lost because the subscription for it is not yet installed (elm/core issues 1110, 1141, 1154). Conforms for mismatches.
eco	The value is checked when the host sends it, then queued, and delivered after the work in progress; a send from inside a subscriber is queued too. A mismatch is reported to the host by an error code and the same message, and the program is unaffected; but when the port's type cannot be checked at the send, the value is checked on delivery, and a mismatch then ends the process.	Conforms for delivery. Does not conform for a mismatch found on delivery.

E.8.3 Outgoing Port Values

Impl.	Choice	Consequences for programs
Elm/JS	Delivered later: a port waits one timer tick before its first delivery, so that the host can subscribe after the program starts, and after each batch. A value sent when the port has no subscriber is discarded.	Conforms.
eco	Delivered at once, during the dispatch of the update that issued it; the program waits while subscribers run. Values are kept until the port's first subscriber and then handed to it in order; once a port has had a subscriber, a value sent when it has none is discarded.	Conforms.

E.8.4 Values at the Boundary

Impl.	Choice	Consequences for programs
Elm/JS	Host values are JavaScript values. An incoming Int is any finite number with no fractional part. A `Json.Encode.Value` passes any host value through, functions and `undefined` included. Objects are written with their keys in field-name order; of several missing fields, the last by name is reported.	Outgoing Ints above 2^{53} are inexact. NaN and the infinities pass as Floats in both directions. Does not conform: an outgoing NaN or infinity is delivered as a host number, where the definition delivers `null`; an outgoing -0 is delivered as -0; an incoming -0 at an `Int` port gives a negative zero, and at a `Float` port -0, where the definition gives 0 and $+0$. An incoming NaN or infinity is a host value outside JSON, and outside the definition.
eco	Host values are JSON text, byte sequences apart (Section E.8.5). An incoming Int is an integer, or a float with an integral value, in IntRange. An outgoing NaN or infinity is written as `null`. Keys in field-name order; missing fields are reported in the order of the type's fields.	`9223372036854775808` and `1e300` are rejected as Ints. A host value that is not JSON cannot be sent. Conforms for NaN and the infinities.

E.8.5 Bytes at the Boundary

The body admits `Bytes` wherever a payload type may appear, alone or inside tuples, closed records, `Maybe`, `List` and `Array`, in the types of ports and in the flags of `main` (Section 5.4); its values cross as byte sequences (Section 8.7). This rule follows the implementations that provide binary data at the boundary. Each implementation chooses the host representation of a byte sequence; in the table, *eco/JS* is eco's output for JavaScript.

Impl.	Choice	Consequences for programs
Elm/JS	No representation: a port or flags type in which `Bytes` occurs is rejected.	Does not conform: every program with `Bytes` in a payload or flags type is rejected.
eco/JS	`Bytes` in any payload position, in ports and in flags; a byte sequence is a `Uint8Array` at the host.	Conforms.
eco	`Bytes` only as the whole payload of a port, a byte sequence being raw bytes at the host; `Bytes` inside a payload type, and in flags, is rejected.	Does not conform: programs with `Bytes` nested in a payload type, or in flags, are rejected.

E.8.6 Flags

Impl.	Choice	Consequences for programs
Elm/JS	Absent flags are the JavaScript value `undefined`, which no decoder for `null` or `Maybe` accepts. Flags that do not decode make `init` throw "Problem with the flags given to your Elm program on initialization."	Does not conform: a program with flags of type `Maybe Int` started without flags does not start.
eco	Absent flags are `null`. Flags that do not decode stop an executable with a message and an abort; an embedding host receives an error code and may try again.	Conforms: the program above starts with `Nothing`.

E.8.7 Processes

Impl.	Choice	Consequences for programs
Elm/JS	A failing spawned process ends silently. Killing a finished or already killed process throws a `TypeError`. `Process.sleep` uses the host's timers: a delay of 0, a negative delay, a non-finite delay or one above $2^{31} - 1$ ms is about 1 ms.	Killing twice ends the program (elm/core issue 955): does not conform. A delay above $2^{31} - 1$ ms, about 24.8 days, is cut short: does not conform.
eco	A failing spawned process ends with a warning on the error output. Killing a finished process has no effect. A delay of 0 or less resumes at once; a non-finite delay, or one above $2^{31} - 1$ ms, is 1 ms.	The warning is outside the definition: conforms. A delay above $2^{31} - 1$ ms is cut short: does not conform.

In both, a process runs until it waits for a timer, a message or another primitive task, and is never suspended at an `andThen` alone; this is one of the behaviours the body allows.

E.8.8 Managers and Programs on a Platform

Impl.	Choice	Consequences for programs
Elm/JS	Every effect module the program uses is a manager: among them `Task`, `Time`, `Http`, `Random`, `Browser.Events` and `Browser.AnimationManager`. Browser programs run their `init` and draw their view; a `main` of type `Html msg` is drawn once. Several programs may be started on one page.	Every command and subscription of these managers is performed.
eco	In native code the managers are `Task`, `Time`, `Http`, those of eco's system package, and ports. A command or subscription for any other manager, such as `Random.generate` or `Browser.Events`, is discarded. `Browser.element`, `document` and `application` return their record without running `init`; `Browser.Navigation` ends the program with status 1; `Browser.Dom` tasks fail with `NotFound`. A `main` of type `Html msg` is evaluated and not drawn. One program per process.	Only worker programs, `Platform.worker`, run natively. Programs relying on `Random` or browser managers do nothing observable in those effects.

E.9 Extensions of eco

The body admits no flags type beyond the port payload types (Section 5.4); `Bytes` is one of them, and is not an extension (Section E.8.5). eco accepts the following programs, which are not Standard Elm.

Impl.	Choice	Consequences for programs
Elm/JS	None.	
eco	A program of type `Program System.Environment model msg`, with `System.Environment` from eco's system package, has its flags decoded by a built-in decoder, with no check of the payload type.	Programs using this extension are eco programs, not Standard Elm, and are rejected by Elm 0.19.1.

E.10 Known Divergences

The tables below list every known point on which an implementation does not conform: the implementation (Elm/JS, eco, or both), the construct, what

the implementation does, what the definition says, and the ruling that settles the point. Where a ruling chose a meaning neither implementation has, both are listed.

E.10.1 Int Arithmetic

Impl.	Construct	Implementation	Definition	Ruling
Elm/JS	Int `+ - *` beyond $\pm 2^{53}$	rounded to a double; an infinity beyond the largest double	norm of the exact result	A40
Elm/JS	integer constant at or above 2^{63}	reduced modulo 2^{64}, then held as a double	norm(n)	A9
Elm/JS	`a // b`, quotient outside $\pm 2^{31}$	reduced to 32 bits	norm of the truncated quotient	A41
Elm/JS	`modBy 0 x`	program stops	0	A42
Elm/JS	`remainderBy 0 x`	NaN held as an Int	0	A42
Elm/JS	Int `b ^ e`, $e < 0$	a fraction or an infinity	1, ± 1 or 0	A43
eco	Int `b ^ e`, $e < 0$, $b = \pm 1$	0	1, or ± 1 by parity	A43
Elm/JS	`round`, `floor`, `ceiling` of NaN or an infinity	NaN or the infinity held as an Int	0, or a bound of IntRange	A44
Elm/JS	`truncate` outside $\pm 2^{31}$	reduced to 32 bits	saturated to IntRange	A44
Elm/JS	`negate 0` and other Int zeros	a negative zero	one zero	A45

E.10.2 Float, Order and Equality

Impl.	Construct	Implementation	Definition	Ruling
Elm/JS	`compare` with NaN	always `GT`	total order, NaN greatest	A46
Elm/JS	`<`, `<=`, `>`, `>=` with NaN	depends on whether an operand is a constant	IEEE predicates	A47
Elm/JS	Float `min`, `max` with NaN or two zeros	depends on the order of the arguments: `min nan 1` and `max 1 nan` are `1`	IEEE 754-2019 minimum and maximum: NaN for both	A48
Elm/JS	Float `clamp` with $lo > hi$, a NaN argument, or zeros of different signs	elm/core definition: `clamp 10 0 5` is `10`, `clamp 0 1 nan` is `1`, `clamp nan 1 0.5` is `0.5`, `clamp 0 1 -0` is -0	`min (max x lo) hi` with IEEE 754-2019 minimum and maximum: +0, NaN, NaN, +0	A47, A48, A49
Elm/JS	Float `1 ^ (1/0)`	NaN	1	A65
both	`v == v` with v containing NaN	`True` when both operands are one object	`False`	A50
Elm/JS	Float `x == 0`, `0` an Int constant	`True` for NaN	`False` for NaN	A50
Elm/JS	`==` reaching two functions	program stops	an unspecified Bool	A51

E.10.3 Characters and Strings

Impl.	Construct	Implementation	Definition	Ruling
both	`\u{D800}` and other surrogate escapes	accepted	not an escape sequence	A8
Elm/JS	`toCode` of a lone surrogate; Chars from case mapping	NaN; a Char of two characters	`toCode` injective into the integers	A52
Elm/JS	Char order	by UTF-16 code units	by `toCode`	A53
both	String order	by UTF-16 code units	lexicographic by characters ordered by `toCode`	A53

E.10.4 Evaluation

Impl.	Construct	Implementation	Definition	Ruling
Elm/JS	`(&&) a b`, `(||) a b` applied in prefix form, in a step of a running program	second argument evaluated only when needed: `(&&) False (Debug.todo "x")` is `False`, and further messages are handled	ordinary strict functions: the step has no result, and no further message is handled	A56, A94
Elm/JS	lambdas made in a self tail-recursive function	share the function's changing parameters: three closures give `[3,3,3]`	each holds the values of its own call: `[3,2,1]`	A63
Elm/JS	a `let` declaration in `update` whose copy at Int, not needed, has no value	evaluated once, as Float; further messages are handled	the copy at Int is evaluated: the step has no result, and no further message is handled	A87, A94
eco	a `number` variable that nothing fixes, such as that of `2 ^ -1 == 0`	Int: `True`	Float: `False`, as Elm/JS gives	A27

E.10.5 Programs and Ports

Impl.	Construct	Implementation	Definition	Ruling
Elm/JS	incoming port value	`update` runs inside the host's `send`	an event handled after the step in progress	A68
eco	incoming value not matching the port, found on delivery	the process ends	rejected at the boundary; program unaffected	A71
Elm/JS	absent flags	`undefined`: `Maybe` flags rejected	`null`	A72
Elm/JS	outgoing Float NaN or infinity	passed as a host number	`null`	A88
Elm/JS	-0 across a port	passed as -0; an incoming -0 at `Int` is a negative zero	the number 0; 0 at `Int`, $+0$ at `Float`	A88
Elm/JS	`Process.kill` of a finished process	program stops	no effect	A80
both	`Process.sleep` `d`, $d > 2^{31} - 1$	about 1 ms	at least d ms	A81
both	a `main` of unsupported type outside the main module	rejected	an ordinary value	A32, A82
both	two ports of one name, or a port named after a manager	accepted; the program stops at start-up	rejected by elaboration	A84
Elm/JS	`Bytes` in a port or flags type	rejected	a payload type, crossing as a byte sequence	A92
eco	`Bytes` inside a payload type, or in flags, native code	rejected	a payload type, crossing as a byte sequence	A92

E.10.6 Elaboration

Impl.	Construct	Implementation	Definition	Ruling
both	a pattern `( )` with a gap	accepted as unit	not a phrase	A1
both	a rigid type variable escaping its annotation	the compiler fails with an internal error	does not elaborate	A20
eco	a record extended through a row variable by a field it already has	accepted, the outer field winning	does not elaborate	A23
Elm/JS	unannotated mutually recursive values used at several types	accepted; some such programs fail at run time	each member monomorphic within its group	A24
eco	two literal patterns that denote one value, spelt differently	second not reported redundant	rejected as redundant	A31
eco	a constrained type variable as a row variable, `{ number | x : Int }`	annotation accepted; every use fails	ill formed	A36
eco	polymorphic recursion through annotated functions, native code	rejected	elaborates	A39
eco	a closure over 2047 variables after copying polymorphic functions, native code	rejected	elaborates	A39
eco	phrases beyond the size limits of Section E.7	rejected	elaborate	A13, A34
both	`import` *modid* `as Basics` for a module other than Basics	accepted (not tested)	does not elaborate	A91

Appendix F

Elm and Standard ML

This appendix compares Standard Elm, as this document defines it, with Standard ML as defined in *The Definition of Standard ML (Revised)* by Milner, Tofte, Harper and MacQueen (MIT Press, 1997). It defines nothing; each entry refers to the place in this document where the construct is defined.

F.1 What Is the Same

The method of this definition is that of the Definition of Standard ML. The language is presented as a grid of phases (syntax, static semantics, dynamic semantics) against levels; each semantics section defines its semantic objects before its rules; the rules infer sentences of the form background, phrase, meaning, one rule group per phrase class, numbered through the whole book and followed by Comments; and the rules are given for a bare language, into which every derived form is translated. Much of the Core coincides as well: type names generated by type declarations, type functions, type schemes and their instances, closure of a type with respect to a context, contexts and environments, function closures whose recursion is expressed by the operation Rec , matches whose failure is the object FAIL, value constructors with their identifier status, and evaluation by value. The differences lie in the effects the dynamic semantics must account for, in the types of records, tuples and overloaded operations, in the modules, and in the syntax, and are listed below.

F.2 Differences by Construct

The differences are shown construct by construct in Figures 56 (evaluation), 57 and 58 (types), 59 (modules, programs and effects) and 60 (syntax). In the figures, "Ch.", "Sec.", "Fig." and "App." refer to chapters, sections, figures and appendices of this document.

Feature	Standard ML	Elm (this definition)	Where defined here
Exceptions	`exception` declarations, `raise` and `handle`; exception names and packets; an exception convention for the dynamic rules.	None: no exception names, no packets, no exception convention. Evaluation of a phrase either gives a value or gives none; absence and failure are values of the types `Maybe` and `Result` of the initial basis.	Ch. 6, Sec. 6.7; Sec. C.4
Incomplete matches	Allowed; a match that fails raises `Match`, a value binding that fails raises `Bind`.	Not allowed: every match, lambda pattern and destructuring binding must be exhaustive, and every match rule must be needed, at the types of the principal derivation.	Sec. 4.11; Sec. 4.10
State	References (`ref`, `:=`, `!`), addresses and states; a state convention, under which the order of premises is the order of evaluation.	None: sentences carry no state, and no state convention is needed. The order of premises has no meaning; the order of evaluation is observable only through `Debug`, and is left to the implementation.	Sec. 6.7; Sec. E.6
Arithmetic failure	`Overflow` is raised when a result leaves the range of `int`; `Div` when a divisor is zero.	Total: every Int result is normalised into the implementation's range, and an operation with no correct result gives a defined one; thus `3 // 0` is `0`.	Ch. D; Sec. E.1
Equality	Equality types: type variables `''a`, an equality attribute on type names; `=` at a function type does not elaborate.	No equality types: `==` elaborates at every type. When it reaches two functions its result is an unspecified Bool, so evaluation may be non-deterministic.	Sec. 4.4; Sec. D.4; Sec. E.4

Figure 56: Standard ML and Elm: Evaluation

Feature	**Standard ML**	**Elm (this definition)**	**Where defined here**
Overloading	A side mechanism (Appendix E of that definition): overloaded identifiers such as `+` and `<`, and overloaded special constants, resolved by context, with `int` as default.	Part of the type system: type variables constrained to the classes number, comparable, appendable and compappend, generalised and instantiated like others. A number variable that the principal derivation leaves undetermined is taken to be Float before evaluation.	Sec. 4.4; Sec. 4.10; Sec. 6.2
Records	Record types are closed. The selector `#`*lab* and the record pattern wildcard `...` require the record type to be determined by the context. No record update.	Record types may be extensible, by a row variable (`{ r | x : Int }`). Field selection, the field accessor `.`*lab* and record patterns apply to every record with the fields named. Record update `{ r | x = e }` keeps the type of the record. A pattern row *lab* stands for *lab* `=` *var*, as in SML.	Sec. 4.2; Sec. 4.9; App. A
Tuples	A tuple is a record with labels `1`, ..., n ($n \geq 2$); the unit is the empty record.	Tuples of two and three components have type names of their own, and are not records; the unit is a type of its own.	Fig. 5; Sec. 4.2
Datatypes	`datatype` with `withtype`, datatype replication, `abstype`; datatypes may be declared locally.	Custom types are declared only at the top level of a module. All custom types of a module form one group, and they and the type aliases of the module may mention one another; there is no `withtype`, replication or `abstype`. A record alias also declares a constructor.	Fig. 10; Sec. A.2; Sec. 5.5

Figure 57: Standard ML and Elm: Types

Feature	**Standard ML**	**Elm (this definition)**	**Where defined here**
Value polymorphism	A value restriction: only a non-expansive expression has its type generalised.	No value restriction: closure applies to every binding, whatever its expression.	Sec. 4.8; Sec. 4.9
Explicit type variables	Bound at a value declaration, explicitly (`val 'a`) or implicitly at the outermost value declaration in which they occur. Type constraints *exp* `:` *ty* on any expression or pattern.	No binder: the type variables of an annotation scope over the annotations nested in its definition, and may not escape it. Annotations are given only for definitions and ports.	Sec. 4.7

Figure 58: Standard ML and Elm: Polymorphism and Type Variables

Standard ML constructs without a counterpart here are the sequential expression, `while`, `local`, `open`, `abstype` and the declarations of exceptions; the nearest to `open` is the exposing list of an import (Section 3.5). Elm constructs without a counterpart in Standard ML are record update and extensible record types, the field accessor, the constrained classes of type variables, and ports.

Feature	**Standard ML**	**Elm (this definition)**	**Where defined here**
Modules	Structures, signatures and functors; sharing; transparent and opaque signature matching, with realisation and enrichment.	Flat modules: a module is a header, imports and top-level declarations, and exports top-level names by an exposing list; a custom type is exposed with or without its constructors. No signatures, functors or sharing; the import relation between modules has no cycle.	Ch. 3; Ch. 5, Sec. 5.2; Ch. 7
Programs	An interactive top level: a sequence of top-level declarations, each elaborated and evaluated in the basis left by those before it; a declaration that fails is rejected and the session continues.	A program is a main module and the modules it imports, elaborated as a whole; its value `main` is executed by a platform, as a labelled transition system whose observable behaviour is its traffic on ports.	Ch. 9, Sec. 9.4
Effects	At the level of the language, only state and exceptions; input and output are left to library functions with side effects.	Evaluation has no effect. Tasks, commands and subscriptions are values, built by functions of the basis, which the platform carries out; their results return to the program as messages.	Ch. 8, Sec. 8.4, Sec. 8.5
Ports	None.	A `port` declaration in a port module gives a command for sending values out or a subscription for receiving them. Only values of port types may cross a port, encoded as JSON values in which byte sequences may stand.	Sec. 5.4; Sec. 8.7

Figure 59: Standard ML and Elm: Modules, Programs and Effects

Feature	**Standard ML**	**Elm (this definition)**	**Where defined here**
Infixed operators	`infix`, `infixr` and `nonfix` declarations in programs, scoped like other declarations; `op` removes infix status.	The fixities are fixed by the initial basis and by library modules; no program declares an operator. Some operators are non-associative. The operator value (*op*) uses an operator as a function.	Sec. 2.7, Fig. 3
Layout	Insensitive to layout; phrases are delimited by reserved words such as `end` and by `;`.	Sensitive to indentation: match rules, the definitions of a `let` and the declarations of a module are separated by layout, and `;` and `and` are abstract syntax.	Sec. 2.6; Sec. 3.4
Recursion	`val rec` and `fun` with `and` group recursive bindings as written; each declaration is in scope only after it.	The definitions of a `let` and the declarations of a module stand in any order and are grouped, by derived forms, into components of the "refers to" relation in dependency order. In a `let`, recursion passes only through definitions with parameters.	Sec. A.1; Sec. A.2; Sec. 2.10
Functions	`fn` takes a match; `fun` defines a function by several clauses.	A lambda expression takes atomic patterns and has one body; a function binding has one clause; several cases are written with `case`.	Fig. 5; App. A

Figure 60: Standard ML and Elm: Syntax