Commentary on Standard ML

Robin Milner
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, Scotland

Mads Tofte
Department of Computer Science
University of Nigeria
Nsukka
Nigeria

May 2, 1991*

*I generated this file on October 12, 2002, from a back-up dated May 2, 1991, since no original dvi file was available in the back-up. I had to change two occurrences of `>From` to `From` in the source files (apparently corruption caused by a mail program). Furthermore, I had to edit the tex files to force page breaks on pages 43, 99, 100, 111, 120, 130, 137 and 138 to be as in the printed version. The page breaks in the Index differ from the printed version, but the number of pages is the same as in the printed version. (Mads Tofte, October 12, 2002)

Contents

Preface

In this book we seek to explain in depth the meaning of a programming language. We present Standard ML by describing the objects in terms of which a programmer thinks when he is building a program, and how these objects work together; in more formal terms, we discuss the semantic theory of the language. The full mathematical definition is given in *The Definition of Standard ML*[1], and we have written this Commentary as a companion to the Definition.

In this preface we explain the complementary rôle played by the Commentary and the Definition, and we suggest how the reader may use them together to attain a complete grasp of the language. In this task one needs formal definition; one also needs informal discussion of the intuitions and motivations which underlie the design. Our hope is that the two books together provide a mode of study which is more effective than either book alone could provide, and even more effective than a single text which interleaves the definition and discussion.

For readers who know little of Standard ML we begin with a brief character sketch of the language.

Standard ML in outline

Standard ML has evolved from use in a demanding research environment, to become a serious candidate for use in large and varied applications. It is primarily a *functional* language, in the sense that it gives full power to treat functions as objects, but it is also equipped with *imperative* power and with an exception mechanism. As far as functions are concerned, recent technical advances in compiler techniques for functions – notably in pattern-matching, which is the workhorse of function application – has made functional languages a serious competitor to procedural languages in efficiency. The design of patterns for function-call in ML takes full advantage of this technology and allows succinct expression. On the other hand, in many applications one naturally deals with objects whose state is both complex and dynamically changing; therefore ML combines expressive imperative features with functions.

To assist program organisation, the language gives the power to define parameterised recursive data types, and it provides a rigorous but convenient polymorphic type discipline which allows the same function to be used over a wide range of types. This discipline was first presented in ML over a decade ago, and has become widely adopted in other languages. The most innovative part of the language consists of an advanced form of parametric module, for the purpose of organised development of large programs. This feature, more than any other, has given rise to new semantic techniques in the Definition; it is also the aspect of the language

[1] by R.Milner, M.Tofte and R.Harper, MIT Press, 1990. It will henceforth be referred to as "the Definition", distinguished by an initial capital letter.

which receives most discussion in this Commentary. We believe that a deep understanding of the semantic issues of parametric modules is an essential step towards the robust design of large systems whose parts can be re-used.

Perhaps the most prominent feature of the language Definition is the sharp distinction between two phases of the execution of a program. First, there is the *static* phase, in which one checks that a program is sound before it is run; then there is the *dynamic* phase in which the program can be run without any dynamic checks for soundness, since this has been fully checked in the static phase. This distinction is of course not new; but in ML it is represented in an advanced form for the polymorphic type discipline and the module discipline. Furthermore, the semantic Definition reflects the distinction faithfully.

Need for a Commentary

The purpose of the Definition was to define exactly what Standard ML is, and what it means. The emphasis was on precision and brevity. The document is almost one big definition, at the technical level that one finds in mathematics. Indeed, whereas mathematical authors normally single out their definitions from their narrative text, we tended to single out comments from the defining text, since the latter so much dominates the former. Moreover, to keep the Definition within reasonable bounds we often omitted theorems which would justify the definitions. Not surprisingly, anybody who sets out to read the Definition from cover to cover finds it tough going.

This kind of language definition, which tells the *what* more than the *why*, can give a distorted impression of the language as being determined beyond argument. Moreover, the use of mathematical notation can give a false air of authority; a reader may say "Well, I don't really understand what is going on here, but someone probably has a good reason for it". (A less charitable reader may say something stronger.) It is true that if a community of people wish to use the same programming language with the same meaning, then a precise definition is a *necessity* – and to be precise it must be in mathematical notation. But, although *necessary*, it is not *sufficient* to gain acceptance by the community. For this acceptance, it is also necessary to achieve a reasonably high level of communal understanding. Thus the main purpose of doing formal semantics, which is to gain greater understanding, has also a solid pragmatic justification. Moreover, formal notations are just the vehicle for approaching this understanding, and formal definitions are just the starting point.

Early in the process of writing the Definition, we decided that to interleave more discussion and theoretical development with the primary material – which exactly describes the execution of ML programs – would lead to an unwieldy text, and that cross-reference between the primary material and the theoretical development would be much easier with two texts than with one; hence this Commentary. We hope thus to make both texts accessible to readers with different purposes:

- Implementers, who wish to stay faithful to the Definition;
- Programmers, familiar to some extent with ML but wishing to deepen their understanding;
- Teachers of courses on programming languages;
- Researchers into programming language design and semantics.

Aims of the Commentary

By writing the Commentary, we intend first to make the Definition easier to understand, and to use. We argued in the preface to the Definition that the understanding of a language must be in terms of the various semantic objects – things like structure, type, reference, exception, channel – which people have in mind when they discuss the language, and whose names often creep into the syntax of the language as keywords. In the Definition we do say what these objects are – i.e. we define them – but we devote only a little space to developing the theory which shows how they hang together, and which thereby illumines this "world of meanings" in which the language finds expression. So in the Commentary we have given much more space to this theory.

We insist that the theory be mathematical. In the past it has not been common, when describing a *particular* programming language, to explain things like type-checking or dangling references or determinacy by means of theorems, though of course there are *general* theories of some such things. But weaknesses in language design, or incompatibilities between different implementations of the same language, have often remained unnoticed or become accepted as part of the landscape ("not a bug but a feature"); we believe one reason for this is that they have not been placed in the searchlight of mathematical analysis. Indeed our own experience gives evidence for this; we had to correct the Definition several times before it reached its published form, due to failures to prove the properties which we wished it to have. Our Commentary, therefore, is not only the place to provide insight *via* discussion; it is also the place to present some of the rigorous analysis which supports the Definition. We have therefore included a selection of theorems which express important properties of the language.

Besides clarifying the Definition in the sense of explaining *what* ML is, we also try in this Commentary to explain *why* it is so. Frequently in the work leading up to the Definition, false steps were taken and retracted; or restrictions were found necessary; or plausible alternative paths were seen to lead to trouble; or other such paths were not seen to lead to to trouble but still not taken. The reasons for particular decisions can therefore vary both in nature and in strength; in important cases, we try in the Commentary to indicate what the reasons were. We also point out some cases where the decision was not fully determined; that is, cases where we cannot argue conclusively against an alternative.

By this kind of clarification, the Commentary aims to become a practical working document in several senses, which correlate quite well with the different kinds of reader that we mentioned above. First, everyone with an interest in ML, or in languages with similar features, can use it to gain understanding; in particular, they can also use it to settle fine points of interpretation. Second, with the benefit of insight rather than just dry specification, implementers will find it easier to design efficient ways of achieving the specified meaning. (An example of this is provided by the notion of *well-formedness* [Sec 5.3, p 32];[2] the Definition demands that semantic objects must be well-formed for an elaboration to be admissible, and by understanding elaborations in detail an implementer can discover exactly how seldom well-formedness needs to be checked.)

But perhaps most importantly, the Commentary and Definition together aim to provide a working platform for future development of ML, and even for other languages too. Nothing remains fixed in language development; but these documents provide a fairly deep analysis of one design choice, and we hope that the influence of this will be – paradoxically perhaps – both stabilising and liberating. The stabilising influence should be that, if anyone proposes a change or extension, then the community is unwilling to accept it unless its semantic effect is fully analysed in terms of the framework which the Definition provides. But what about the liberating influence? Well, the semantic theory – when properly understood – does indeed provide a way to go about assessing any proposed extension. One can see clearly whether the extension is as general as it could be; one can test its soundness (for example: how many of the theorems proved in the Commentary remain true in the presence of the extension?); in many ways one can replace inconclusive argument with conclusive analysis, and this leads to firmer agreement to accept an extension.

Instances of this kind of investigation are already occurring. One example is to do with the `abstraction` declaration which was part of David MacQueen's original proposal for Modules. Rightly or wrongly, this form of declaration is not included in the Definition. But with the help of the semantic theory several things can be immediately seen about it; in particular, that there are alternative programming styles which can achieve the effect of `abstraction` but perhaps not so conveniently, and that there is at least one natural generalisation of the notion which may be preferable.[3] Therefore, even if it turns out that `abstraction` itself is indeed the preferred extension, the choice will have been made on the basis of much firmer understanding than if it had been included at the outset.

[2]When we refer to the Definition we shall do so using square brackets, e.g. [Sec 4.8, p 21] for "Section 4.8, page 21" or [App B, Fig 22, p 73] for "Appendix B, Figure 22, page 73". References like Chapter 3, or Section 6.1, or page 45, or Theorem 3.4 (or combinations of such) refer to the Commentary itself. But inference rules in the Definition will be referred to as e.g. rule 57, without square brackets.

[3]See the discussion in Section 8.4.

The link with research

We finish this discussion by reflecting briefly on the way in which the specialised theory of ML can contribute to broader research on language design and semantics. Here, we are aware of a conflict of purpose. In this Commentary we are trying to enlighten a real working language, and such languages tend to combine features which, though each theoretically tractable in isolation, become less so when combined. (Example: the combination of polymorphic types with exception-handling; see Section 4.5.) On the one hand, therefore, we have to juggle with *many interactions at once*, because we insist on providing an instance of a full-size language whose design has been, in a sense, mathematically ratified. On the other hand, in scientific development one normally studies *one feature at a time*, or at most *one interaction at a time*, not all at once; this is the only way to isolate the source of each difficulty.

The conflict is this: If we study one interaction at a time we may succeed, but we may have scant evidence as to which interactions – which combinations of language features – are the right ones to study; we therefore lack directed motivation. If on the other hand we deal with all features at once, in order to test in the field the effect of combining these features in a mathematically ratified language, then we may simply fail; either the analysis is too large to do, or when done it is too large to be generally intelligible.

We claim that this conflict has been resolved in the case of ML; the language is large enough to be a working language, but just small enough to allow us to succeed to a large extent in ratifying it mathematically. This achievement is relevant to general language research in more than one way. First, by observing the experience with ML in the field, we can try to assess how much the user community gains from the fact that the design of a language has been mathematically tested. (Among these benefits, we hope, is a much smaller variation in behaviour between different implementations.) This approach – treating the language and its use as an experiment to learn from – should help us to assess the *value* of semantic research and semantically based design.

But we also suggest that the *content* of research in semantics and language design can be guided by this kind of case-study in analysing a working language, even though the study itself may be a bit too large for a theoretician's comfort. In fact, from the study one hopes to abstract results, principles and methods which can be applied generally, not just to a single language. (A simple example is Theorem 2.4, which asserts that evaluation is determinate.) Furthermore, as the language develops new semantic problems will arise. These problems can be formulated within the present semantic framework; then it will be easy to abstract them into a simpler theoretical framework (e.g. the semantics of a much reduced language) where their solution can first be attempted.

We can already give two examples where this process should be fruitful. One is to do with adding the evaluation function `eval`, as in LISP, and with introducing

dynamically-typed values (values whose types can be analysed at run-time).[4] The other is the question of adding higher-order functors to ML; see Section 8.5. In both these cases it is easy to come up with design proposals and tentative semantic definitions. One can then study their soundness in an abstract setting, and finally try to show that the proof of their soundness scales up to the full language.

How to read the Commentary

We imagine that most readers will not be already familiar with the Definition. So the immediate question is how to tackle both texts; which should come first, or should they be interleaved in some way? (We are assuming at least some familiarity with ML, there are good textbooks on how to program in the language.)

The first thing to do is to get a perspective by reading the preface and introduction to the Definition, as well as the present preface. After that, Chapter 1 is designed with some care to lead you into the semantic method, and to give some feel for the *whole* topic without dwelling on subtleties. If you are unfamiliar with using inference trees to represent execution of programs, then it will be a very good investment not only to *read* Chapter 1 but also to build some inference trees as suggested in the exercises there. At that point, little mystery should remain about the way programs are executed at the top level, as defined in [Sec 8, p 63].

Beyond this point, a good strategy is to let each Commentary chapter direct you to the relevant sections and subsections of the Definition; the Commentary tries to maintain a decent pedagogic sequence, and while following that sequence it should become clear when to devote solid attention – as opposed to just making reference – to a part of the Definition. Some paragraphs are in small print; these can safely be omitted on a first reading.

In deciding your path though the Commentary, first note that the static and dynamic semantics can be studied independently of one another. We dealt with the *static* semantics first in the Definition [Sec 4,5], and have emphasised the independence by dealing with the *dynamic* semantics first in the Commentary. Thus you can start either with Chapters 2 and 3 on dynamic semantics, or with Chapters 4–8 on the static semantics. As far as static semantics is concerned, Chapters 4 and 5 deal with the Core Language, while Chapters 6, 7 and 8 give a good overview for the Modules.

The final three chapters, Chapters 9–11, are concerned with the fine detail of signatures and signature matching in ML. Appendix A contains the complete proof of the most important theorem which underlies the use of signatures in ML.

In Appendix B we supply some rules to determine the class to which each occurrence of an identifier belongs (this the only place where we discuss the syntax of ML). Solutions to *all* exercises are in Appendix C. In Appendix D we list the

[4] A recent paper *Dynamic typing in a statically-typed language*, by M.Abadi *et al*, Proceedings of 16th ACM POPL Conference, gives a good lead in this direction.

mistakes and ambiguities in the Definition. Last, but not least, we have compiled a detailed index.

Acknowledgements

First we wish to acknowledge the pioneering work of David MacQueen, in his original design of ML Modules; thereby we also acknowledge the work of Rod Burstall and his group in Edinburgh at the beginning of the 1980s, from which these design ideas were distilled. We have found much satisfaction in refining these ideas, we hope without distortion, into a completely detailed language design and specialised theory.

We have special debts to Simon Finn and to Don Sannella. Simon Finn pointed out to us in mid-1989 a few important problems about signatures. Due to his insight we were able to correct the treatment of principal signatures, and were prompted to do the full proof of this correctness, before the Definition was published. Don Sannella has been a creative and detailed critic, most especially in his thorough reading of the whole of this text; many important improvements of presentation are due to him.

We would like to thank Marie Virginia Aponte, Dave Berry, Mike Fourman, Renaud Marlet, David Matthews, Nick Rothwell and David Turner for their reading of parts of this manuscript and for their helpful comments on it, or for their contribution through conversations about the language.

We pay tribute to Donald Knuth and Leslie Lamport for inventing TeX and LaTeX, in which all our text preparation has been done – including diagrams – both here and in the Definition. For really complex text of this kind, the elimination of the whole phase of type-setting and subsequent proof-correction has been of great value.

Mads Tofte wishes to thank the University of Nigeria for granting him leave of absence to complete the book, and the Laboratory for Foundations of Computer Science at the University of Edinburgh for all the support they have given. Robin Milner wishes to thank the University of Edinburgh for granting him sabbatical leave to write the book, and the Computer Laboratory at the University of Cambridge for their hospitality while much of the writing was done. We both acknowledge financial support from the Science and Engineering Research Council for the Standard ML project, of which this Commentary is a part.

Last but not least we want to thank our wives, Lucy and Joan, for their continual encouragement and tolerance.

Robin Milner
Mads Tofte
University of Edinburgh, July 1990

1 Executing a simple program

There are two kinds of difficulty in mastering the ML Definition; the understanding of new concepts, and the management of detail. To begin with, we should like to help readers with the second difficulty. By following the execution of a simple program we hope to introduce them at the same time to the structure both of the language and of the Definition document.

Here is the program:

```
structure ARITH =
   struct
      datatype NAT = Zero | Succ of NAT
      fun twice(Zero) = Zero
        | twice(Succ x) = Succ(Succ(twice x))
   end ;

open ARITH ;

val two = Succ(Succ Zero) ;

twice two ;
```

1.1 Execution

In the Introduction [Sec 1, p 1] we learn that there are three phases in execution: *parsing*, *elaboration* (type- and structure-checking), and *evaluation*. We also learn that [Sec 8] deals with all three of these at the level of programs. In [Sec 8, p 63] we discover that a program is a sequence of top-level declarations, so that our program is of the form

$$program_1 \;=\; topdec_1 \;;\; topdec_2 \;;\; topdec_3 \;;\; topdec_4 \;;$$

where

$$\begin{aligned} topdec_1 &= \texttt{structure} \cdots \\ topdec_2 &= \texttt{open} \cdots \\ topdec_3 &= \texttt{val} \cdots \\ topdec_4 &= \texttt{twice two} \end{aligned}$$

We also learn the following:

- The execution of a program is expressed as a sentence of the form

$$s, B \vdash program \Rightarrow B', s'$$

where B is the basis – all the declared information – and s the state *before* execution, while B' and s' are the basis and the state *after* execution. (We shall not be concerned with the state in this example.)

- A basis B is a pair $B_{\mathrm{STAT}}, B_{\mathrm{DYN}}$ – a *static* and a *dynamic* basis. The static basis contains all type and structure information provided by previous declarations and is relevant to future elaboration; the dynamic basis contains all associations of identifiers with values, resulting from previous declarations, and is relevant to future evaluation.

- Rule 196 [p 64] tells us that, since our program takes the form

$$\mathit{program}_1 \;=\; \mathit{topdec}_1 \;;\; \mathit{program}_2$$

its execution should normally consist of the following parts:

$$\begin{array}{ll} B_{\mathrm{STAT}} \vdash_{\mathrm{STAT}} \mathit{topdec}_1 \Rightarrow B'_{\mathrm{STAT}} & -\text{the elaboration of } \mathit{topdec}_1; \\ s, B_{\mathrm{DYN}} \vdash_{\mathrm{DYN}} \mathit{topdec}_1 \Rightarrow B'_{\mathrm{DYN}}, s' & -\text{the evaluation of } \mathit{topdec}_1; \\ s', B \oplus B' \vdash \mathit{program}_2 \Rightarrow B'', s'' & -\text{the execution of } \mathit{program}_2. \end{array}$$

(However, abnormal execution of topdec_1 would be dealt with by rule 194 or 195.)

This is our first visit to an inference rule, and we shall now explain rules and their use. Every one of the 196 rules in the Definition is in fact a schema, containing meta-variables which range over either syntactic objects (e.g. *topdec*) or semantic objects (e.g. B_{DYN} or s'). Execution is built out of *instances* of rules. As another example, leaping from the very top to the very bottom of the language, in rule 1 [p 23]

$$\frac{}{C \vdash \;\mathit{scon} \Rightarrow \mathrm{type}(\mathit{scon})}$$

C ranges over contexts (a kind of cut-down basis, containing various sets and environment components), while *scon* ranges over special constants. A typical instance of the rule is

$$\frac{}{(\emptyset, \emptyset, (\{\}, \{\}, \{\}, \{\})) \vdash \;\texttt{3.1415} \Rightarrow \texttt{real}}$$

where C has been instantiated to the simplest context there is, with all components empty.[5]

Rules are built from two ingredients; *sentences* of the general form

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

[5] `3.1415` is a phrase in the language, i.e. a numeral as opposed to a number; `real` is a semantic object, a so-called *type name*.

where A and A' are semantic objects and *phrase* is a phrase of ML, and *side-conditions*. The *conclusion* of the rule (below the line) must be a sentence; each *hypothesis* (above the line) may be either a sentence or a side-condition. In a sentence, the turnstile ($\vdash$) and the arrow ($\Rightarrow$) have intuitive significance; as is common in formal logic, we put semantic objects that can be thought of as "given" on the left-hand side of the turnstile, while the arrow ($\Rightarrow$) suggests that execution is a process which results in the object on the right-hand side of the arrow. Note that rule 1 has no premises; on the other hand rule 196 has three premises which are sentences, and one which is a side-condition. More precisely, rule 196 really consists of two rules, as explained in the remark about options [p 23] – one in which everything between angle brackets $\langle\ \rangle$ is excluded, and one in which it is all included.

We shall use the term *inference* to mean a rule instance which contains no meta-variables, and whose side-conditions are true; side-conditions are no longer part of an inference. By fitting inferences on top of each other so that every premise is the conclusion of some other inference we can form *inference trees*. Thus an inference tree represents the elaboration and/or evaluation of a specific phrase. When it uses only the *static* rules 1–102, we call it an *elaboration* (*tree*); when it uses only the *dynamic* rules 103–193, we call it an *evaluation* (*tree*). We shall see examples of both, later in this section.

There are two ways of interpreting a sentence of the form $A \vdash \mathit{phrase} \Rightarrow A'$. In the *cumulative* interpretation, A' is the result of modifying A by *phrase*, typically by adding bindings to A. In the *incremental* interpretation, A' is the semantic representation of *phrase* when elaborated (or evaluated) in A. In many cases it is obvious which interpretation is intended; for example, if A and A' are not the same kind of object, $A \vdash \mathit{phrase} \Rightarrow A'$ must be interpreted incrementally. Also all declarations, including top-level declarations, are interpreted incrementally. Thus the elaboration of

```
val a = 3 val x = "Monday"
```

results in the environment $E = \{\mathtt{a} \mapsto \mathtt{int}, \mathtt{x} \mapsto \mathtt{string}\}$, regardless of the basis in which the elaboration takes place. As described in Section 4.1, the incremental interpretation is ideal when bindings are to have local scope; the cumulative interpretation is not appropriate in this case, as it does not distinguish between those parts of A' that stem from A and those parts that stem from *phrase*. The cumulative interpretation is only used in the rules for programs, and in the dynamic semantics, where the evolution of the state is cumulative although environments are formed incrementally.

Let us return to rule 196; what does $\oplus$ mean? $B \oplus B'$ is the result of superimposing the information in B' upon that in B; for example, if B' contains information about a variable *var* or a type *ty*, then any information about *var* or *ty* in B is overwritten. Note that the result of elaborating or evaluating a *topdec* is an incremental basis B'_{STAT} or B'_{DYN}, and rule 196 superimposes (using $\oplus$) this increment

upon the total basis. $B \oplus B'$ can also be expressed $B_{\mathrm{STAT}} \oplus B'_{\mathrm{STAT}}, B_{\mathrm{DYN}} \oplus B'_{\mathrm{DYN}}$; in fact static and dynamic information accumulate separately, and this allows us to treat elaboration and evaluation completely independently below the level of programs. The only way in which they interact is enshrined in rules 194 and 195. Rule 194 says that if a *topdec* fails to elaborate then its evaluation is skipped, while rule 195 says that if a *topdec* evaluates to an exception packet p – i.e. if it raises an exception – then its elaboration is forgotten.

For this reason, and because all the *topdec*s in our program will execute normally, we can conveniently concentrate first upon the elaboration of our *whole* program, and then upon its evaluation.

1.2 Elaboration

We shall now drop the subscript STAT, understanding that we are dealing only with a *static* basis and with *elaboration*. So we look for elaborations of our four *topdec*s:

$$\begin{aligned} & B_1 \vdash \mathit{topdec}_1 \Rightarrow B'_1 \\ B_2 = B_1 \oplus B'_1, \quad & B_2 \vdash \mathit{topdec}_2 \Rightarrow B'_2 \\ B_3 = B_2 \oplus B'_2, \quad & B_3 \vdash \mathit{topdec}_3 \Rightarrow B'_3 \\ B_4 = B_3 \oplus B'_3, \quad & B_4 \vdash \mathit{topdec}_4 \Rightarrow B'_4 \end{aligned}$$

Visiting the rules for the semantics of Modules [Sec 5], we find that the elaboration of topdec_1 must be inferred by the inference tree shown in Figure 1, which is decorated with the numbers of the rules used. In this inference tree, and often in later ones, side-conditions are omitted (e.g. the side-condition on rule 100); but remember that the inference is not valid unless side-conditions are satisfied. The root (conclusion) of the tree is at the bottom; we have omitted the Core part of the elaboration at the top.

One can understand elaboration better, perhaps, as the *process* of generating such a tree. In this process one does not simply start at the root and work upwards, or start at the leaves and work downwards. Instead, roughly speaking, one proceeds from the root upwards filling in the *left-hand* part of each sentence (up to $\Rightarrow$); then upon reaching each leaf one can proceed downwards filling in the *right-hand* parts, the results of elaboration.

To understand the inference tree in detail, we must consider the form of the various semantic objects, with the help of [Fig 11, Sec 5, p 31]. Of the four components N, F, G, E of a (static) basis B, only N and E need concern us in the example (F and G record functor and signature information). In fact, in the basis B'_1 to which topdec_1 elaborates, only these two components – the name set names(E') and the environment E' – are non-empty (the injection function "in Basis" pads out the basis B'_1 with empty functor and signature environments); here E' is the environment to which the *strdec*

```
structure ARITH = ···
```

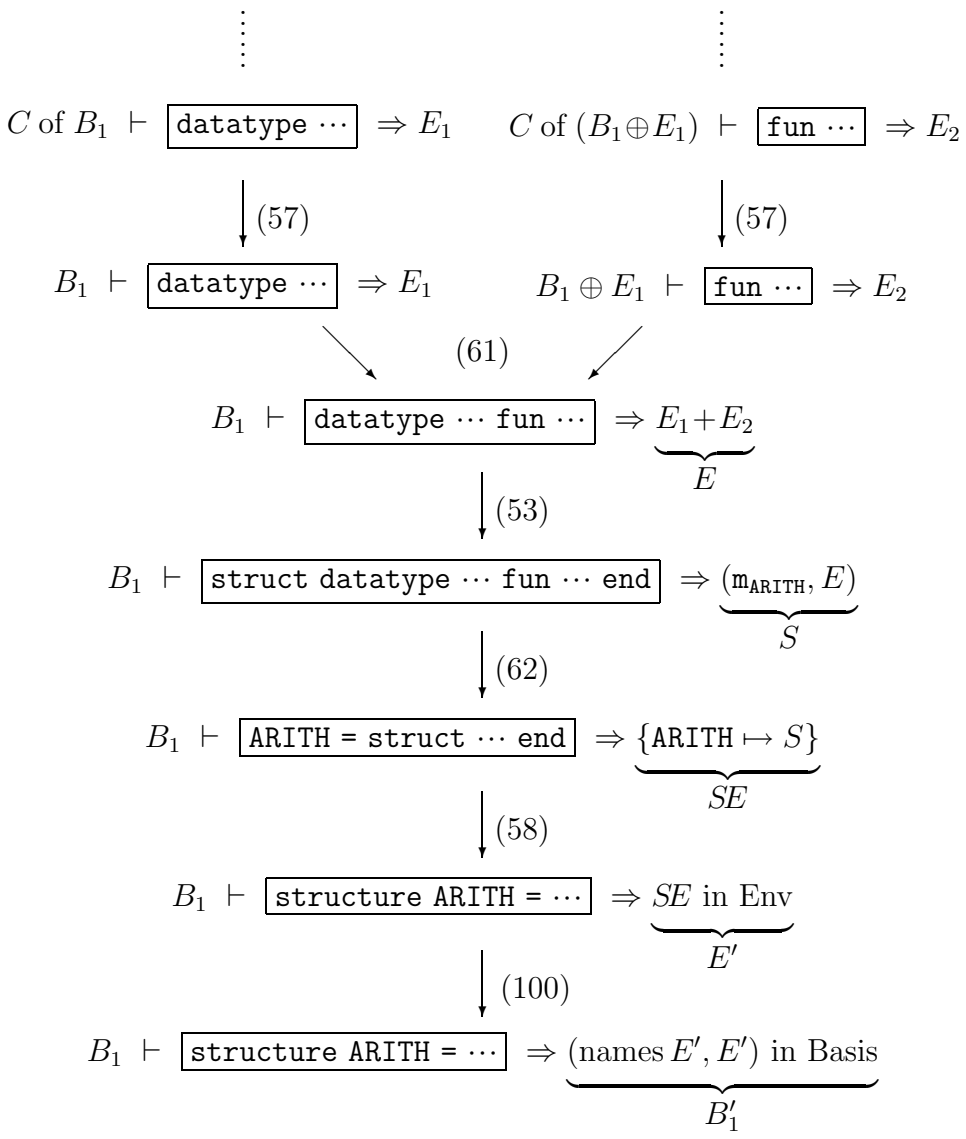

Figure 1: Part of the elaboration of $topdec_1$

elaborates, and names E' records the structure names and type names generated by this *strdec*.

Proceeding up the tree, we now need to learn from [Fig 10, Sec 4, p 17] that the four components SE, TE, VE, EE of an environment E contain information about structures, types, variables and exceptions respectively. Here, $E' = SE, \{\}, \{\}, \{\}$, where SE is the structure environment to which our *strbind*

$$\texttt{ARITH = struct} \cdots \texttt{end}$$

elaborates. Now in general a structure environment is a finite map from structure identifiers to structures, and here it is a singleton; it maps only `ARITH` to the structure S to which our *strexp*

$$\texttt{struct datatype} \cdots \texttt{fun} \cdots \texttt{end}$$

elaborates. And what is a structure? Just an environment paired with a unique structure name; so $S = (\mathtt{m}_{\mathtt{ARITH}}, E)$, where $E = E_1 + E_2$ is the superposition of the two environments to which our two *dec*s elaborate. (A *dec* is a Core language declaration.)

The Modules part of the inference tree is complete, and the Core part is supplied by the rules for the static semantics for the Core [Sec 4]. Let us first discover the environment E_1 to which the `datatype` declaration elaborates. E_1 will have a non-empty type environment TE_1 because a type has been declared, and also a non-empty variable environment VE_1 (we shall soon see why). In fact

$$E_1 = \{\}, TE_1, VE_1, \{\}$$

where

$$TE_1 \;=\; \{\mathtt{NAT} \mapsto (\mathtt{t}_{\mathtt{NAT}}, CE_1)\}$$

and

$$VE_1 \;=\; CE_1 \;=\; \{\mathtt{Zero} \mapsto \mathtt{t}_{\mathtt{NAT}},\ \mathtt{Succ} \mapsto \mathtt{t}_{\mathtt{NAT}} \to \mathtt{t}_{\mathtt{NAT}}\}$$

In general, a type environment maps type constructors to type structures (here TE_1 is a singleton map), and the type structure for a data-type constructor such as `NAT` contains two components:

1. a unique type name – here written $\mathtt{t}_{\mathtt{NAT}}$ – which distinguishes the type declared by *this particular elaboration* from all others;

2. a constructor environment – here CE_1 – which maps each value constructor of the data-type to its type scheme. (A *type scheme* is more general than a type, since it records the polymorphism of an object; but here there is no polymorphism.) The value constructor information is recorded again in the variable environment – here VE_1; by this means the declaration of a value constructor overrides the declaration of a variable with the same identifier, as it should do.

Exercise 1.1 Guided by the preceding remarks, complete the elaboration tree for the `datatype` declaration. You will need rules 19, 29, 30 and 49.

Next, let us discover the environment E_2 to which the `fun` declaration elaborates. The first thing to notice is that `fun` $\cdots$ is a derived form of value declaration, whose bare form is

```
val rec twice = fn Zero => Zero
                 | Succ x => Succ(Succ(twice x))
```

(See [Sec 1, p 1–2] for the distinction between *bare* and *derived* forms, and [App A, p 66] for all the derived forms.) So, since we have a value declaration, only the variable environment of E_2 will be non-empty. As we have already seen, a variable environment has the same form as a constructor environment; in fact

$$E_2 \;=\; \{\}, \{\}, VE_2, \{\}$$

where

$$VE_2 \;=\; \{\texttt{twice} \mapsto \texttt{t}_{\texttt{NAT}} \rightarrow \texttt{t}_{\texttt{NAT}}\}$$

Exercise 1.2 Complete the elaboration tree for the `fun` declaration. You will need rules 17, 27, 26, 42, 35, 14, 15, 16, 36, 9, 3, 43, 10, 2 and 7. You may find it helpful to look first at the elaboration of $topdec_3$ which is described below in detail.

Finally, knowing E_1 and E_2, we know the right-hand sides of all sentences involved in the elaboration of $topdec_1$. Note in particular that two names have been generated: the structure name $\texttt{m}_{\texttt{ARITH}}$ and the type name $\texttt{t}_{\texttt{NAT}}$. So the first component of the resulting basis B_1' is the name set $\{\texttt{m}_{\texttt{ARITH}}, \texttt{t}_{\texttt{NAT}}\}$.

Let us proceed more quickly with the elaboration of the three remaining *topdec*s.

For $topdec_2$, namely

$$B_2 \;\vdash\; \boxed{\texttt{open ARITH}} \Rightarrow B_2'$$

recall that $B_2 = B_1 \oplus B_1'$ contains the structure environment

$$\{\texttt{ARITH} \mapsto (\texttt{m}_{\texttt{ARITH}}, E)\}$$

where E records the declarations of `NAT`, `Zero`, `Succ` and `twice`. Now, via rules 100, 57 and 23, `open` just extracts the environment E; thus

$$B_2' = (\text{names}\, E, E) \text{ in Basis}$$

This allows the ensuing declarations to refer directly to `NAT`, `Zero`, `Succ` and `twice`.

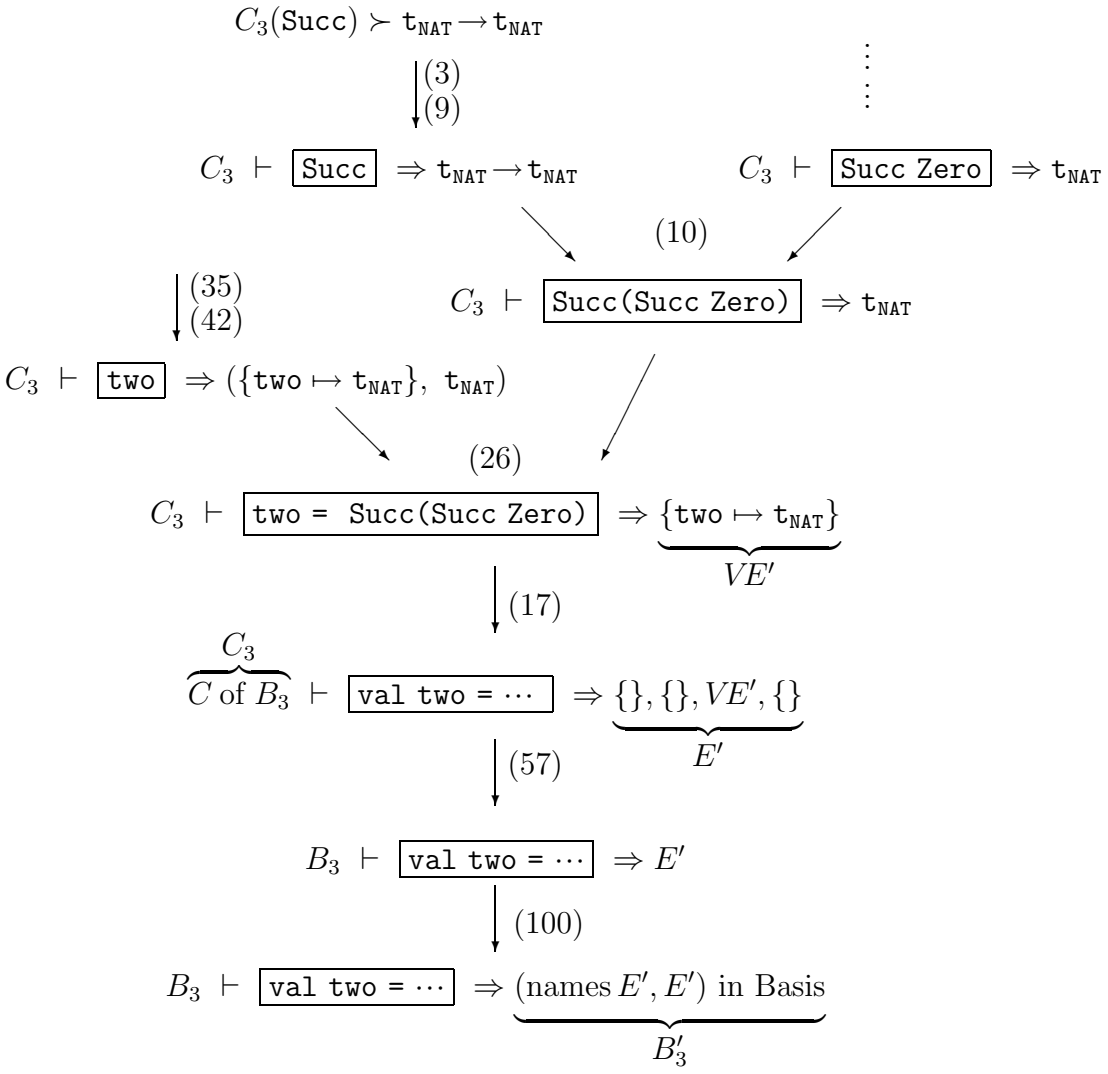

Figure 2: Part of the elaboration of $topdec_3$

For $topdec_3$, namely

$$B_3 \vdash \boxed{\texttt{val two = Succ (Succ Zero)}} \Rightarrow B_3'$$

the result basis B_3' will just contain the variable environment $\{\texttt{two} \mapsto \texttt{t}_{\texttt{NAT}}\}$; see Figure 2.

Finally, $topdec_4 =$ `twice two` is a derived form [Fig 18, p 68], whose bare form is

```
val it = twice two
```

so its elaboration will produce a basis which contains the variable environment $\{\texttt{it} \mapsto \texttt{t}_{\texttt{NAT}}\}$. This represents the special treatment of `it`, the variable which always records the type and value of the last expression executed at top level.

1.3 Evaluation

Analogous to elaboration, we now look for *evaluations* of our four *topdec*s:

$$\begin{array}{rl} & B_1 \vdash topdec_1 \Rightarrow B_1' \\ B_2 = B_1 \oplus B_1', & B_2 \vdash topdec_2 \Rightarrow B_2' \\ B_3 = B_2 \oplus B_2', & B_3 \vdash topdec_3 \Rightarrow B_3' \\ B_4 = B_3 \oplus B_3', & B_4 \vdash topdec_4 \Rightarrow B_4' \end{array}$$

where we now understand B_1, B_1', ... to be *dynamic* bases, and where we have dropped the subscript DYN.

The evaluation is inferred by the inference tree shown in Figure 3. The lower part of the tree uses the rules for dynamic semantics for Modules [Sec 7], while the upper half uses rules for dynamic semantics for the Core [Sec 6]. Without going into details,[6] the `datatype` declaration evaluates to the environment

$$E_1' = \{\texttt{Zero} \mapsto \texttt{Zero}, \texttt{Succ} \mapsto \texttt{Succ}\} \text{ in Env}$$

The lower half of the tree closely parallels that for elaboration of $topdec_1$ in Figure 1. We only remark that, as we learn from [Sec 6.3, p 47], there is no difference in the *dynamic* basis between an environment E and a structure. This is because the unique name m which labels a *static* structure has no relevance to evaluation.

This upper part of the tree illustrates an interesting point about function declarations. Note that, by rule 123, a `fn` *match* expression evaluates to a *closure* $(match, E, \{\})$ [Sec 6.6, p 49]. Here *match* is the body, or code, of the function while E is the environment in which it will be evaluated. The third component –

[6] As explained in Section 2.7, an inference rule concerning datatype declarations is missing in the Definition.

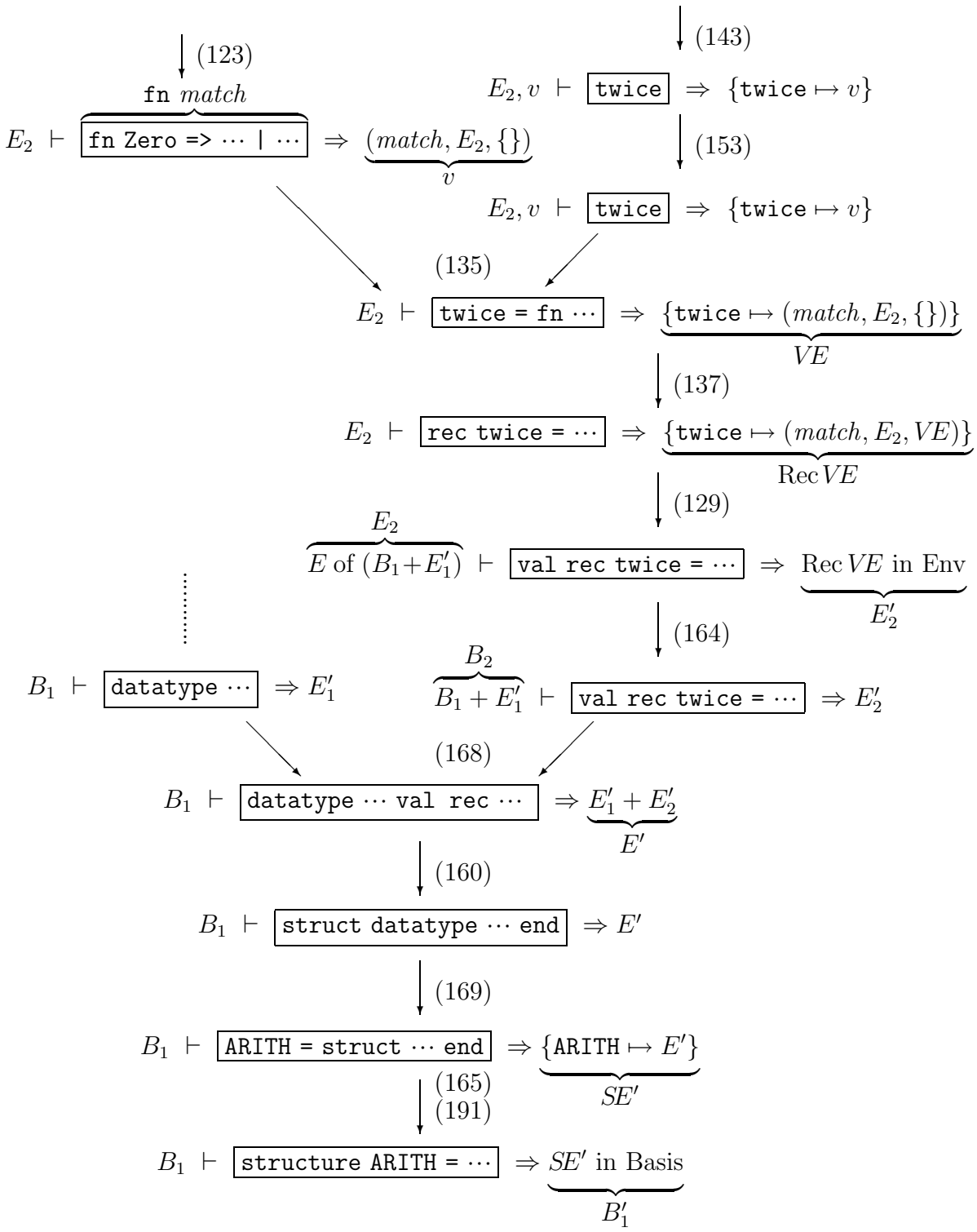

Figure 3: The evaluation of $topdec_1$

here $\{\}$ – is a variable environment, but what is the point of it? The answer has to do with recursion. The *non*-recursive value-binding

$$\texttt{twice = fn}\ match$$

evaluates by rule 135 to

$$VE \;=\; \{\texttt{twice} \mapsto (match, E_1, \{\})\}$$

Now the effect of the qualifier `rec` is to fold this variable environment inside itself, so that VE itself replaces the empty third component of the closure:

$$\mathrm{Rec}\, VE \;=\; \{\texttt{twice} \mapsto (match, E_1, VE)\}$$

We shall soon see how this enables `twice` to unroll its recursion as far as it needs to, in any application.

We can now summarise the evaluations of $topdec_2$ and $topdec_3$ quite briefly.

For $topdec_2$, namely

$$B_2 \;\vdash\; \boxed{\texttt{open ARITH}} \Rightarrow B_2'$$

we obtain via rules 191, 164 and 132 that the resulting basis B_2' has just one non-empty environment component, namely E' as defined in Figure 3. This allows the ensuing *topdec*s to refer directly to `twice`.

For $topdec_3$, namely

$$B_3 \;\vdash\; \boxed{\texttt{val two = Succ (Succ Zero)}} \Rightarrow B_3'$$

the evaluation uses rules 191, 164, 129, 135, 112, 111, 109, 105, 153 and 143; the only non-empty component in the result basis B_3' is the variable environment

$$\{\texttt{two} \mapsto (\texttt{Succ}, (\texttt{Succ}, \texttt{Zero}))\}$$

which is therefore incorporated into B_4 for future evaluations. The main point of interest here is that, despite the fact that the declaration of value constructors such as `Zero` and `Succ` is recorded in the basis, they evaluate to themselves without reference to the basis (see rule 105).

Finally, let us see how recursive evaluation works in $topdec_4$, in inferring the sentence

$$B_4 \;\vdash\; \boxed{\texttt{val it = twice two}} \;\Rightarrow\; B_4' \qquad (1)$$

Having followed previous inference trees, the reader should have no difficulty in discovering that the inference tree must contain the following sentence:

$$E_4 \;\vdash\; \boxed{\texttt{twice two}} \;\Rightarrow\; v \qquad (2)$$

Here $E_4 = E$ of B_4, and we expect to find that

$$v = (\texttt{Succ}, (\texttt{Succ}, (\texttt{Succ}, (\texttt{Succ}, \texttt{Zero}))))$$

To infer (1) from (2) requires rules 191, 164, 129, 135, 153 and 143.

Now in the inference of (2), `twice` will evaluate in E_4 to the closure which we have met already; in fact we can infer

$$E_4 \ \vdash \ \boxed{\texttt{twice}} \ \Rightarrow \ (match, E_1, VE) \tag{3}$$

from rules 111 and 104. Hence rule 117 must be used, with sentence (3) as its first hypothesis, to infer (2) above. The second hypothesis for this inference, namely

$$E_4 \ \vdash \ \boxed{\texttt{two}} \ \Rightarrow \ (\texttt{Succ}, (\texttt{Succ}, \texttt{Zero})) \tag{4}$$

is also deduced by rules 111 and 104. Now the third hypothesis for rule 117 must take the form

$$E_1 + \mathrm{Rec}\, VE \ , \ (\texttt{Succ}, (\texttt{Succ}, \texttt{Zero})) \ \vdash \ \boxed{match} \ \Rightarrow \ v \tag{5}$$

It is this use of the 'folding' operator Rec which ensures that, when the evaluation of `twice` is again required in the inference tree for sentence (5), it will once more evaluate to the closure $(match, E_1, VE)$.

The assiduous reader will no doubt be able to complete the inference tree for (5), using the rules 124–128 for matches, and the rules 140–159 for patterns. Working up the tree, he will find that (5) must be inferred by rule 126, and will then understand how the failure of a pattern-matching evaluation (yielding the special result FAIL) is handled. Once this is understood, all the points which the present example illustrates have been explored.

2 Dynamic Semantics for the Core

As we remarked in Section 1.1, static and dynamic semantics are independent of one another below the level of programs; this means that they can be presented independently. There is, in fact, a sense in which the static semantics takes precedence; elaboration "occurs before" evaluation, because part of its purpose is to reject certain top-level declarations as unacceptable. (The "prior occurrence" is reflected by rule 194 for programs, which makes no use of the evaluation of any *topdec* whose elaboration fails.) This is why the static semantics is presented first in the Definition. Readers who prefer to study it first can safely skip to Chapter 4.

Nevertheless, there are good reasons for *discussing* dynamic semantics first, as we shall do here. First, it is simpler in many ways than the static semantics; second, it deals with ideas which are more familiar to many people; third, by putting it first we actually reinforce our claim that it is truly independent. We shall find that type-checking and signature-checking really do not have to be understood when we assess the way in which evaluation is accomplished. Indeed, in [Sec 6.1, p 46] we stipulate that for the dynamic semantics we delete from the program text everything to do with types (though this stipulation needs minor modification; see Section 2.7 below).

2.1 Semantic objects

We shall begin with a few remarks about semantic objects [Fig 12, p 46; Fig 13, p 47]. In both static and dynamic semantics, all objects are built from simpler ones by four means: disjoint union ($\cup$), cartesian product ($\times$), finite subsets (Fin) and finite maps ($\xrightarrow{\text{fin}}$), and the identity of two objects is defined as usual for these constructions. Note particularly that records are maps, not lists; the order of components in a record does not affect its identity. This illustrates that our semantic definition is not fully concrete; it leaves to the implementer how to represent such objects.

Semantic objects are of course manipulated freely in the rules; they are combined or changed, and components are extracted. This is the essence of evaluation, and convenient notations are essential. These are all defined in the section on static Core semantics [Sec 4.3, p 18], but are used throughout the whole Definition.

An essential feature of operational – as opposed to denotational – semantics is that user-defined functions are not represented as abstract objects in a domain but, more dirtily, by the code which computes them, together with the environment in which that code should run; this is why the object class Closure, the user-defined functions, is built from the syntax class Match. The latter is the only compound syntax class which finds its way into the semantics, but of course many identifier classes appear in the semantic objects (e.g. Lab appears in Record).

2.2 The initial dynamic basis

In ML, as in most languages, there are many pre-defined objects. These are values and exception constructors which cannot, or cannot efficiently, be defined within ML itself, and they are known as the *basic values* BasVal and the *basic exceptions* BasExName [Sec 6.4,6.5, p 48]. Their meanings are fully described in [App D, p 77] which details the initial dynamic basis, B_0; for those basic values b which are functions, this determines the value APPLY(b, v) (for appropriate values v), which is used in rule 116 [p 52]. The name used for each b in the semantics [Sec 6.4, p 48] is chosen to be the same as the identifier to which it is bound in the initial dynamic basis B_0; but this is a mere convenience, and the user can overwrite these bindings by subsequent declarations.

The basic values b which are not concerned with input and output are all functions, and are fully described on [p 77–79] – some by means of ML declarations (but likely to be implemented more efficiently), others in normal mathematical notation. These functions have no side-effect upon the memory, but many of them (and also of the input/output functions) are capable of raising basic exceptions.

The basic values concerned with input and output are mostly functions, and are described [p 80] in terms of the simple notion of an *infinite character stream*; these streams constitute two basic ML types `instream` and `outstream`. Apart from the top-level dialogue, streams are the only defined interaction between ML and its environment. The standard input stream `std_in` (of type `instream`) and the standard output stream `std_out` (of type `outstream`), are among the basic values; by means of the basic stream functions the programmer can create other streams.

Most of the basic exceptions are raised by one or more of the standard functions; but the exceptions `Match` and `Bind` are raised by rules 118 [p 52] and 136 [p 54] as a result of the failure of pattern-matching in function application and in declarations, while the exception `Interrupt` is raised by user intervention.

There is no need to comment on most of the basic values, but we discuss the meaning of the equality function `=` briefly. It is a polymorphic predicate, and its effect upon the type system is dealt with in Section 5.2; here we are concerned with its evaluation. The brief description on [p 79] says that two objects are equal (`=`) if and only if they are identical. We have defined the identity of objects above, so the definition is precise, but we wish to draw attention to two points. First, the type system will ensure that two *function values* – whether user-defined functions or basic functions – are never tested for equality; this avoids arbitrary and unnatural decisions about what such a test should mean. Second, since a reference a is a *value*, two different reference values are always unequal – even if they "point to" equal values in the memory. In terms of LISP, this makes ML's `=` more like `eq` than like `equal`; for example

```
ref 1 = ref 1
```

evaluates to `false` (because each application of the constructor `ref` creates a new address). The advantage of this choice is that an implementation of `=` never has to proceed beyond an address to analyse the value addressed, and this avoids the complication of having to detect the cycles which can exist via addresses.

2.3 Evaluation order

Phrases in ML are evaluated from left to right. In a *pure* functional language, i.e. one without the imperative features of exceptions and assignment, this need not be stipulated – the great advantage of the absence of imperative features is that evaluation order need not be fully specified and yet the result of evaluation can be fully determined. But ML incorporates both exceptions and reference values (with assignment), which bring their own advantages for expressiveness; with these imperative features, we can only ensure determinacy by fully specifying the evaluation order. We indeed wish to ensure determinacy, so that there can be no doubt about the order in which side-effects occur.

This has led to two conventions in presenting the evaluation rules, namely the *state* convention and the *exception* convention [Sec 6.7, p 49–50]; by means of them the discipline of left-to-right evaluation is represented in a schematic way for all those rules which are not concerned directly with the imperative features. To clarify these conventions, let us take a simple example of applying them to a rule. Consider rule 112 for applying a constructor (omitting the side-condition for convenience):

$$\frac{E \vdash exp \Rightarrow con \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (con, v)}$$

In this rule, the result of the first evaluation is not needed for the second (contrast rule 108), so if *exp* and *atexp* could cause no side-effects no order constraint would be needed. But to achieve determinacy, ML must specify when the side-effects in the two sub-evaluations occur. In fact, the state convention dictates that the full form of rule 112 is

$$\frac{s, E \vdash exp \Rightarrow con, s' \qquad s', E \vdash atexp \Rightarrow v, s''}{s, E \vdash exp\ atexp \Rightarrow (con, v), s''}$$

i.e. the side-effects (state-change) of the left sub-evaluation occur first. But this still does not take account of the fact that a sub-evaluation may raise an exception, which should preclude later sub-evaluations; thus the exception convention decrees that rule 112 be augmented by two further rules – representing the two cases in which an exception is raised by a sub-evaluation:

$$\frac{s, E \vdash exp \Rightarrow p, s'}{s, E \vdash exp\ atexp \Rightarrow p, s'}$$

$$\frac{s, E \vdash exp \Rightarrow con, s' \qquad s', E \vdash atexp \Rightarrow p, s''}{s, E \vdash exp\ atexp \Rightarrow p, s''}$$

2.4 Function application

The heart of a functional programming semantics is the application of a function to arguments. The first thing to note is that ML is an *eager*, not a *lazy*, functional language, i.e. it evaluates the function argument before executing the function body. One reason for this is that the imperative features – exceptions and assignment to references – are confusing in a lazy language.

The activity of function application is represented by the seven rules 112–118. They evaluate the phrase *exp atexp* when *exp* evaluates to any of the following:

$$con \qquad en \qquad \texttt{ref} \qquad \texttt{:=} \qquad b \qquad (match, E, VE)$$

These have quite different effects.

The first two, *con* and *en*, are both constructor-like (an exception name *en* is a kind of dynamic constructor - see Section 2.6 below), and in these cases application merely performs a construction, i.e. it pairs the constructor with its argument as shown in rules 112 and 113. Pairing is the simplest way of composing two objects into one in such a way that the components can be recovered; of course it can be implemented in many ways.

The rules for `ref` and `:=` are, together with rule 158 for matching a reference value, the only rules which represent the use of references and assignment in ML. Rule 114 treats `ref` as a special sort of constructor; the pair (a, v) – where a is a new address – is constructed, just as rule 112 constructs the pair (con, v), but the construction (a, v) is kept in a special place; it becomes an element $a \mapsto v$ in the memory component *mem* of the state s, where it can submit to change. In rule 115, one may be confused by seeing that the argument of the assignment function `:=` is expected to be a record; but note that the type of 2-tuples in ML is just the type of records whose fields are labelled `1` and `2` [App A, Fig 15, p 67]. These 2-tuples may be called pairs, but at a higher level; they are not the same kind of pair as (con, v), by which our semantics represents the application of a constructor. Another point to note is that `:=` really is (the notation for) a value – let us call it the "assigner" ; in the initial dynamic basis [App D, p 77] the identifier `:=` is bound to the assigner, but one can bind it to other values – and indeed bind other identifiers to the assigner!

Rule 116 is the means of incorporating the meaning of basic function values b, using APPLY. This has already been discussed.

Rules 117 and 118 define the application of user-defined functions; an example was given in Section 1.3 showing how the environment components of the closure are used. Note that the result depends upon whether or not the argument v of the application succeeds in matching one of the patterns in the *match* – i.e. whether or not the function has been defined on v. When *match* contains no pattern matched by v, then the result of evaluating it is the special semantic object FAIL, which is not a value. According to the second restriction in [Sec 4.11, p 30], the *match* in a function expression `fn` *match* should be *exhaustive*, i.e. every value (of the right

type) should match at least one of its patterns; the compiler – or the elaborator – is supposed to issue a warning if this restriction is not met, but the elaboration is not to be failed on that account. Thus a program which has not evoked such a warning should never raise the exception `Match` when it is evaluated. But of course the evaluator, having no knowledge of types, cannot know whether a match is exhaustive. Evaluation is like a railway locomotive which is perfectly well built, but cannot know that the benevolent railway designer has built a track which never runs into the sand.

There is, *a priori*, another way in which the evaluation could be inadequate; for there are values to which *exp* may evaluate – such as special values $sv \in \mathrm{SCon}$ [Sec 2.2, p 3] – although no rule defines how to apply them as functions. Why do we not have a rule such as

$$\frac{E \vdash exp \Rightarrow sv}{E \vdash exp\ atexp \Rightarrow [\texttt{Wrong}]}$$

so that the user can detect such bad applications and recover from them, by handling the exception `Wrong`? Again, the answer is that such a bad application has to be the consequence of a badly typed expression, such as `4.6(true)` , which will have been rejected by elaboration. In fact we believe the following to be true, though we have not proved it:[7]

> Every program which does not compute infinitely, and which elaborates successfully, also evaluates successfully.

When stated formally, this implies that there are no cases of evaluation within a well-typed program which are not covered by the existing rules.

2.5 Pattern matching

If function application is the heart of evaluation, perhaps pattern matching is its life-blood; in every application of a user-defined function, the argument is to be tested against all patterns in the *match* – perhaps simultaneously – and the *mrule* selected is the left-most successful one. This is the work of rules 124–126 [p 53] for matches. Much of the the remarkable efficiency recently gained for functional programming is due to good algorithms for matching of a value to several patterns at once. And we are free to juggle with the order in which patterns are matched just because pattern matching has no side-effects. This latter fact is perhaps obvious; for, given a pattern *pat*, an environment E and a value v to be matched, there is no expression evaluation involved in the evaluation

$$E, v \vdash pat \Rightarrow \ldots$$

[7] An analogous result for a fragment of ML was proved in *A theory of type polymorphism in programming languages* by R.Milner, J. Comp. Sys. Sci., Vol 17, 1978, pp348–375.

All that has to be done is to pick v apart, as dictated by the structure of *pat*, collecting along the way a binding for each variable in *pat*. This is the work of rules 140–159 [p 54–56]. We can formulate this "no side-effects" property neatly, and the proof is straightforward:

Theorem 2.1 (Pattern matching) *Let E, v and pat be any environment, value and pattern. Suppose that*

$$s, E, v \vdash pat \Rightarrow r, s'$$

can be inferred for states s, s' and result r. Then r is either a value environment VE, or else the special result FAIL *– it cannot be an exception packet p. Moreover, $s = s'$.*

Proof First one must expand all the rules 140–159 according to the state and exception conventions. Then the proof proceeds easily by structural induction on *pat*.

The presence of *either* kind of side-effect – exception-raising or assignment – would invalidate simultaneous pattern matching, because the order of matching could affect the result.

It is worth noting that the environment E and the state s play only a small part in pattern matching; E is only needed in rules 156 and 157 to look up the exception name bound to an exception constructor, while s is only needed in rule 158 to look up a reference.

Before leaving patterns we should mention the restriction that no variable should occur more than once in a pattern [Sec 2.9, p 9]. There is no semantic barrier to relaxing this restriction, for values whose types admit equality [Sec 4.4, p 18]; to relax it completely would allow us to test equality of all values – even functions! – by declaring

```
fun equal(x,x) = true
  | equal  _   = false
```

However, the effect of repeated variables on efficient simultaneous matching needs investigation.

2.6 Exceptions

An exception packet $p = [e]$ has as its content an exception value $e \in$ ExVal, i.e. an exception name *en* possibly paired with a value; that is, $e = en$ or $e = (en, v)$. We shall use the term "exception" to mean an exception value e, not a packet $[e]$. As indicated in [Fig 13, p 47] exceptions are indeed a kind of value; they can be treated in computation just like any other value (passed as function arguments, bound to identifiers etc.). In fact they constitute a type `exn`, one of the basic types of ML [App C, p 74].

The distinguishing feature of exceptions (among values) is just that they, and only they, can be enveloped in packets and *raised*, by rule 122, and subsequently *handled* by rules 120 and 121. Note, however, that a packet $p = [e]$ is *not* a value; its tag or envelope, [ ], serves to distinguish it and allow it to be treated specially by the rules introduced by the exception convention (see Section 2.3 above), thus aborting evaluations appropriately.

Exception constructors *excon* behave in some ways just like value constructors *con*; when an exception e is handled by rule 120 or 121 it is analysed by the *match* exactly as datatype values are analysed in function application. (Indeed, as we said above, exceptions themselves can *be* function arguments.) But there are important differences. First, the quasi-datatype `exn` is *extensible*; its constructors are not given once and for all, but new constructors can be declared at any time by the `exception` declaration, rule 130 [p 53]. Second, there may be two different declarations of the same exception constructor, and a way has to be found to avoid a confusion which may result.[8] Consider a skeletal example:

```
exception E ;
fun f() = let exception E in raise E end ;
f() handle E => 3
         | _ => 4
```

If the result were `3`, it would mean that the first *mrule* of the *match* was successful, and this would imply that a user at top-level could accidentally handle an exception whose constructor was defined locally inside a function-body and should therefore be inaccessible outside. In fact the result is `4`, because the exception raised uses the *inner* declaration of `E`, while the pattern `E` in the handling expression corresponds to the *outer* declaration. To achieve this, an exception constructor `E` is not represented semantically just by itself (as are value constructors, cf. rule 105 [p 51]), but by a newly generated exception name *en*, see rule 138 [p 54]. The side-condition in that rule, $en \notin ens$ of s, shows why the state s must contain a component *ens* representing the exception names previously generated.

Because new exception constructors can be declared anywhere, and exceptions treated as normal values, the type `exn` has all the qualities of a *general* type: a type into which one can inject the values of any type at all. We can think of such general types as extensible datatypes, and can imagine a more general feature in ML which would allow the user to define his own extensible datatypes at will. It is not clear how much more valuable this would be than having just one such type. As it stands, ML can perhaps be criticized for providing a general type as a by-product of its exception mechanism, rather than directly; but a better alternative is not obvious. The combination of the two concepts may turn out to be more natural than it seems at first.

[8] No confusion can result from two declarations of the same *value* constructor, because they must have different types — and also because value constructors in any case evaluate to themselves.

Given the general nature of the type `exn`, the reader will hardly be surprised to find that equality is not defined on exceptions [App C, p 74].

2.7 Constructors versus variables

In this section we clarify the distinction between variable identifiers and constructor identifiers, and also correct a mistake in the Core dynamic semantics which is relevant to the distinction (though the mistake only shows up in Modules, with the use of signatures).

The fourth rule concerning the status of identifiers [p 5] implies that an identifier which is introduced as a value constructor or exception constructor cannot be used as a variable. Thus, after

```
datatype T = A | B | C
```

an attempt to re-use `A` as a variable, e.g. to redeclare it by `val A = ...` or to use it as a formal parameter by `fn A => ...` will fail.[9] With one exception, if the above declaration is at top level then no ensuing *topdec*s of the *Core language* can remove `A`'s status as a value constructor. Note in particular that if we redeclare `T` by

```
datatype T = C | D | E
```

then `A` and `B` will still be constructors belonging to the hidden type `T`.

The single exception is the use of the `open` declaration. Suppose a structure `S` contains a variable component `A` (i.e. `S.A` is a long variable), say with value `1`. Now consider

```
datatype T = A ;
open S ;            (*1*)
A ;                 (*2*)
val A = "Monday" ;
```

At point `(*1*)` the identifier `A` has acquired variable status, and moreover denotes a new value, `1`, which is printed at `(*2*)`. Further, the rebinding of `A` to `"Monday"` will succeed.

The above example shows that, as far as the Core language is concerned, the only way of removing constructor status from an identifier `A` is to unpack a packaged declaration of `A`. But another phenomenon arises in Modules, which is best considered at this point. If the declaration `datatype T=A` occurs inside a structure, then it is possible – by ascribing a certain signature to that structure – to cause `A` to change its *status* from constructor to variable without changing its

[9] The phrase may be valid without having the intended meaning; for example `val A = A` will have no effect at all, because the pattern `A` contains no variables, but just consists of a single constructor.

meaning (i.e. the value which it denotes). Because of this, the remark in [Sec 6.1, p 46] – that datatype declarations can be ignored during evaluation – is not quite correct. For, in order to prepare for the possible status-change just mentioned, a datatype declaration must generate a suitable variable environment. (This does not affect the fact that, while an identifier still preserves its constructor status, it does indeed evaluate to itself by rule 105 [p 51], without use of the variable environment.)

To formulate the appropriate rule for evaluating datatype declarations, suppose that a datatype binding *datbind* introduces the value constructors $con_1, \ldots, con_k$. Let $VE_{datbind} = \{con_1 \mapsto con_1, \ldots, con_k \mapsto con_k\}$. We then add a new rule for declarations, whose natural place is between rules 129 and 130 [p 53]:

$$E \vdash \texttt{datatype}\ datbind \Rightarrow VE_{datbind} \text{ in Env}$$

Note that the result does not depend upon E. The use of this rule is illustrated by Exercise 3.1 in Chapter 3. A small change is also needed in rule 130, to allow status-change also for exception constructors; the rule should now read

$$\frac{E \vdash exbind \Rightarrow EE \qquad VE = EE}{E \vdash \texttt{exception}\ exbind \Rightarrow (VE, EE) \text{ in Env}} \tag{130}$$

However, `abstype` declarations can still be ignored during evaluation; the reason is that no structure can provide access to the constructors of an abstract type.

These corrections bring the dynamic semantics into accord with rules 19 and 21 [p 25] in the static semantics of datatype and exception declarations, and with rules 73 and 74 for the corresponding specifications, all of which generate a *VE* component.

2.8 Evaluation theorems

In this section we collect a few theorems which represent interesting properties of evaluation. They also serve to introduce the proof technique of induction on the depth of inference trees, which is one of the advantages of an operational semantic definition. We should point out that Theorem 2.1 was particularly simple; it could be proved by structural induction on patterns because in the pattern-matching rules the pattern in a hypothesis is always a subpattern of that in the conclusion. This structural property is not true for the phrases in all rules, cf. rule 117 or 118.

For those not interested in proof, the results themselves, and the fact that their proofs are not hard, should give some confidence in the semantic method.

We begin with a few properties of states $s = (mem, ens)$ which are by no means obvious from the inference rules, though they may be properties which one would either expect or hope for. First, it is quite easy to prove that, throughout evaluation, the state always increases. Since the property holds for the whole language, not just for the Core, we state it generally. As usual, for every finite map f, we denote the domain of f by $\mathrm{Dom}(f)$ [Sec 4.2, p 17].

Theorem 2.2 *For any phrase, let the sentence*

$$s, A \vdash \mathit{phrase} \Rightarrow A', s' \qquad (*)$$

be inferred, where $s = (\mathit{mem}, \mathit{ens})$ *and* $s' = (\mathit{mem}', \mathit{ens}')$ *and* A, A' *are semantic objects. Then*

$$\mathrm{Dom}(\mathit{mem}) \subseteq \mathrm{Dom}(\mathit{mem}') \quad \textit{and} \quad \mathit{ens} \subseteq \mathit{ens}'$$

Proof The proof is by induction on the depth of inference of the inferred sentence $(*)$. Since it is an evaluation, the inference must conclude with one of the rules 103–159 for the Core, 160–193 for Modules or 194–196 for programs (in its full form according to the state convention), or else with one of the rules added according to the exception convention. Whichever rule is used, the hypotheses have shorter proofs, so we can assume the theorem for those proofs.

Now for any rule expanded by the state convention it is easy to see that the required result for the conclusion follows from the assumed result for the hypotheses. See for example the full form of rule 112 displayed in Section 2.3 above, or the general form at the top of [Sec 6.7, p 50]. For if there are no hypotheses at all in the rule by which $(*)$ is inferred, then $s = s'$; if there is only one then s and s' are the same in the conclusion as in the hypothesis; if there are two or more – as for rule 112 – then the result follows simply by transitivity of $\subseteq$. A glance at the rules added for rule 112 by the exception convention, displayed at the end of Section 2.3, shows that the result is easy for such rules also.

It remains to deal with rules 114, 115, 138, 158 and 194–196 to which the state convention does not apply, and with their companions added by the exception convention. In each case, a simple check yields the required result. Take for example rule 114:

$$\frac{s, E \vdash \mathit{exp} \Rightarrow \texttt{ref}, s' \qquad s', E \vdash \mathit{atexp} \Rightarrow v, s'' \qquad a \notin \mathrm{Dom}(\mathit{mem} \text{ of } s'')}{s, E \vdash \mathit{exp}\ \mathit{atexp} \Rightarrow a, s''+\{a \mapsto v\}}$$

By applying induction to the two hypotheses of the rule, and then using transitivity, we get

$$\mathrm{Dom}(\mathit{mem} \text{ of } s) \subseteq \mathrm{Dom}(\mathit{mem} \text{ of } s'')$$

and *a fortiori*

$$\mathrm{Dom}(\mathit{mem} \text{ of } s) \subseteq \mathrm{Dom}(\mathit{mem} \text{ of } s''+\{a \mapsto v\})$$

as required. The result for *ens* is even more direct.

The import of this simple theorem is that the semantic definition does not cater for garbage collection; the state is always growing, even though some addresses

and some exception names may become inaccessible because all uses of them have been erased from the basis by the superposition of new declarations. Of course implementers will take advantage of this inaccessibility to re-use memory locations.

Another fact, with a similar proof, is that no attempt is ever made to look up a memory address which is not in the domain of the current memory – i.e. which is not currently associated with a value – or to raise an exception whose name is not recorded in the current state. Unlike the previous result, we would be greatly concerned if this were not true, or not easily deducible from the rules.

Let us express the property precisely, and in a form which is amenable to inductive proof. We need to assume that, if the evaluation takes place against semantic background (s, A), then any address a which occurs anywhere in the background – including, perhaps, in a value stored in s – is itself bound to a value in s. A similar assumption must be made about exception names. To express this assumption, let $\mathrm{Addrs}(A)$ and $\mathrm{Ens}(A)$ be the addresses and exception names occurring anywhere in a semantic object A.

Theorem 2.3 (No Dangling References) *For any phrase, let the sentence*

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

be inferred, where $s = (\mathit{mem}, \mathit{ens})$ *and* $s' = (\mathit{mem}', \mathit{ens}')$*, and* A*,* A' *are semantic objects. Assume also that* $\mathrm{Addrs}(s, A) \subseteq \mathrm{Dom}(\mathit{mem})$ *and* $\mathrm{Ens}(s, A) \subseteq \mathit{ens}$*. Then*

$$\mathrm{Addrs}(s', A') \subseteq \mathrm{Dom}(\mathit{mem}') \quad \textit{and} \quad \mathrm{Ens}(A', s') \subseteq \mathit{ens}'$$

Proof The structure of the proof is very similar to the previous one, and we shall only look at one interesting case, rule 158:

$$\frac{s(a) = v \qquad s, E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}, s}{s, E, a \vdash \mathtt{ref}\ \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}, s}$$

We assume $\mathrm{Addrs}(s, E, a) \subseteq \mathrm{Dom}(\mathit{mem}\ \text{of}\ s)$ and $\mathrm{Ens}(s, E, a) \subseteq \mathit{ens}\ \text{of}\ s$. So in particular, since v occurs in s, we have $\mathrm{Addrs}(v) \subseteq \mathrm{Dom}(\mathit{mem}\ \text{of}\ s)$ and $\mathrm{Ens}(v) \subseteq \mathit{ens}\ \text{of}\ s$; hence

$$\mathrm{Addrs}(s, E, v) \subseteq \mathrm{Dom}(\mathit{mem}\ \text{of}\ s) \quad \text{and} \quad \mathrm{Ens}(s, E, v) \subseteq \mathit{ens}\ \text{of}\ s$$

Hence we can apply induction to the hypothesis, and the required result is immediate.

We conclude with a theorem about determinacy. A guiding principle of the Definition is that the effect of each program should be completely determined, so that there is no chance for different implementations to make different arbitrary choices of interpretation when the language definition admits freedom of choice. True, we have violated this principle in one or two cases. One case is that the

range of integers, and the range and precision of reals, have not been specified; another is that the precise points at which a program may be interrupted (by a user-generated `Interrupt`) is not indicated.

Notwithstanding these few shortcomings, the dynamic semantics is determinate in (informally) the following sense: for each *phrase*, s and appropriate A there is at most one pair (A', s') such that the sentence

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

can be inferred. This is slightly inaccurate, because a *new* address a, or a *new* exception name en, may be chosen arbitrarily in rule 114 or 138. A completely accurate statement, therefore, is as follows:

Theorem 2.4 (Determinacy) *Let the two sentences*

$$s, A \vdash \mathit{phrase} \Rightarrow A', s' \qquad s, A \vdash \mathit{phrase} \Rightarrow A'', s''$$

both be inferred. Then (A'', s'') only differs from (A', s') by a one-to-one change of addresses and exception names which do not occur in (s, A).

Proof Again, the proof is a long induction which is almost straightforward. However, one needs first to prove an auxiliary theorem (perhaps it has some interest its own right) which asserts that if $s, A \vdash \mathit{phrase} \Rightarrow A', s'$ can be inferred, and we change the addresses and exception names occurring in (s, A) in a one-to-one manner, the sentence can still be inferred with appropriate one-to-one change of names in (A', s').

This theorem complements the result stated informally at the end of Section 2.4 – that there are enough rules to ensure that evaluation is always successful. Together they assert that (at least for terminating programs) there is *exactly* one evaluation.

One important theorem, which we do not formulate here, concerns evaluation and elaboration together. It asserts that if a program is executed successfully then it yields a *sound* basis B, in the sense that the *value* bound to any variable in B_{DYN} does indeed possess the *type* bound to the variable in B_{STAT}. This theorem has been formulated and proved for a smaller language including imperative features.[10]

We should emphasize that all the theorems in this section hold for the entire language, not just for the Core.

[10] M. Tofte, *Type inference for polymorphic references*, to appear in Information and Computation, 1990.

3 Dynamic Semantics for the Modules

In the Core, evaluation is separated from type checking, and consequently type information in ML programs can be erased from Core programs [Sec 6.1, p 46] before evaluation (with the exception of datatype declarations, see Section 2.7). The dynamic semantics of Modules is done in the same spirit. In this chapter we proceed by pursuing the goal of separating out evaluation from elaboration; by discovering what static information is required for evaluation, for there is some, one grasps the idea of Modules more clearly than by a rule-by-rule commentary.

We begin in Section 3.1 by discussing the basic design underlying Modules. In Section 3.2 we treat structures by themselves without signatures or functors. In Section 3.3 we deal with interfaces – the aspect of signatures which affects evaluation. In Section 3.4 we extend the discussion to functors. Finally, in Section 3.5 we look at an alternative meaning for signatures in structure and functor declarations.

3.1 Structures and signatures

It is well known from practical programming that, especially when large programs are developed, a major part of the development effort lies in deciding "what goes where". It is also common experience that one wishes to encapsulate one or more types and operations on those types so that they can be regarded as a single unit. Finally, it is desirable that manipulation of such units can be expressed in the programming language itself, so that it becomes subject to automatic checking.

This raises the interesting question: What kind of thing is such a unit? It is not a type in the usual sense, for it contains values. Nor is it a value in the usual sense, for it contains types. The ML term for such a hybrid object is a *structure*. A structure results from executing a declaration and encapsulating the environment so obtained, and one can name this structure by means of a *structure declaration*. A very simple example is

```
structure lamp =
  struct
    datatype bulb = ON | OFF
    fun switch(ON) = OFF | switch(OFF) = ON
  end
```

Having encapsulated this pair of declarations as a structure, one can then (or later) decapsulate it using the `open` declaration

```
open lamp
```

and the effect is just that the components of the structure, here `bulb` and `switch`, become directly available.

One way of viewing structures is provided by theories of *dependent types*.[11] From this point of view, one can think of the above structure as a pair $\langle$`bulb`, $cl\rangle$ of a type and a value, here a (particular) type `bulb` and a closure cl for `switch`. (Let us ignore the constructors just now.) Moreover, still from the point of view of dependent types, the type of the lamp structure is something like

$$[bulb : \text{TYPE},\ \texttt{switch} : bulb \to bulb]$$

Here TYPE is not a type itself, but a *kind*. (Kinds classify types, whereas types classify values.) Also note that *bulb* is now a variable over types, rather than the name of a particular type, and that the type of `switch` depends on *bulb*.

In ML, structure types are called *signatures*; in the above example, a signature for `bulb` is declared by

```
signature APPLIANCE =
  sig
    type bulb
    val switch : bulb -> bulb
  end
```

The above example, trivial though it is, shows that the concept of structure type is important. First, a signature summarises the contents of a structure without all its detail. But more importantly, a signature is an abstract description of all structures whether written or yet to be written, that match the signature; hence `APPLIANCE` is a signature for all things that have at least a bulb and a switch. (As we shall see later, a structure is allowed to have more than just a lamp and a switch and still match the signature.) This opens up possibilities for information hiding in various ways; for example, ML allows one to curtail a structure by a signature, thereby creating a view of the structure in which only what is specified by the signature is accessible.

We have seen that structures are a hybrid form of values and types, and that signatures are somewhat unusual types in that the value part of a signature can depend on its type part. Since structures can contain both values and types, if we are not careful the distinction between elaboration and evaluation could be lost. This *phase distinction*[12] is at the heart of efficient execution for any typed language. Certainly, one would not happily depart from the principle that the type of an expression has to be deduced just once. In the Core we have seen that

[11] See for example D. MacQueen, *Using dependent types to express modular structure: experience with Pebble and ML*, Proc. 13th Annual ACM Symp. on Principles of Programming Languages, 1986; also J. Mitchell and R. Harper, *The essence of ML*, Proc. 15th Annual ACM Symp. on Principles of Programming Languages, 1988.

[12] This term is taken from R. Harper, J. Mitchell and E. Moggi, *Higher-order modules and the phase distinction*, Proc. 17th Annual ACM Symp. on Principles of Programming Languages, 1990.

it is sufficient to elaborate every expression just once, and this is the consequence of two properties. The first is the phase distinction which ensures that the type of an expression never depends on any dynamic object; the second is the existence of principal types, which will be discussed later.

The design of the Modules lifts these two basic principles to the higher level of structures and signatures. The phase distinction is manifest in the inference rules of the Definition, simply because dynamic and static objects are treated separately. Also, by analogy with types, the existence of principal signatures is proved in this Commentary.

Essentially the same kind of lifting works for other languages as well;[13] The lifting does, however, rely on a slight redundancy between the semantic objects of evaluation and elaboration. Evaluation involves *interfaces*, which are stripped-down versions of signatures in the sense that they are concerned solely with the shape of objects, not with their kinds or types; see Section 3.3.[14]

Let us finish this section with an example of the kind of language construct that would destroy the phase distinction. Suppose we could write

```
structure A1 = struct val x = 3 end
      and A2 = struct val x = "Monday" end
fun f(b: bool) =
    let (if b then open A1 else open A2)
      in x
    end
```

Then it would not be clear where the last occurrence of `x` is bound, nor indeed what type it has. In fact, ML preserves the phase distinction by forbidding conditional declarations.

3.2 Structures only

Already in the Core dynamic semantics there is a need for a *structure environment*, but only for the purpose of looking up long identifiers and for evaluating the `open` declaration, rule 132 [p 53]. A structure environment maps structure identifiers to *structures*, and as far as evaluation is concerned a structure is just an environment $E = (SE, VE, EE)$, whose first component SE is again a structure environment. Thus the definitions of Env and StrEnv in [Fig 13, p 47] are mutually recursive. We tend to use the term "structure" for an environment when we are concerned with encapsulating it or naming it.

[13] see D.T. Sannella and L. Wallen, *A calculus for the construction of modular Prolog programs*, Proc 1987 IEEE Symp. on Logic Programming, San Francisco, 1987, 368–378; to appear in Journal of Logic Programming.

[14] Interfaces are not needed in the afore-mentioned type-theoretic treatments of modules, apparently because those accounts are not concerned with variations in shape.

We shall begin by studying a first approximation to Modules, which provides nothing more than the ability to encapsulate and name environments. This entails a sublanguage containing essentially just structure *expressions* and structure *declarations*. If this were all that Modules provided, it would still be of some use. The relevant evaluation rules would be as shown in Figure 4; these are the rules 160–169 [p 59,60] but with rule 162 omitted (since we have omitted functors), and with rule 169 simplified (since we have omitted signatures). In reading the rules, first note that in the full semantics [Fig 14, p 57] a basis B contains both functors and signatures, but since we are omitting these we can for now think of B as just an environment.

It is worth exhibiting the very close parallel between these rules for structures and a subset of the rules for Core language expressions; we can then see that, for structures alone, there are hardly any new questions to discuss about dynamic semantics. Rules 160,161 and 163 correspond closely to rules 109, 104 and 108 [p 50,51] for expressions. Rule 164 represents the only link between the Core and the Modules evaluation; thus we see that a Core declaration *dec* – yielding an environment – is atomic from the Modules viewpoint. This indicates that, at least for evaluation, it is clear how ML Modules could sit above any programming language. Rules 165–168 are exactly parallel to rules 129, 131, 133 and 134 for Core declarations [p 53], while the simplified rule 169 is exactly what rule 135 for value bindings would become if a single value binding took the simple form *var* `=>` *exp* (rather than the more general *pat* `=>` *exp*).

3.3 Signatures and interfaces

To see the effect of an explicit signature in a structure binding, consider the following program:

```
structure A: sig val x: int end
        = struct val x = 3 and y = true end ;
A.y
```

What will happen? ML's answer is that the execution will fail. The reason is that the explicit signature `sig val x: int end` curtails the view, seen through `A`, of the structure generated by the `struct...end` expression. In fact the *elaboration* of the *topdec* `A.y` will fail; thus, by rule 194 [p 64], its *evaluation* will not even be attempted.

It is tempting to try to preserve the simplified rule 169, Figure 4, as the dynamic semantics for structure bindings; in other words, we would like to delete all mention of signatures from a program for the purpose of evaluation. To justify defining Modules evaluation in this way, completely ignoring the effect of structure-curtailment, one would claim that the only cases which would be improperly handled are those where elaboration fails, as in the above example, and that these cases can therefore be ignored.

Structure Expressions $\boxed{B \vdash \mathit{strexp} \Rightarrow E/p}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow E} \tag{160}$$

$$\frac{B(\mathit{longstrid}) = E}{B \vdash \mathit{longstrid} \Rightarrow E} \tag{161}$$

(OMITTED) (162)

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B + E \vdash \mathit{strexp} \Rightarrow E'}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E'} \tag{163}$$

Structure-level Declarations $\boxed{B \vdash \mathit{strdec} \Rightarrow E/p}$

$$\frac{E \text{ of } B \vdash \mathit{dec} \Rightarrow E'}{B \vdash \mathit{dec} \Rightarrow E'} \tag{164}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \mathit{SE} \text{ in Env}} \tag{165}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{166}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{167}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle\texttt{;}\rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{168}$$

Structure Bindings $\boxed{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}/p}$

$$\text{(SIMPLIFIED)}\ \frac{B \vdash \mathit{strexp} \Rightarrow E \qquad \langle B \vdash \mathit{strbind} \Rightarrow \mathit{SE}\rangle}{B \vdash \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{strbind}\rangle \Rightarrow \{\mathit{strid} \mapsto E\}\ \langle + \mathit{SE}\rangle} \tag{169}$$

Figure 4: Simplified rules for structures only

Unfortunately the claim is false, but only because of the `open` declaration. A simple example – using the same structure declaration as above – shows why:

```
val y = false ;
structure A: sig val x: int end
        = struct val x = 3 and y = true end ;
open A ;
y ;
```

If the signature here had no curtailing effect, then the evaluation of `y` at the end would return `true`, because the `y` in `A` would supersede the `y` previously declared. But in fact the evaluation should return `false`, since the curtailment should exclude `y` from `A`. Note particularly that the program elaborates perfectly well.

This leads to the notion of *interface*, introduced in [Sec 7.2, p 57,58]. See also the semantic object classes [Fig 14, p 57]. An interface is a kind of *shape* for an environment, and signature expressions evaluate to interfaces. While an environment is a triple $E = (SE, VE, EE)$, an interface is a triple $I = (IE, vars, excons)$. An interface environment IE maps structure identifiers to interfaces. We could say that E *conforms* to I if $\mathrm{Dom}\, SE \supseteq \mathrm{Dom}\, IE$, $\mathrm{Dom}\, VE \supseteq vars$, and $\mathrm{Dom}\, EE \supseteq excons$, and recursively each environment in SE conforms to the corresponding interface (if existent) in IE.

Now the shape of the signature ascribed to `A` in our example is

$$I = (\{\}, \{\mathtt{x}\}, \{\})$$

– no structures, one variable `x`, no exceptions. On the other hand the structure expression for `A` evaluates to the environment

$$E = (\{\}, \{\mathtt{x} \mapsto \mathtt{3}, \mathtt{y} \mapsto \mathtt{true}\}, \{\})$$

which clearly conforms to the shape. The operator $\downarrow$ [p 58], which curtails an environment by an interface, can then be applied to yield

$$E \downarrow I = (\{\}, \{\mathtt{x} \mapsto \mathtt{3}\}, \{\})$$

and this is the result of evaluating the structure declaration according to the full rule for structure bindings. The full rule is therefore

$$\frac{\begin{array}{c} B \vdash strexp \Rightarrow E \qquad \langle \mathrm{Inter}\, B \vdash sigexp \Rightarrow I \rangle \\ \langle\langle B \vdash strbind \Rightarrow SE \rangle\rangle \end{array}}{\begin{array}{l} B \vdash strid\ \langle \mathtt{:}\ sigexp \rangle\ \mathtt{=}\ strexp\ \langle\langle \mathtt{and}\ strbind \rangle\rangle \Rightarrow \\ \qquad \{strid \mapsto E\langle\downarrow I\rangle\}\ \langle\langle +\ SE \rangle\rangle \end{array}} \tag{169}$$

Note that there is no side-condition in the rule to ensure conformity, i.e. to ensure that the environment E has *enough* components; but this will already have been

ensured by the corresponding elaboration rule 62 [p 39] in the static semantics, by its side-condition concerning *enrichment* $\prec$ [Sec 5.11, p 34]. This remark about static semantics can be ignored on first reading; however, it is in the same spirit as some of our remarks in the previous chapter, where we pointed out that some aspects of good behaviour of Core programs need not be checked by the evaluation rules.

We shall now illustrate how a signature can change the status of an identifier from constructor to variable. Consider

```
structure S: sig type T val A: T end
          =  struct datatype T = A | B end
```

If the signature expression were absent, then the identifier `A` would have constructor status after this declaration. But the signature gives it variable status instead; thus it cannot be used (as a constructor) in a pattern, but still retains its meaning – it still *denotes* the constructor `A`. We pointed out in Section 2.7 that, for this reason, datatype declarations cannot be completely ignored in the dynamic semantics of the Core, as was wrongly stated in [Sec 6.1, p 46]; the following exercise shows the need for the new rule defined in Section 2.7.

Exercise 3.1 What environment E does the structure expression (after `=`) evaluate to, in the above example, using the new rule? What interface I does the signature expression evaluate to? What environment becomes bound to `S`?

Rules 170–186 [p 60, 61] are entirely to do with evaluating a signature expression to an interface. After the above discussion, there is little need for further comment on them. Note that the signature environment G of the dynamic basis binds each declared signature identifier to an interface I. Note also that a signature expression is not evaluated in a basis B, but in an abstraction of B which is an *interface basis* IB; if $B = (F, G, E)$ then $IB = (G, IE \text{ of } (\text{Inter } E))$, where Inter abstracts an environment to its shape. Finally note the reduced syntax described in [Sec 7.1, p 57]; not only has all mention of types been removed before evaluation, but also all mention of *sharing*; this means that evaluation does not depend at all on the *identity* of structures, but only upon their contents: values, exceptions and other structures.

Exercise 3.2 The `include` specification need not have been defined independently, since it can be defined in terms of `local` and `open`. In fact

$$\texttt{include}\ sigid_1 \cdots sigid_n$$

is exactly equivalent to

$$\texttt{local structure}\ strid_1 : sigid_1\ \texttt{and} \cdots \texttt{and}\ strid_n : sigid_n \\ \texttt{in open}\ strid_1 \cdots strid_n\ \texttt{end}$$

Prove this equivalence as far as evaluation is concerned; that is, prove that for any *IB* and I the former phrase evaluates to I in *IB* if and only if the latter does. (See also Exercise 6.1 for the analogous equivalence in the static semantics.)

3.4 Functors

A functor is a mapping from structures to structures. Syntactically, functors are declared by *functor declarations* and applied by *functor applications*. Unlike functions, functors are always named. The declaration of a single functor *funid* in its bare form is written

$$\texttt{functor}\ \mathit{funid}\ \texttt{(}\ \mathit{strid}\texttt{:}\ \mathit{sigexp}\ \texttt{)}\langle\texttt{:}\ \mathit{sigexp}'\rangle\ \texttt{=}\ \mathit{strexp} \qquad (*)$$

where the structure identifier *strid* is the *formal argument* (of *funid*), *sigexp* is the *argument signature* and *strexp* is the *body* of the functor. When present, *sigexp*′ is the *result signature* of *funid*.

The bare form of functor application is the structure expression

$$\mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)}$$

where *strexp* is the *actual argument* to *funid*.[15]

In many ways functors resemble functions. For example, a functor declaration is elaborated once, namely when the functor is declared, in the basis in force at the time of declaration.

Suppose that the functor declaration $(*)$ is to be evaluated in a basis B. What should be the effect when later

$$\mathit{funid}\ \texttt{(}\ \mathit{strexp}'\ \texttt{)}$$

is evaluated in another basis B'? The answer is quite natural, now that we have discussed interfaces: Evaluate *strexp*′ in B', yielding E'; cut E' down according to the argument signature *sigexp*, and bind *strid* to it; evaluate *strexp* in B augmented by this new binding, yielding E; finally cut E down according to the result signature *sigexp*′ if present.

So, by analogy with function closures [Sec 6.6, p 49], the evaluation of $(*)$ results in binding *funid* to a *functor closure* $(\mathit{strid}: I, \mathit{strexp}\langle: I'\rangle, B)$ [Fig 14, p 57]. The B component is the basis in which the functor is declared; this basis is used when the functor is applied in accordance with the principle that functors, like functions, are statically scoped. Similarly, I and the optional I' are the results of evaluating

[15] For the full grammar for functor declarations and applications, see [Fig 6, p 12], [Fig 8, p 14] and the derived forms in [Fig 18, p 68], which provide convenient syntax to make it look as if functors can also take values, exceptions and types as arguments.

sigexp and *sigexp′* at declaration time (i.e. in the basis B). It is helpful to see rule 187 for functor bindings [p 61] without its second option:

$$\frac{\text{Inter}\, B \vdash \mathit{sigexp} \Rightarrow I \qquad \langle \text{Inter}\, B + \{\mathit{strid} \mapsto I\} \vdash \mathit{sigexp}' \Rightarrow I' \rangle}{\begin{array}{c} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\texttt{:}\ \mathit{sigexp}\ \texttt{)}\langle\texttt{:}\ \mathit{sigexp}'\rangle\ \texttt{=}\ \mathit{strexp} \Rightarrow \\ \{\mathit{funid} \mapsto (\mathit{strid}\texttt{:}\, I, \mathit{strexp}\langle\texttt{:}\, I'\rangle, B)\} \end{array}}$$

Note in particular that *sigexp′* can be evaluated at declaration time, even though it may refer to *strid* for which no actual parameter structure is presently available; this is because the only knowledge of the actual parameter it needs is its shape – and this shape I is provided by the evaluation of *sigexp*. We can now see how the functor closure provides the right information, and no more, for the functor to be applied to an actual structure using rule 162:

$$\frac{\begin{array}{c} B(\mathit{funid}) = (\mathit{strid}\texttt{:}\, I, \mathit{strexp}'\langle\texttt{:}\, I'\rangle, B') \\ B \vdash \mathit{strexp} \Rightarrow E \qquad B' + \{\mathit{strid} \mapsto E \downarrow I\} \vdash \mathit{strexp}' \Rightarrow E' \end{array}}{B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow E'\langle\downarrow I'\rangle} \tag{162}$$

It is interesting to note that signature ascription – at least in a top-level structure declaration – can be defined in terms of functor application. For, given any *sigexp*, we can define the corresponding "curtailing" functor:

```
functor F(X: sigexp) = X
```

Thereafter, wherever we write

```
structure A: sigexp  =  strexp
```

we can get exactly the same effect by writing instead

```
structure A = F(strexp)
```

Exercise 3.3 Assume that *sigexp* evaluates to the interface I. Let B be the basis resulting from the declaration of `F`. What is the functor closure bound to `F` in B? Now suppose that *strexp* evaluates in B to the environment E. What structure is bound to `A` after each of the above structure declarations?

Of course, to justify the claim that two phrases are equivalent we have to show that they have the same static semantics too. This is true here; in particular, elaboration will fail for the signature ascription in exactly the same cases as it will fail the functor application.

3.5 Alternative semantics

To put the treatment of signatures in perspective, we finish this chapter by looking at an alternative meaning for signature ascription. One could decide that signature ascription should have no curtailing effect, but should merely act as a conformity test (we defined conformity above). Recalling that conformity will always be ensured by elaboration, we might then expect that interfaces would be unnecessary, and that we could indeed take the simple course of ignoring all mention of signatures as far as evaluation is concerned. Certainly this would work for the example in Section 3.3 above. But if the argument signature of a functor were to have no curtailing effect, then we would meet a strange phenomenon in using the `open` declaration inside a functor. Consider

```
functor F(A: sig end) =
  struct open A
         val y = x
  end
```

What will happen in an application `F (`*strexp*`)` ? If (the structure denoted by) *strexp* has a value component named `x`, then the `y` component of the result structure will have the value of this `x`; otherwise it will have the value of some other `x` non-local to `F`. But this dynamic variation of variable reference is foreign to ML. Indeed, it would cause severe difficulty in elaboration; intuitively, the type checker would not know which `x` was meant. So a language which adopts the alternative semantics can take two courses. Either it can give a curtailing effect to a signature when it is ascribed to a functor *argument*, but not elsewhere; or it can change the `open` declaration to mention a signature explicitly, e.g. `open` *strid* `to` *sigexp* , and give a curtailing effect only to signatures in this `open` context. Whichever choice is made, it does not seem to be possible to eliminate interfaces from evaluation.

This alternative semantics, though not chosen, was a serious candidate for ML. It has considerable utility. For example, if the argument signature has no curtailing effect, one can define the *identity functor*

```
functor Id(X: sig end) = X
```

since the empty signature will be matched by every structure. In a future language, we may be able to get the best of both worlds by including curtailment as a separate operation, rather than as an effect of signature ascription.

Exercise 3.4 How would you alter the dynamic semantics, if signature ascriptions were merely to act as conformity tests when ascribed to structure bindings or to the result in a functor binding, but were to retain their curtailing effect in functor arguments?

4 Static Semantics for the Core

This chapter and the next concern the static semantics for the Core [Sec 4]. In Chapter 1 we saw roughly how elaboration works; here we begin to discuss its finer details. In particular, in this chapter we discuss types versus type schemes, the closure operation on types and environments, explicit type variables, polymorphic references and exceptions.

It is good to begin with an exercise:

Exercise 4.1 Construct an elaboration tree for the atomic expression

```
{mother = "Elisabeth", age = 27}
```

starting from an arbitrary context C. Use the index of the Definition to find out – or recall – the meaning of common notations such as $\langle\ \rangle$ and $+$.

While doing the above exercise, the reader will probably have noticed the difference between writing, for example `{mother = "Elisabeth"}` and {`mother` $\mapsto$ `string`}. The former is a phrase in ML but the latter denotes a finite map. We are free write either {`mother` $\mapsto$ `string`, `age` $\mapsto$ `int`} or {`age` $\mapsto$ `int`, `mother` $\mapsto$ `string`} for they denote the same map. One is not always free to permute the fields of an expression row, since they are evaluated from left to right and this can be significant when the expressions have side-effects.

4.1 Contexts, environments and scope

The Definition does not contain an explicit definition of the *scope* of identifiers (except for certain occurrences of explicit type variables, cf. [Sec 4.6]). However, the scope rules of the language are implicitly defined by the elaboration rules. Let *program* be a program and let o be an occurrence of some phrase *phrase* occurring in *program*; also let P be an elaboration tree whose conclusion is $B \vdash \mathit{program} \Rightarrow B'$, for some B, B'. Corresponding to o there will be precisely one occurrence in P of a sentence of the form $C \vdash \mathit{phrase} \Rightarrow A$, for some context C and semantic object A. Moreover, the identifiers that are in scope at the occurrence o are precisely the identifiers that can be accessed in C, although of course not all such identifiers need occur free in *phrase*.

A context C is a triple T, U, E [Fig 10, p 17]. Here T is a set of type names, the meaning of which will be explained later. U is a set of type variables; intuitively speaking, it contains those explicit type variables that are in scope at the occurrence o. Finally, E is an *environment* (SE, TE, VE, EE) consisting of four components: a *structure environment* (SE), a *type environment* (TE), a *variable environment* (VE) and an *exception environment* (EE) [Fig 10, p 17]. A (perhaps long) identifier of either of these four kinds is in scope precisely if it can be looked up in E.

New environment components *TE*, *VE* and *EE* are created by the rules for bindings, rules 26–32 [p 26,27]; these are converted into environments E by the rules for declarations, rules 17–25 [p 25,26]. Environments thus built from declarations are *increments*, to be incorporated later in a context C – e.g. by rule 22.

There are just two operations for modifying the context, namely the + and the $\oplus$ operations defined in [Sec 4.3, p 18]. $C + E$ is C extended with the bindings of E; thus $(C + E)(id)$ is $E(id)$, if this is defined, and $C(id)$ otherwise. $C \oplus E$ is the same as $C + E$ except for its T-component; in fact T of $(C + E) = T$ of C, while T of $(C \oplus E) = T$ of $C \cup$ tynames E [Sec 4.3, p 18]. A typical use of $\oplus$ is in rule 6 (which is displayed below); it ensures that any types declared inside *exp* will be kept distinct not only from those in C but also from those declared in *dec*. This point is discussed further in Section 5.1.

The rules that are interesting to look at, as far as scope is concerned, are those where different contexts occur before the turnstile. The following two rules are representative.

- Rule 6, the rule for `let` expressions:

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau}$$

  Loosely speaking, the scope of the identifiers declared by *dec* is *exp*.

- Rule 27, the rule for recursive value bindings:

$$\frac{C + VE \vdash valbind \Rightarrow VE}{C \vdash \texttt{rec}\ valbind \Rightarrow VE}$$

  Notice the *VE* on the left-hand side of the turnstile. It expresses that any variable declared by *valbind* is itself in scope within *valbind*.
  Example: `rec fac = fn n => if n=0 then 1 else n * fac(n-1)`.

4.2 Types and type schemes

The purpose of this section is to clarify the distinction between types and type schemes, a distinction which may at first seem technical but in fact is essential for an understanding of what one can and cannot do with ML polymorphism.

The simplest kind of types are *monotypes*, i.e. types that contain no type variables. An example of a monotyped function is the successor function `fn x => x+1` which has type `int` $\rightarrow$ `int`. In general, types can contain type variables; then they may be called *polytypes*. To obtain a *type scheme* σ, one can take a type τ and universally quantify zero or more type variables; the result is written in the form $\forall\alpha^{(k)}.\tau$, where $\alpha^{(k)}$ is a tuple of k distinct type variables ($k \geq 0$). (Note the difference between `'a`, which *is* a type variable, and α which *ranges* over type variables.)

The quantified variables are often referred to as the *bound* variables of σ.[16] Also, τ is sometimes referred to as the *body* of σ. An example of a function which has a type scheme is

```
fun end_cons [] x = [x]
  | end_cons (y::ys) x = y :: end_cons ys x
```

Its type scheme is $\forall$`'a.'a list`$\rightarrow$`'a`$\rightarrow$`'a list`. This type scheme is *closed*, i.e. of the form $\forall\alpha^{(k)}.\tau$ where every type variable occurring in τ occurs in $\alpha^{(k)}$. All values declared by a top-level declaration will have a closed type scheme (even the successor function does, namely $\forall()$.`int`$\rightarrow$`int`).

A type scheme can be regarded as a compact notation for a perhaps infinite set of types. For instance, the type scheme for `end_cons` expresses that the function has type τ `list`$\rightarrow\tau\rightarrow\tau$ `list`, for any type τ. Thus we are permitted to use `end_cons` first with $\tau =$ `int list` and then with $\tau =$ `bool list`, as in

```
(end_cons [5,7] 9, end_cons [true,false] true)
```

A type scheme σ *generalises* a type τ, written $\sigma \succ \tau$, if τ can be obtained from the body of σ by substituting types for the bound type variables of σ (see [Sec 4.5, p 19] for the precise definition).[17] Notice that generalisation acts on bound type variables only so for example $\forall()$.`'a`$\rightarrow$`'a` only generalises one type, namely `'a`$\rightarrow$`'a`.

While the inference rules allow one to instantiate bound type variables in certain places, they never allow one to substitute types for free type variables. Hence, if some variable `f` has been inferred to have the type scheme $\forall()$.`'a`$\rightarrow$`'a` we cannot in that context apply `f` to an integer, for `'a` and `int` are different.

The more closed a type scheme of a variable is, the more usages of that variable become typable. It turns out that one cannot simply close all types completely without destroying the soundness of the type inference system. The precise transition from types to type schemes is defined by the Clos operation ([Sec 4.8, p 21]), which we will return to later on. However, a few basic facts about elaboration give an overview of the power of polymorphism in ML. Let us say that a type scheme is *simple* if it has no bound type variables, and *general* if it is not simple.

An important point is that any type can be regarded as a (simple) type scheme, but only simple type schemes can be regarded as types. This is a consequence of the definition of the set Type [Sec 4.2, p 17], which does not admit inner quantification in types. For example

$$(\forall\texttt{'a.'a}\rightarrow\texttt{'a})\rightarrow\texttt{int}$$

[16] In the literature these are also referred to as *generic* type variables.

[17] In the literature one also sees the terminology "τ is a *(generic) instance* of σ" for "σ generalises τ".

is neither a type nor a type scheme in ML. Consequently, the parameter to a function cannot have a general type scheme; this means that a formal parameter of a function must have the same type at each of its occurrences within the function body.

A variable environment *VE* maps variables to type schemes (as opposed to types) [Fig 10, p 17]. However, in some cases the variable environments that occur in the elaboration rules always contain simple type schemes only:

Theorem 4.1 *If* $C \vdash pat \Rightarrow (VE, \tau)$ *or* $C \vdash valbind \Rightarrow VE$ *then* $VE(var)$ *is simple, for all* $var \in \mathrm{Dom}(VE)$.

Proof An easy induction on the depth of the elaboration tree, applied to the inference rules 26, 27 and 33–46. The base cases include the case of rule 35, from which we see that a variable occurring as an atomic pattern always elaborates to a simple type scheme.

This result immediately reveals the language constructs that bind variables to simple type schemes only. Rule 16 [p 25] thus tells us that any variable bound in a *match* (be it in a `fn` or a `handle` expression) must have a simple type scheme.

Exercise 4.2 Which of the following expressions elaborate?

(1) `fn (length, s, n) => 1 + n + length s`
(2) `fn (length, s) => 1 + length s + length [1,3]`
(3) `fn (length, s) => 1 + length s + length [1,3] + length [true]`
(4) `case s of x as [ ] => 0::x::x | other => [0]::other`

A recursive function can never be used as a polymorphic function within its own body. This is seen from rule 27,

$$\frac{C + VE \vdash valbind \Rightarrow VE}{C \vdash \texttt{rec}\ \ valbind \Rightarrow VE}$$

for by the theorem *VE* contains simple type schemes only.

Exercise 4.3 Which of the following declarations elaborate?

(1)
```
fun g [ ] = 1
  | g (x::xs) = 1 + g(xs)
```
(2)
```
fun h [ ] = 1
  | h (x::xs) = h(xs) + h [7,9]
```
(3)
```
fun k [ ] = 1
  | k (x::xs) = 1 + k [7,9] + k [true]
```

4.3 Closure of variable environments

We saw in the previous section that the power of polymorphism arises from general type schemes rather than types. As can be seen from rules 9–14, all expressions elaborate to types, not type schemes. However, types can be turned into type schemes by the closure operation Clos defined in [Sec 4.8, p 21]. The act of forming a type scheme $\forall\alpha^{(k)}.\tau$ from a type τ is known as *quantification*. In the present section we shall discuss the closure operation in some detail (matters related to polymorphic references and exceptions are deferred to Section 4.5).

Value variables are bound to type schemes at precisely one point in the rules, namely in rule 17:

$$\frac{C + U \vdash \mathit{valbind} \Rightarrow VE \quad VE' = \mathrm{Clos}_{C,\mathit{valbind}} VE \quad U \cap \mathrm{tyvars}\, VE' = \emptyset}{C \vdash \mathtt{val}_U\ \mathit{valbind} \Rightarrow VE \text{ in Env}}$$

Here VE is a variable environment mapping variables to simple type schemes, and VE' maps the same variables to type schemes that are obtained from VE by closing every type τ in the range of VE to get a type scheme $\forall\alpha^{(k)}.\tau$, $k \geq 0$, where the $\alpha^{(k)}$ is determined by the Clos operation.

Example 4.1 Consider the expression

```
fn x => let val g = (fn y => y x) in g end
```

It is of course equivalent to `fn x => (fn y => y x)` , but we wish to illustrate the closure operation in elaborating the `val` declaration. If the expression is elaborated in the initial context $C_0 = C$ of B_0, then the `val` declaration will be elaborated in $C = C_0 + \{\mathtt{x} \mapsto \mathtt{'a}\}$, where `'a` is a free type variable. Here is the relevant part of the elaboration:

$$\vdots$$

$$C \vdash \boxed{\mathtt{g = (fn\ y => y\ x)}} \Rightarrow \underbrace{\{\mathtt{g} \mapsto (\mathtt{'a} \to \mathtt{'b}) \to \mathtt{'b}\}}_{VE}$$

$$\Big\downarrow\ (17)$$

$$C \vdash \boxed{\mathtt{val\ g = (fn\ y => y\ x)}} \Rightarrow \overbrace{\{\mathtt{g} \mapsto \forall\mathtt{'b}.(\mathtt{'a} \to \mathtt{'b}) \to \mathtt{'b}\}}^{\mathrm{Clos}_C VE} \text{ in Env}$$

Note how closure has quantified `'b`, but not `'a` since it is free in C.

One cannot in general simply quantify all the type variables that occur free in VE. This would lead to unsound elaborations.

Exercise 4.4 Assuming that we naïvely obtain VE' from VE by closing VE completely, show that, for any context C and type τ, we would be able to elaborate the following nonsensical expression to τ in C:

$$e_0 = \begin{cases} \texttt{let val f = fn x =>} \underbrace{\texttt{let val y = x in y 5 end}}_{e_1} \\ \texttt{in f 3} \\ \texttt{end} \end{cases}$$

Exercise 4.4 illustrates that a type variable which occurs in VE can be quantified only if it does not occur free in C. This is ensured by the definition of Clos in [Sec 4.8, p 21].

However, because of polymorphic references and exceptions the converse is not true. That is, sometimes it is unsound to quantify a type variable which occurs in VE even though it does not occur free in C. This is explained in more detail in Section 4.5.

Exception declarations never give rise to quantification of type variables. For example,

```
exception e of '_a
```

does not mean that `e` has type $\forall$`'_a.'_a` $\rightarrow$ `exn`. This would in fact destroy the soundness of the type inference system.

Example 4.2 Consider for example the expression

```
let exception e of '_a
in (raise e(5)) handle e(f) => f(7)
end
```

Were `e` to get type $\forall$`'_a.'_a` $\rightarrow$ `exn`, the above expression would elaborate despite the fact that, if evaluated, it would attempt to use `5` as a function.

Therefore, although the type of an exception can contain type variables, a static exception environment EE maps exception constructors to types, not type schemes [Sec 4.2, p 17].

4.4 Explicit type variables

Explicit type variables are scoped according to the syntactical rules given in [Sec 4.6, p 20]. These rules define how one can "decorate" every occurrence of a value declaration `val` *valbind* by a set U of type variables, namely the set of type variables that are *scoped* at that occurrence. Hence the subscript U in rule 17.

The one principle that programmers need to keep in mind is that in any well-typed program, the value declaration at which an explicit type variable is syntactically scoped is precisely the point where it becomes quantified. Thus the inference rules must ensure that

(a) if a type variable α is explicitly scoped at some occurrence o of $\texttt{val}_U$ *valbind* then α really can be quantified at o

(b) if α is a type variable that occurs unguarded in *valbind* and α is quantified by the closure operation at o, then α really is scoped at o (i.e. α does not occur unguarded in any larger value declaration containing o).

Neither of these conditions is automatically met, without the presence of a side-condition in rule 17. The first would be violated by the expression

$$\texttt{(fn x => let val}_{\{\texttt{'a}\}}\texttt{ y :'a = x in y y end) 5}$$

which would elaborate were we to allow quantification of `'a` at the point it is syntactically scoped, although of course evaluating this expression would lead to a sad attempt to apply 5 to itself. Therefore, rule 17 contains the side-condition $U \cap \text{tyvars}\, VE' = \emptyset$; it ensures that, once the closure operation has been performed, none of the explicit type variables scoped at that occurrence has been left free.

The second condition (b) prevents us from quantifying α if it is scoped further out, even in cases where quantification would be perfectly sound. An example is

```
val{'a} x = (let val∅ Id :'a -> 'a = fn z=>z in Id Id end,
             fn z => z :'a)
```

This does not elaborate although it would have been semantically sound to quantify `'a` at the inner value declaration. To prevent quantification in such cases, the context C contains a component U, which is (so to speak) the set of explicit type variables scoped further out. As these now occur free in C, the closure operation will not quantify them.

Exercise 4.5 In the above declaration, what happens if we remove the type constraint from `Id`? What happens if we instead remove the constraint from `z`?

4.5 Polymorphic references and exceptions

To ensure the soundness of the type inference rules in the presence of references and exceptions requires certain constraints on the rules. We shall first explain why that is so and then explain the particular constraints adopted in the Definition.

Let us first show that the type inference rules which work for a purely functional language break down in the presence of references. To this end, assume that the function `ref`, which creates a new reference, has the type $\forall$`'a.'a`$\rightarrow$`'a ref`, that the assignment operator, `:=`, has type $\forall$`'a.'a ref * 'a`$\rightarrow$`unit` and that `!`, the de-referencing function, has type $\forall$`'a.'a ref`$\rightarrow$`'a`. Now consider the following expression *exp*:

```
let val r = ref [ ]
in r:= [7]; !r
end
```

```
let val (e',f) =
        let exception e of 'a
         in (e, fn g => g() handle e(x) => x)
        end
    val X = f(fn () => raise e'(7))
 in
    X(2)+3
 end
```

Figure 5: Unsound use of polymorphic exceptions

This expression evaluates to the value `[7]` so it should elaborate to one and just one type, namely `int list`. Nevertheless, if we use unconstrained polymorphic type inference, then in fact *exp* elaborates to the type τ `list`, for *any* type τ. In particular, it elaborates to `bool list`, so we will be able to infer a type for the expression

`map not` *exp*

even though evaluation of this expression presumably would lead to a run-time error.

The important parts of the elaboration tree by which one "proves" that *exp* elaborates to τ `list`, are shown in Figure 6. We assume that C is closed, i.e. contains no free type variables. We write C' for $C + \{\texttt{r} \mapsto \forall\texttt{'a.'a list ref}\}$.

The crucial step is the one at **A**, where we quantify `'a` on the grounds that it does not occur free in C. Then the type scheme $\forall$`'a.'a list ref` can be instantiated as shown at **B** and **C**, leading to the false conclusion.

But why is it wrong to quantify `'a` in this example? To understand this, we need to note that when references are values, the type of a value depends not just on the types of the values in the context C but also on the types of the values in the memory. If the memory contains the value `[7]` at some address, a, then then type of a must be `int list ref`.

Imagine a dynamic environment whose values have been tagged with types according to the static context. Moreover, we could type the memory by a *memory typing* which maps every address in the memory to the type of the value it contains. Thus a type variable may occur free in the static context or the memory typing. In either case it would be wrong to quantify `'a`. *But* `'a` *may occur free in the memory typing without occurring free in the context.* In such a case, closing with respect to the context alone will result in an unsound quantification. This is precisely what happened in our example above. The expression `ref [ ]` evaluates to an address a to which we can ascribe the type `'a list ref`, since we can elaborate the expression `ref [ ]` to that type. Thus the memory typing after the creation of the reference will contain `'a` free, and it becomes unsound to quantify `'a`.

Similar observations apply to exceptions. Recall that a *state* consists of a memory and a set of exception names [Fig 13, p 47]; so a *state typing* would consist of a memory typing and an exception typing. If we type the exception names as they are generated during evaluation, there is still a place where type variables can occur free.

> In Example 4.2 we showed why the type scheme to which an exception constructor is bound is always simple. We now illustrate how, even so, to achieve soundness special care must be taken when any type variable occurring in this simple type scheme is quantified by the elaboration of some outer `val` declaration.
>
> Consider the expression in Figure 5. By quantifying type variables at the first value declaration, `e'` gets type scheme $\forall$`'a.'a` $\rightarrow$ `exn` and `f` gets type scheme $\forall$`'a.(unit` $\rightarrow$ `'a)` $\rightarrow$ `'a`. This makes it possible to give `X` whatever type we like, for example `int` $\rightarrow$ `int`. However, `X` turns out to be `7`, which of course is not a function.

How can we prevent these faulty quantifications? Unfortunately, we cannot type addresses and exception names statically, for they are generated dynamically. In other words, one cannot elaborate state typings. But we can use elaboration to distinguish between the type variables that *may* occur in a state typing and type variables that *will definitely not* occur free in the state typing.[18] Corresponding to these two possibilities we have two syntactically disjoint classes of type variables, namely the *imperative* and the *applicative* type variables [Sec 2.4, p 4].

Suppose that we have an elaboration with conclusion

$$C \vdash exp \Rightarrow \tau$$

for some *exp*, C and τ and that we want to find out whether it is sound to quantify some type variable α which occurs free in τ. There are two cases:

α is applicative. Then α will definitely not occur free in the state typing, since state typings by definition cannot contain applicative type variables. Therefore, the usual rules for quantification apply.

α is imperative. If we are certain that the evaluation of *exp* generates neither an address nor an exception name, then the set of type variables occurring in the state typing is not increased by the evaluation of *exp*, so again the rules for the purely applicative language apply. If, however, the evaluation may generate a new reference or exception then we disallow quantification of α, because α might be in the state typing after the evaluation.

To avoid sophisticated analysis of precisely when the evaluation of *exp* may generate new addresses or exceptions, we simply define that *exp* is *non-expansive* (meaning that the evaluation of *exp* will definitely not create a reference or an exception)

[18] It is to be expected that elaboration only gives an approximation of the set of the type variables that do definitely not occur free in the state typing, but the approximation is safe as long as it does not claim, for some type variable that can actually occur in the state typing, that it will definitely not so occur.

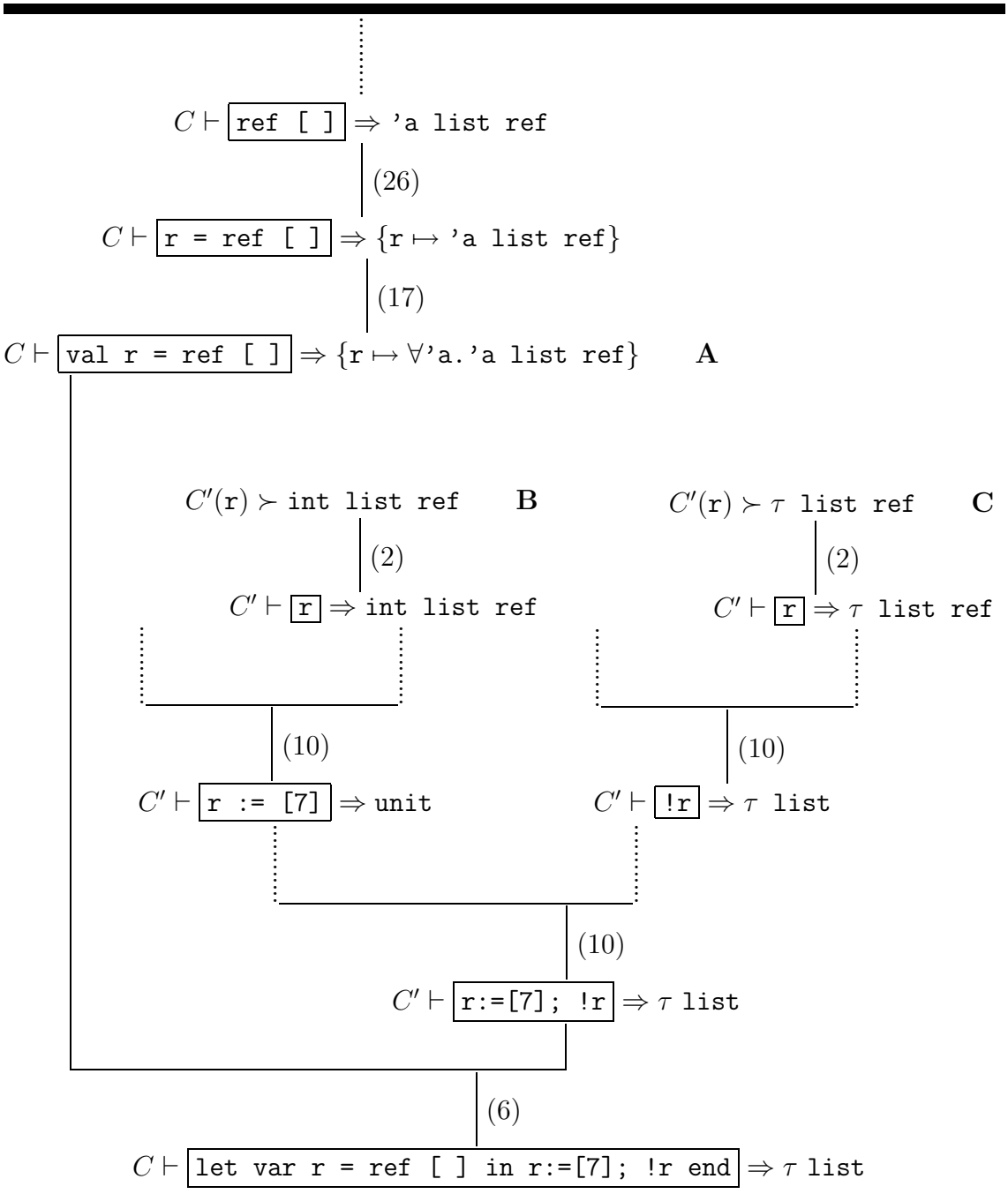

Figure 6: Unsound type inference with references

if it is a variable, a constructor, or a `fn` expression, possibly constrained by one or more type expressions. All other expressions are called *expansive* [Sec 4.7, p 20]. Of course, a wider class of expressions could easily be defined as non-expansive, but we have chosen a simple class which should be easy to remember.

To sum up, the rules for quantification are as for the purely applicative language, except that quantification is forbidden in the case that *exp* is expansive and α is imperative.

This scheme does not affect programs that do not use references and use monotyped exceptions only. But in general, imperative type variables enter elaboration in two ways. First, the type of the constructor `ref` is $\forall$`'_a.'_a`$\rightarrow$`'_a ref` [App C, p 75]. (The types of `!` and `:=` do not contain imperative type variables, since they cannot extend the state typing with new type variables.) Second, applicative type variables are banned from the types of exceptions [rule 31, p 27].

Finally, free imperative type variables are not allowed to propagate to the top-level [rules 100–102, p 44] so that all type variables that are reported to the user at top-level are implicitly quantified. For example `val r = ref [ ];` is not a valid program.

Example 4.3 The following tail recursive implementation of a function for fast reversal of long lists

```
fun reverse l =
  let val res = ref [ ]
      fun loop [ ] = !res
        | loop (hd::tl) =
            (res:= hd::(!res); loop tl)
   in loop l
  end
```

has type $\forall$`'_a.'_a list` $\rightarrow$ `'_a list`. (The underbar (_) in `'_a` indicates that `'_a` is imperative [Sec 2.4, p 4].)

Exercise 4.6 Mr Reno Vator, objecting to the garbage created by `reverse` (one new reference for each call), suggested the following variant

```
val reverse =
  let val res = ref [ ]
   in fn l =>
    (res:= [ ];
     let fun loop [ ] = ! res
           | loop (hd::tl) =
              (res:= hd::(!res); loop tl)
      in loop l
     end)
  end
```

but he soon had second thoughts. Why?

Example 4.4 The declaration

```
fun Id x = let val r = ref x in !r end
```

elaborates to $\{\texttt{Id} \mapsto \forall\texttt{'\_a.'\_a} \to \texttt{'\_a}\}$. Given this declaration of `Id` we have that the declaration `val Id' = Id Id` elaborates to $\{\texttt{Id'} \mapsto \tau \to \tau\}$ for any imperative type τ, but the same declaration does not elaborate to $\{\texttt{Id'} \mapsto \forall\texttt{'\_a.'\_a} \to \texttt{'\_a}\}$ since the expression `Id Id` is expansive. However, we do have that

$$C \vdash \texttt{fun Id' x = Id Id x} \Rightarrow \{\texttt{Id'} \mapsto \forall\texttt{'\_a.'\_a} \to \texttt{'\_a}\}$$

because `fn x=> Id Id x` is non-expansive.

This last trick, known as *η-conversion*, is often very useful to obtain polymorphism when using references. However, there is no way of getting rid of the imperative type variables altogether. This can cause some inconvenience in connection with Modules. The type scheme $\forall\texttt{'a.'a} \to \texttt{'a}$ generalises the type scheme $\forall\texttt{'\_a.'\_a} \to \texttt{'\_a}$, *but not the other way around*. Thus a function which has the latter type cannot match a value specification with the former type. Consequently, one will have to modify the signature containing the specification if one insists on using side-effects in the implementation.

5 Type Declarations and Principality

In this chapter we continue discussing the static semantics of the Core. We consider type and datatype declarations, type functions and type structures, types which admit the equality predicate, and the notion of principal or "most general" environment.

5.1 Types, datatypes and type functions

It is important to distinguish between *type constructors* and *type names*. The former are identifiers which can denote the latter. Two type constructors that in some context are bound to the same type name can be used interchangeably in that context. A datatype declaration creates "new" datatypes in the sense that the type constructors declared by the declaration are bound to type names which, in a sense which is made precise by the semantics, are distinct and fresh.

Example 5.1 Consider the following declaration

```
datatype 'a tree = LEAF of 'a
                 | TIP of 'a * 'a tree * 'a tree
type 'a heap = 'a tree
type intheap = int heap
```

It introduces three type constructors, namely `tree`, `heap`, and `intheap`, but it only generates one type name, `t` say, which is bound first to `tree` and then to `heap`.

Every type name t has an *arity* $k \geq 0$, and also possesses an equality attribute. For example, the arity of `int` is 0 and the arity of `list` is 1. See Section 5.2 for discussion of equality attributes.

It is not in general enough to bind type constructors simply to type names. For a datatype constructor, we need also to know its value constructors and their types. (This is necessary in order to determine whether the datatype admits equality [Sec 4.9, p 21]; it is also necessary in order to enable checking of consistency [Sec 5.2, p 32].) Moreover, for a type constructor which is not a datatype constructor, the semantic value may in general be a function from types to types, rather than just a type name. Such a function is called a *type function*; θ is used to range over type functions [Fig 10, p 17]. For example, consider the declaration

```
type 'a pair = 'a * 'a
```

Here θ is

$$\Lambda\texttt{'a.'a} * \texttt{'a}$$

The capital Λ is used rather than the more conventional λ as a reminder that type functions map types to types, not values to values. The general form of a type function is

$$\Lambda\alpha^{(k)}.\tau$$

where τ as usual is a type, k is the *arity* of the type function and $\alpha^{(k)}$ is a sequence of k type variables.

All type constructors, regardless of whether they are introduced by `datatype`, `abstype` or `type` declarations, are mapped to the same kind of semantic object, called a *type structure*. A type structure is a pair

$$(\theta, CE)$$

of a type function and a constructor environment [Fig 10, p 17]. In the case of a `type` declaration, CE is always the empty map. In the case of a `datatype` declaration, CE is always non-empty and θ is of the particular form

$$\Lambda(\alpha_1, \cdots, \alpha_k).(\alpha_1, \cdots, \alpha_k)\ t$$

for some type name t with arity k and some type variables $\alpha_1, \ldots, \alpha_k$. Such a special type function is identified with the type name t [Sec 4.4, p 19], so that we can think of the semantic value of a datatype as being a pair

$$(t, CE), \qquad CE \neq \{\}$$

In the case of a `datatype` declaration which declares constructors $con_1, \ldots, con_n$, the non-empty constructor environment is of the form

$$\{con_1 \mapsto \sigma_1, \ldots, con_n \mapsto \sigma_n\}$$

where σ_i is either t, if con_i is a constant, or otherwise of the form $\forall\alpha^{(k)}.(\tau \rightarrow \alpha^{(k)}t)$.

Let us say that two type structures *share* if they have the same type function (up to renaming of the bound type variables). Two such type structures need not have identical constructor environments, but if both are non-empty then they must at least have the same domain (cf. the definition of consistency, [Sec 5.2, p 32]). The type structures bound to `tree` and `heap` in the example above share but they are different: the former has a non-empty constructor environment, but the latter has an empty constructor environment.

A *type environment*, TE, is a finite map from type constructors to type structures [Fig 10, p 17]. Elaboration of type bindings and datatype bindings produces type environments, see rule 28 and 29 [p 27]. We can now see what it means to create a "new" datatype. A context C contains a component, T, which is a set of type names [Fig 10, p 17]. Intuitively, T is the set of type names that are already used; we shall call them the *rigid* type names. A new datatype is created by choosing a type name t which is not a member of T. It will always be the case that

any type name occurring in the other components of C is in T (so that T records at least all the currently accessible type names) but by keeping T explicitly in the basis we make sure that the chosen t is new also with respect to type names that (perhaps temporarily) have disappeared from the context. This explains the side-condition $\forall(t, CE) \in \operatorname{Ran} TE, t \notin T$ of C, found in rules 19 and 20 [p 25].

Exercise 5.1 Consider the declaration from Example 5.1. Write down type structures to which the type constructors are bound as a result of elaboration. (Ignore the side-condition of rule 19 that concerns equality.)

We have already seen that a variable environment VE maps variables to type schemes. In addition, variable environments map value constructors and exception constructors to their types [Fig 10, p 17]. A *datatype* declaration elaborates to a pair (VE, TE) [rule 19, p 25], where VE subsequently can be added to the context, for example with rule 25 [p 26]. (The reason for including constructors in the variable environment, as well as in the type environment, is explained in Section 2.7. For the purpose of +, an identifier which is a constructor is not distinguished from the same identifier used as a variable, so a binding of a constructor can overwrite a binding of a variable.) There is no requirement that a constructor can occur in at most one datatype declaration. For example, the declaration

```
datatype options = YES | NO
datatype positive = YES
datatype options = MAYBE
```

does elaborate.

Exercise 5.2 To what? After these declarations, which of the following expressions elaborate? (a) `YES`, (b) `NO`, (c) `if true then YES else NO`.

5.2 Equality

In the dynamic semantics the equality predicate can be applied to special values (i.e. values denoted by special constants [Sec 6.2, p 46]), constants (i.e. nullary constructors), and addresses. Moreover, equality can be applied to constructed values if it can be applied to the values out of which they are constructed, and to records provided it can be applied to the component values. Equality cannot be applied in any other cases; in particular, one cannot test closures (i.e. function values) for equality.

In ML, it is impossible to elaborate an invalid application of the equality predicate, i.e. type-correct programs can never contain such invalid applications. The subclass EtyVar of TyVar, the *equality type variables*, consists of those type variables that start with two or more primes [Sec 2.4, p 4]. An equality type variable ranges over types that admit equality. For example, the type scheme of

`=` is $\forall$`''a.''a * ''a` $\rightarrow$ `bool`. Moreover, every type name possesses an equality attribute [Sec 4.1, p 16]. For example, the equality attribute of `list` is true. This means that *if* a type τ admits equality *then* τ `list` admits equality. The notion of *equality type* is defined in [Sec 4.4, pp 18–19].

Exercise 5.3 Is equality defined on exceptions (i.e. does the type `exn` admit equality)?

Exercise 5.4 Which of the following expressions elaborate?

```
(1)  {a = [ ], b = false} = {a = [1,2], b = true}
(2)  (fn x => x + 1) = succ
(3)  ref(fn x => x + 1) = ref(fn x => x + 1)
```

We shall now discuss how the equality properties of type constructors declared by `type`, `datatype` or `abstype` are determined.

A `type` declaration

`type` *typbind*

introduces no new type names, so the equality attribute of each type constructor so declared is a function of the equality attributes of the type names in terms of which it is declared. More precisely, recall from Section 5.1 that every type constructor will be bound to a type structure of the form $(\theta, \{\})$, where the type function θ is of the form $\Lambda\alpha^{(k)}.\tau$. By definition, θ admits equality if τ admits equality when the type variables $\alpha^{(k)}$ have been chosen to be equality type variables [Sec 4.4, p 19].

Exercise 5.5 Which of (the type functions corresponding to) the following type constructors admit equality?

```
(1)  type 'a intmap = 'a -> int
(2)   and 'a pair = 'a * 'a
(3)  type 'a intmap_store = 'a intmap list ref
```

Exercise 5.6 What is the difference between the following two declarations?

```
(1)  type 'a pair = 'a * 'a
(2)  type ''a pair = ''a * ''a
```

However, a declaration of the form

`datatype` *datbind*

or

`abstype` *datbind* `with` *dec* `end`

does introduce new type names, one for each type constructor it declares. We shall now explain how the equality attributes of these new type names are determined.

First, recall from Section 5.1 that the elaboration of *datbind* results in a type environment where every type constructor is bound to a type structure of the special form (t, CE), where t is a type name and CE is a non-empty constructor environment. Now the first criterion for t to admit equality is that the arguments to which each constructor *con* in CE is applied will always be of a type which admits equality. If a type structure (t, CE) satisfies this criterion, it is said to *respect equality*; formally, this means that if t admits equality, then either $t =$ `ref`[19] or, for each $CE(con)$ of the form $\forall\alpha^{(k)}.(\tau \to \alpha^{(k)}t)$, the type function $\Lambda\alpha^{(k)}.\tau$ also admits equality. Further, a type environment *TE respects equality* if all its type structures do so [Sec 4.9, p 21].

But it is not enough simply to demand that each type structure should respect equality. Consider

```
datatype T = C of T | D
```

It gives rise to a type structure of the form $(t, \{\texttt{C} \mapsto t \to t,\ \texttt{D} \mapsto t\})$. Now this clearly respects equality, whether or not t possesses the equality attribute. In other words, "*TE* respects equality" does not uniquely determine equality attributes (this is due to the recursive nature of a *datbind*). But we want to be able to use equality whenever it is safe to do so, and therefore we want as many type names as possible to have the equality attribute. Formally then, let *TE* be a type environment, and let T be the set of type names t such that (t, CE) occurs in *TE* for some $CE \neq \{\}$. Then *TE* is said to *maximise equality* if (a) *TE* respects equality, and also (b) if any larger subset of T were to admit equality (without any change in the equality attribute of any type names not in T) then *TE* would cease to respect equality [Sec 4.9, p 21]. This condition is required for *TE* in rules 19 and 20 [p 25] for `datatype` and `abstype` declarations.

> The side-condition "*TE* maximises equality" really does determine uniquely the equality attribute of each of the chosen type names. This can be seen as follows. Let T be defined as above. Let us say that a subset T_0 of T is *nice* if, whenever the type names in T_0 are chosen to admit equality and the type names in $T \setminus T_0$ are chosen not to admit equality, then *TE* respects equality. Thus *TE* maximises equality if and only if the set of type names in T that are chosen to admit equality is maximal (with respect to set inclusion) among the nice sets. It is easy to check that a union of nice sets is nice. There is at least one nice set, namely the empty set. Thus we can define $T_{\text{eq}} \subseteq T$ to be the union of all nice sets and T_{eq} is nice. T_{eq} is therefore uniquely maximal among the nice sets, hence giving a *TE* which maximises equality.

[19] This condition can be ignored in the Core; it applies only to type structures which appear in signatures. See the discussion in Section 11.4

> The set of new type names that admit equality can be computed from *datbind* by an iterative process as the maximum fixed point of a monotonic operator. One starts out by assuming that all the new type names admit equality. In each iteration, we consider all the type structures of the type environment in question. If the *CE* component of a type structure (t, CE) reveals that t cannot admit equality, we change the equality attribute of t to false. This process is repeated till a fixed point is reached.

Finally, note the discussion at the end of [Sec 4.9] about rule 20 [p 25] for `abstype`; a datatype declared by `abstype` is deprived of both its constructors and its equality attributes, for use outside the `abstype` expression. The `abstype` declaration was introduced into ML before the Modules part of the language was designed; with Modules, its effect can be achieved to some extent by the use of signatures. For example, see how datatype constructors can be hidden by using a signature in a structure declaration in Example 6.3. More of this effect, e.g. hiding the equality attribute, could be achieved if the `abstraction` declaration proposed by MacQueen were introduced; see Section 8.4. Despite this overlap `abstype` has been preserved in the language, partly because its use is not restricted to structure-level, and partly because it provides a useful abstraction mechanism in the Core language without the considerable extra implementation effort required for the Modules.

Exercise 5.7 Consider the two declarations

`abstype T = A with` *dec* `end`

and

`local datatype T = A in type T = T;` *dec* `end`

Is their effect the same?

5.3 Principal types and environments

Given a context C and an expression *exp*, there may be zero, one or even infinitely many types τ such that

$$C \vdash \text{exp} \Rightarrow \tau$$

Considered as a kind of computation whose input is C and *exp* and whose output is τ, elaboration is therefore not determinate. Formally this shows up at several places in the inference rules, where one is often faced with having to make choices when constructing the inference tree. For example the rules require one to "guess" the type of any variable occurring in a pattern.

The non-procedural nature of the inference rules is convenient for the purpose of semantic definition, especially because one expression can have many types. However, the pragmatic significance of the particular set of inference rules for

elaboration lies in the fact that there exists a *type-checking algorithm* (or *type checker*) which, given any C and *exp*, returns FAIL if the set T defined by

$$T \stackrel{\text{def}}{=} \{\tau \mid C \vdash exp \Rightarrow \tau\}$$

is empty, and returns a type scheme σ such that

$$T = \{\tau \mid \sigma \succ \tau\}$$

if T is non-empty. In the latter case, σ is said to be a *principal type scheme* for *exp* in C. Conceptually, the type checker works by applying substitutions to type variables that occur free in some incomplete elaboration tree. (A *substitution* is a map μ from type variables to types such that for all α, if α is imperative, then $\mu(\alpha)$ is imperative[20] and if α is an equality type variable then $\mu(\alpha)$ admits equality.) For example the type checking of

```
fn x => x + 1
```

proceeds as follows: first the pattern `x` is given type `'a`, assuming `'a` is a fresh type variable. In this context, the argument to `+`, i.e. the pair `(x,1)` has type `'a * int` and since the type of `+` is `int * int` → `int` (conveniently ignoring overloading at this point), the algorithm employs the well-known *unification algorithm* to deduce that `'a` ought to be int. It therefore substitutes `int` for `'a` in the entire elaboration tree constructed so far. The reason this is valid is that, provided we ignore explicit type variables for the moment, the following theorem holds for all contexts C, phrases *phrase* and semantic objects A:

Theorem 5.1 (Substitution) *Let $C \vdash phrase \Rightarrow A$, where phrase contains no type variables in explicit type constraints. Then $\mu(C) \vdash phrase \Rightarrow \mu(A)$.*

Explicit type variables [Sec 4.6, p 20] pose a slight problem for the theorem, because their elaboration is strict in the sense that if α occurs explicitly in *phrase* then α elaborates to α and not to anything else (see [p 25, comment (11)]).

However, they can be dealt with as follows. For any substitution μ, let the *support* of μ, written Supp(μ), be the set $\{\alpha \mid \mu(\alpha) \neq \alpha\}$; also let the *yield* of μ, written Yield(μ), be the set of type variables occurring in any $\mu(\alpha)$ such that $\alpha \in$ Supp(μ). For any value declaration $\texttt{val}_U$ *valbind* the $\texttt{val}_U$ can be viewed as a binding operator, so the usual notions of free and bound occurrences (including the notion of renaming bound variables) apply to explicit type variables in *phrase*. To get a more general substitution theorem we now impose the following constraints on μ, C and *phrase*. First, we require that Supp(μ) be disjoint from U of C, for a substitution should not affect explicit type variables already in scope. With this requirement, if we write C in the form T, U, E, then $\mu(C)$ means simply $T, U, \mu(E)$. Second, we must demand that Supp(μ) is disjoint

[20] A type is *imperative* if it contains no applicative type variables [Sec 4.4, p 19].

from tyvars(*phrase*), the set of explicit type variables that occur free in *phrase*. This is needed because of the above-mentioned strictness of elaboration of type expressions. The theorem can then be generalised as follows:

Let $C \vdash$ *phrase* $\Rightarrow A$. *Assume also that* $\mathrm{Supp}(\mu) \cap (U \text{ of } C \cup \mathrm{tyvars}(\textit{phrase})) = \emptyset$, *and that (perhaps as a result of renaming) no explicit type variable occurring bound in phrase is a member of* $\mathrm{Yield}(\mu)$. *Then* $\mu(C) \vdash$ *phrase* $\Rightarrow \mu(A)$.

In proving this theorem, rules 11, 31, 45–47 and notably rule 17 are crucial.

The existence of principal types relies on the existence of most general unifying substitutions for types. The well-known unification result for first-order terms is that if two terms can be unified then there is a most general unifying substitution through which all other unifying substitutions can be factored. Although our notion of substitution to do with equality types and imperative types is slightly specialised, a similar result still holds.

A few obstacles to principality are caused by *overloading*. The first is that several pervasive arithmetic operators and relations [App C, p 75] have two possible types. Clearly the expression `op +` does not have a single principal type scheme, so the user has to put type constraints to guide the type checker, when necessary, in whatever way the implementer requires. The second is the typing of the *wildcard pattern row* (. . .) which may also require an explicit type constraint to achieve a principal type within the existing scheme of types and substitutions.

The aim of these explicit type constraints is to ensure that overloading is resolved, in the following precise sense. Given a phrase *phrase* and a context C, we say that overloading in *phrase* is *resolved for* C if

1. For every occurrence of one of the pervasive operators `* + - < > <= >= ~ abs` there is at most one type which can be inferred for that occurrence, and

2. For every occurrence of a pattern row, there is at most one set $\{lab_1, \ldots, lab_n\}$ of record labels[21] such that a type of the form $\{lab_1 : ty_1, \ldots, lab_n : ty_n\}$ can be inferred for that occurrence

in a successful elaboration of *phrase* in C.

It is only required that overloading be resolved in each *topdec*. But implementers may find it reasonable to require it to be resolved in some smaller textual unit. For example, a natural choice is to require it to be resolved in every *valbind*; in that case, the *topdec*

[21] Recent work by Rémy, following ideas of Wand, indicate that one can retain principal typing in a richer type system where a pattern row can match records with a different number of fields on different occasions. See M. Wand, *Corrigendum: Complete type inference for simple objects,* Proc. of the Third Symposium on Logic in Computer Science, l988 and D. Rémy, *Typechecking records and variants in a natural extension of ML*, Sixteenth Annual ACM Symposium on Principles of Programming Languages, Texas, 1989.

```
val x = let fun plus x y = x + y
          in plus 3 4
        end
```

would be rejected even though the *topdec* contains enough information to resolve the overloading of `+`.

We can now state the theorem asserting the existence of principal environments. The theorem is asserted for environments rather than for type schemes, because it is only at the interface between the Core and the Modules – namely when a *dec* is parsed as *strdec* so that rule 57 [p 38] is invoked – that principality is required, to secure the uniqueness of the elaboration of any *strdec*.

Let C be a context and *dec* a declaration. We say that an environment E is *principal for dec in* C if $C \vdash dec \Rightarrow E$ and moreover, for all E' for which $C \vdash dec \Rightarrow E'$, we have $\mathrm{Clos}_C E \succ E'$. (We must point out that the definition of principality of E [Sec 4.12, p 30] wrongly used "$E \succ E'$" in place of "$\mathrm{Clos}_C E \succ E'$".[22])

Theorem 5.2 (Principal Environments) *Assume that overloading in dec is resolved for* C*. Let* $C \vdash dec \Rightarrow E'$ *. Then there exists an principal environment for dec in* C*.*

The proof relies upon Theorem 5.1; in the proof one establishes a similar, but more general, property for all core language phrases.

Example 5.2 Let *dec* be the declaration `val g = (fn y =>y x)` considered in Example 4.1, and C the context considered in that example. Then the environment $\{\texttt{g} \mapsto \forall\texttt{'b}.(\texttt{'a} \to \texttt{'b}) \to \texttt{'b}\}$ is principal for *dec* in C. An example of a non-principal environment for *dec* in C is $\{\texttt{g} \mapsto (\texttt{'a} \to \texttt{int}) \to \texttt{int}\}$.

[22]The point is slightly subtle. The environment E required by rule 57 to be principal will be almost closed but not quite; it may still contain free imperative type variables left unquantified by rule 17 [p 25]. (See also the discussion in Section 4.5.) The relevant example is *dec* = `val x = ref []` ; this will elaborate to $E = \{\texttt{x} \mapsto \texttt{'\_a list ref}\}$, for any `'_a`, where `'_a` is still free because `ref []` is expansive. This E will later be rejected by rule 100 [p 44], but it is cleaner to ensure that a principal environment exists even in this case, and the slight adjustment achieves this while leaving unaffected the cases in which E is already closed.

6 Static Semantics for the Modules

This and the following chapters concern the static semantics for the Modules [Sec 5]. The static semantics for Modules will be described in more detail than was done for the static semantics of the Core.

The principal concepts are *structures*, *signatures* and *functors*, which have no parallel in many other programming languages. The rationale underlying the Modules was discussed in Section 3.1. In the present chapter we introduce structures and signatures by small programming examples. Readers who already know structures and signatures may still want to read this introductory chapter, because it explains the semantic objects. In Chapter 7 we comment on the definition of signature matching. Functors are discussed in Chapter 8.

In order to explain the more refined technical details, we then proceed to discuss the notion of *admissibility*, which underlies the Modules semantics (Chapter 9). The inference rules for signature expressions and specifications are treated in Chapter 10.

Just as the static semantics of the Core rests on a theory of principal types, so the static semantics of Modules rests on a theory of *principal signatures*. This theory is explained in Chapter 11. Finally, for the theoretically minded reader, we present rigorously in Appendix A the full proof that principal signatures exist; this theorem underlies the theory of Chapter 11, and to have a detailed proof has greatly increased our confidence in the design and the semantics.

6.1 Structures

In the static semantics, a *structure* is an environment stamped with a unique name, which distinguishes it from other structures; that is, it is a pair

$$(m, E)$$

where m is a *structure name* (the stamp) and E is an environment [Fig 10, p 17]. Structures can be declared using *structure declarations*, whose simplest form is

`structure` *strid* `=` `struct` *dec* `end`

where *strid* is a *structure identifier* and *dec* is a (Core) declaration. Figure 7 shows two structure declarations. Structure names enable us to distinguish between structures that are different, even though the static environments they contain are identical. For example, consider the structures in Figure 7. The two structures that result from the elaboration will have the same static environments but different structure names. Dynamically, the two structures will of course be different. Like type names, structure names play a role in the static semantics only. In the dynamic semantics, as we saw in Chapter 3, the concepts of structure and environment coincide. Indeed, implementers often choose to represent both structures and environments as record values at run-time.

```
structure OrdItem =
  struct
    type item = int
    val leq(i: item, j: item) = i<=j
  end

structure InvOrdItem =
  struct
    type item = int
    val leq(i: item, j: item) = j<=i
  end
```

Figure 7: Declaring two different structures

The phrases that denote structures are called *structure expressions*. A *structure-level* declaration, *strdec*, can declare the same kinds of objects as a Core declaration *dec*, but in addition it can declare structures [Fig 6, p 12]. Thus structures can be used as substructures in other structures, resulting in a hierarchy of structures. Figure 8 shows the declaration of a structure `Pair` which subsequently is incorporated as a substructure of structure `Complex`.

A *generative* structure expression is a structure expression of the form

`struct` *strdec* `end`

It is called generative because the elaboration of it results in a structure with a fresh name, see rule 53 [p 37].

Once a structure has been declared, its components can be referred to individually using *qualified identifiers*, such as `P.pair` and `P.mk_pair` in Figure 8 [Sec 2.4, p 4]. The general form of a qualified identifier is

$$strid_1 . \cdots . strid_k . id \quad (k \geq 1)$$

Since structures can contain substructures, a (static) environment E is actually a quadruple

$$(SE, TE, VE, EE)$$

where SE is a *structure environment*, i.e. a finite map from structure identifiers to structures [Fig 10, p 17].

Example 6.1 The sequential structure declaration from Figure 8 elaborates to the following environment, where we have omitted empty environment com-

```
structure Pair =
  struct
    type 'a pair = 'a * 'a
    fun fst(x,y) = x
    fun snd(x,y) = y
    fun mk_pair p = p
  end

structure Complex =
  struct
    structure P = Pair
    type complex = real P.pair
    fun mk_complex p = P.mk_pair p
    fun plus(c,c'): complex =
      P.mk_pair(P.fst c + P.fst c', P.snd c + P.snd c')
  end
```

Figure 8: Declaring a structure with a substructure

ponents:

$$\left\{ \begin{array}{l} \texttt{Pair} \mapsto S\ , \\ \texttt{Complex} \mapsto \Big(\texttt{m2},\ \{\ \texttt{P} \mapsto S\ \},\ \{\ \texttt{complex} \mapsto (\Lambda().\texttt{real} * \texttt{real}, \{\})\ \},\ \{\ \texttt{mk\_complex} \mapsto \forall\texttt{'a}.\texttt{'a} \to \texttt{'a}\ ,\ \texttt{plus} \mapsto ty\ \}\ \Big)\ \Big\} \end{array} \right.$$

where

$$ty = (\texttt{real} * \texttt{real}) * (\texttt{real} * \texttt{real}) \to \texttt{real} * \texttt{real}$$

and

$$S\ =\ (\ \texttt{m1},\ \{\ \texttt{pair} \mapsto (\Lambda\texttt{'a}.\texttt{'a} * \texttt{'a}, \{\})\},\ \{\ \texttt{fst} \mapsto \forall\texttt{'a'b}.\texttt{'a} * \texttt{'b} \to \texttt{'a}\ ,\ \texttt{snd} \mapsto \forall\texttt{'a'b}.\texttt{'a} * \texttt{'b} \to \texttt{'b}\ ,\ \texttt{mk\_pair} \mapsto \forall\texttt{'a}.\texttt{'a} \to \texttt{'a}\ \}\ )$$

(Empty environment components which have been omitted include, for example, the *SE* and *EE* components of the environment of S. We shall consistently omit empty environment components from now on.)

Note that although `Complex` is expressed indirectly in terms of `P`, the structure to which `Complex` elaborates in no way hides the true identity of the types; for example, `plus` gets type (`real`∗`real`)∗(`real`∗`real`)→`real`∗`real` (`real` is a type name here!). The structure names `m1` and `m2` stem from the

elaboration of the two generative structure expressions, so `m1` and `m2` must be chosen to be fresh, i.e. must be *generated*, in a sense made precise by the semantic rules.

Figure 9 shows part of the elaboration of the declarations from Figure 8. The reader is encouraged to use the figure as a guided tour of the inference rules for Modules. Notice that at a certain point, the elaboration reduces to elaboration of Core declarations.

6.2 Signatures

Intuitively, a *signature* is a structure type.[23] Whereas a structure expression *declares* a structure, a signature expression *specifies* a structure (or rather: a class of structures). For almost every kind of structure-level declaration *strdec* there is a corresponding kind of specification *spec* (compare the grammar for structure level declarations [Fig 6, p 12 and Fig 3, p 8] with that for specifications [Fig 7, p 13]). Moreover, corresponding to the generative structure expression

`struct` *strdec* `end`

we have the signature expression

`sig` *spec* `end`

which, for lack of a better name, is called the *generative* signature expression, although it does not generate any new structures or types.

Signatures are objects in their own right; they can be bound to signature identifiers and included in other signatures to form larger signatures. Figure 10 shows the declaration of two signatures (compare with Figure 8).

A signature is not the type of any particular structure, but rather of a whole class of structures, namely all the structures that *match* the signature. For example, `PAIR` can be matched by any structure which has at least a type `pair` (of arity one) and functions `fst`, `snd`, and `mk_pair` with the specified types.

How can we represent a signature semantically? A first attempt might be to elaborate a signature expression to exactly the same kind of object as a structure expression, i.e. to a structure. However, it would not be obvious how to choose the names occurring in this structure, for we can easily have two structures that both match the signature although they have different names. The point is that some of the structure and type names that occur in the signature are "generic" (or "flexible") names that range over "real" (or "rigid") names or type functions that occur in structures. Thus we take a signature to be an object of the form

$$(N)S$$

[23] We shall use this analogy between *signature* and *type*, to avoid too much pedantry. But there is a more precise analogy in which the three entities *signature*, *static structure* and *dynamic structure* correspond respectively to *type scheme*, *type* and *value*.

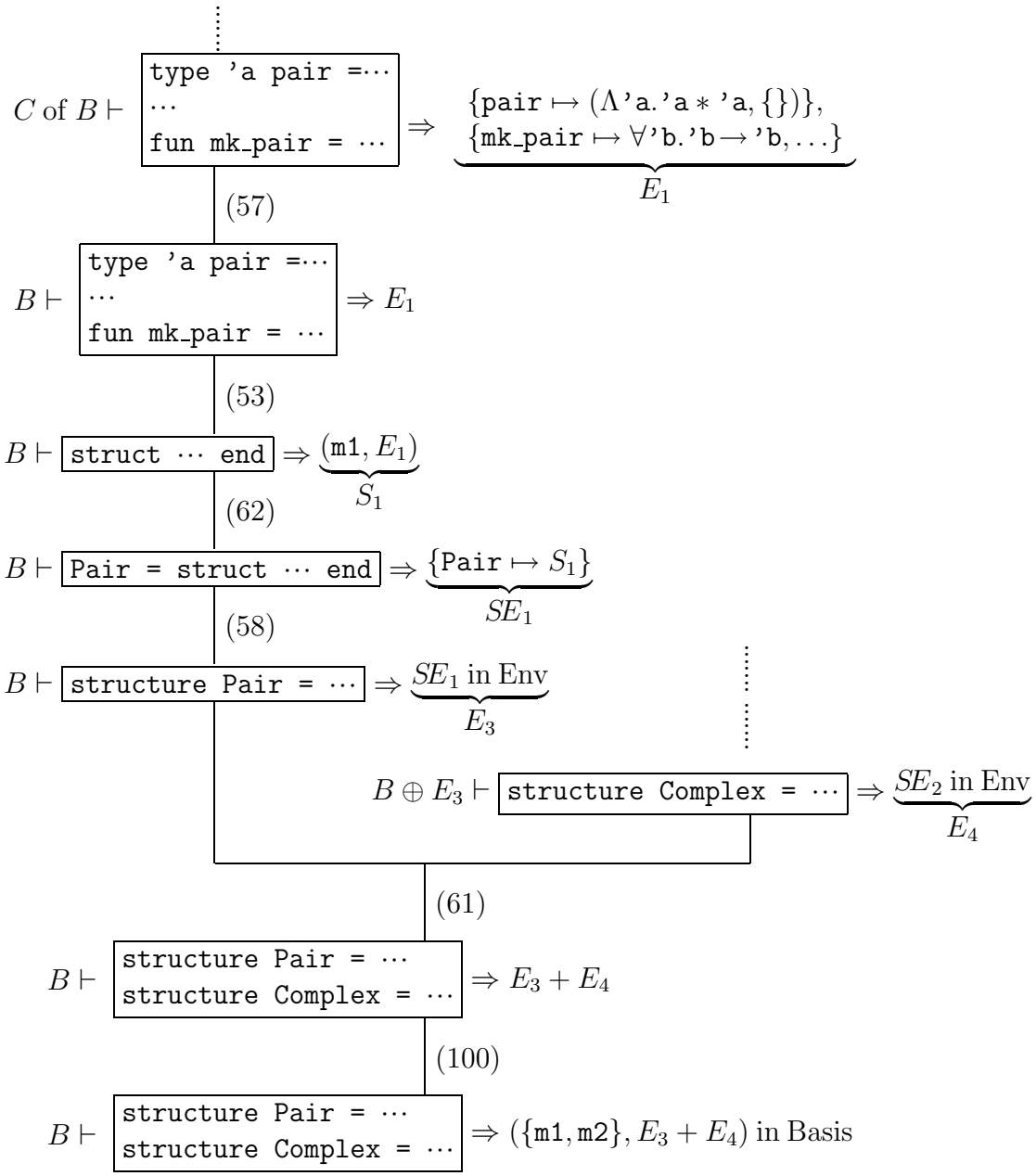

Figure 9: Elaboration of a structure-level declaration

```
signature PAIR =
  sig
    type 'a pair
    val fst: 'a pair -> 'a
    val snd: 'a pair -> 'a
    val mk_pair: 'a * 'a -> 'a pair
  end

signature COMPLEX =
  sig
    structure P: PAIR
    type complex
    val mk_complex: real P.pair -> complex
    val plus: complex * complex -> complex
  end
```

Figure 10: Declaring two simple signatures

where S is a structure, N is a pair (M, T), where M is a set of structure names and T a set of type names, and the parenthesis-pair $(\_)\_$ is a binding operator. To avoid too many brackets we mix M and T together and write e.g. $\{\mathtt{m}, \mathtt{m'}, \mathtt{t}\}S$ instead of $((\{\mathtt{m}, \mathtt{m'}\}, \{\mathtt{t}\}))S$. A name which occurs in S is *bound* (*by* N *in* $(N)S$) if it occurs in N and it is *free* otherwise. The bound names will be called *flexible* and the free names *rigid*. The reason behind this terminology is that flexible names can be instantiated every time one matches a structure against the signature, whereas the free names in the signature must be matched exactly by names in the structure, see Chapter 7. We use Σ to range over signatures [Sec 5.1, p 31].

A signature is *closed*, if it contains no free names. This means that it depends upon no structure or type defined elsewhere. (Informally, we can relax this definition slightly, and say that a signature is *closed* if every name that occurs free in it also occurs free in the initial static basis. In this sense, a signature is closed if it does not depend on any *user-defined* structure or type.) One can view closed signatures as living at a level which lies properly above the level of structures. In designing a large program, it can be useful to work with closed signatures only, in order to resist the temptation to rely on particular structures.[24] One can restrict oneself to closed signatures by adhering to the closure restrictions given in [Sec 3.6,p 14]. However, the language does not preclude the use of non-closed signatures.

[24]D.B. MacQueen, *Modules for Standard ML*, Proc. Symp. on Lisp and Functional Programming, Austin, Texas, 1984, pp 198–207, ACM, New York.

Example 6.2 The declaration of `PAIR` in Figure 10 elaborates to the signature environment $\{G = \texttt{PAIR} \mapsto \Sigma\}$, where Σ is the signature

$$\{\texttt{m},\texttt{t}\}\Big(\texttt{m},\{\texttt{pair} \mapsto (\texttt{t},\{\})\},\ \{\texttt{fst} \mapsto \forall\texttt{'a.'a t} \to \texttt{'a},\ \texttt{snd} \mapsto \forall\texttt{'a.'a t} \to \texttt{'a},\ \texttt{mk\_pair} \mapsto \forall\texttt{'a.'a} * \texttt{'a} \to \texttt{'a t}\}\Big)$$

Incidentally, Σ is closed. If we had the additional specification

```
val lth: 'a pair * 'a pair -> bool
```

in `PAIR`, we would have to add

$$\texttt{lth} \mapsto \forall\texttt{'a.'a t} * \texttt{'a t} \to \texttt{bool}$$

to Σ, which would no longer be closed, because of the free type name `bool`. Similarly, the declaration of `COMPLEX` yields a signature

$$\{\texttt{m},\texttt{m'},\texttt{t},\texttt{t'}\}\Big(\texttt{m'},\{\texttt{P} \mapsto (\texttt{m},\{\texttt{pair} \mapsto (\texttt{t},\{\})\},\ \{\texttt{fst} \mapsto \forall\texttt{'a.'a t} \to \texttt{'a},\ \texttt{snd} \mapsto \forall\texttt{'a.'a t} \to \texttt{'a},\ \texttt{mk\_pair} \mapsto \forall\texttt{'a.'a} * \texttt{'a} \to \texttt{'a t}\})\},\ \{\texttt{complex} \mapsto (\texttt{t'},\{\})\},\ \{\texttt{mk\_complex} \mapsto \texttt{real t} \to \texttt{t'},\ \texttt{plus} \mapsto \texttt{t'} * \texttt{t'} \to \texttt{t'}\}\Big)$$

Figure 11 shows part of the elaboration of the signature declaration from Figure 10. Once again, the figure is intended to give the reader an opportunity to take a look at some of the inference rules. Notice the strong similarity with the elaboration in Figure 9. However a significant difference is that, in the elaboration of signature expressions, the only use of the Core rules is to elaborate type expressions. The rules for the various kinds of *descriptions* [rules 82–87, p 41–42] are used during the elaboration of specifications to obtain the same kind of information as is obtained from the rules for Core declarations in the case of elaboration of structure-level declarations.

Exercise 6.1 In Exercise 3.2 you were asked to show that, as far as evaluation is concerned, the `include` specification

$$\texttt{include}\ \mathit{sigid}_1 \cdots \mathit{sigid}_n$$

is equivalent to

$$\texttt{local structure}\ \mathit{strid}_1 : \mathit{sigid}_1\ \texttt{and} \cdots \texttt{and}\ \mathit{strid}_n : \mathit{sigid}_n\ \texttt{in open}\ \mathit{strid}_1 \cdots \mathit{strid}_n\ \texttt{end}$$

Now show that the same holds for elaboration; that is, prove that for any B and E the former phrase elaborates to E in B if and only if the latter does. These two exercises together justify the claim that the `include` specification could have been given in the Definition as a derived form.

At first, one might be surprised to find that structure expressions elaborate to structures, not signatures. However, every structure S can be regarded as the trivial signature $\Sigma = (\emptyset)S$. If *strexp* elaborates to S in B then Σ is so to speak the *most precise* signature which the structure matches. (To talk about *the* most precise signature is justified by the fact that elaboration of structure expressions is deterministic up to the choice of generative names.) Since all names in Σ are rigid, only a structure which has the same structure names and type functions as S can match Σ. Values are represented by their types in S, so S is not to be confused with some dynamic structure to which *strexp* evaluates; S is merely a convenient shorthand for a signature in which the identity of all type and structure names is fully determined. In short, it represents *strexp* without any loss of information.

6.3 Sharing

We say that two structures *share*, if they have the same name. Similarly, two type structures *share* if they have the same type function.

Continuing the above example, we can even specify that the structure `P` is supposed to share with the `Pair` structure using a *sharing specification*:

```
signature COMPLEX_Pair =
  sig
    structure P: PAIR
    sharing P = Pair
    type complex
    val mk_complex: real P.pair -> complex
    val plus: complex * complex -> complex
  end
```

This will yield the following more specific signature

$$
\begin{aligned}
&\{\texttt{'m},\texttt{'t}\}\Big(\texttt{'m},\{\texttt{P} \mapsto (\texttt{m1},\{\texttt{pair} \mapsto (\Lambda\texttt{'a}.\texttt{'a} * \texttt{'a}, \{\})\},\\
&\qquad\qquad \{\texttt{fst} \mapsto \forall\texttt{'a}.\texttt{'a} * \texttt{'a} \to \texttt{'a},\\
&\qquad\qquad\ \texttt{snd} \mapsto \forall\texttt{'a}.\texttt{'a} * \texttt{'a} \to \texttt{'a},\\
&\qquad\qquad\ \texttt{mk\_pair} \mapsto \forall\texttt{'a}.\texttt{'a} * \texttt{'a} \to \texttt{'a} * \texttt{'a}\})\},\\
&\qquad \{\texttt{complex} \mapsto (\texttt{t'}, \{\})\},\\
&\qquad \{\texttt{mk\_complex} \mapsto \texttt{real} * \texttt{real} \to \texttt{t'},\\
&\qquad\ \texttt{plus} \mapsto \texttt{t'} * \texttt{t'} \to \texttt{t'}\}\Big)
\end{aligned}
$$

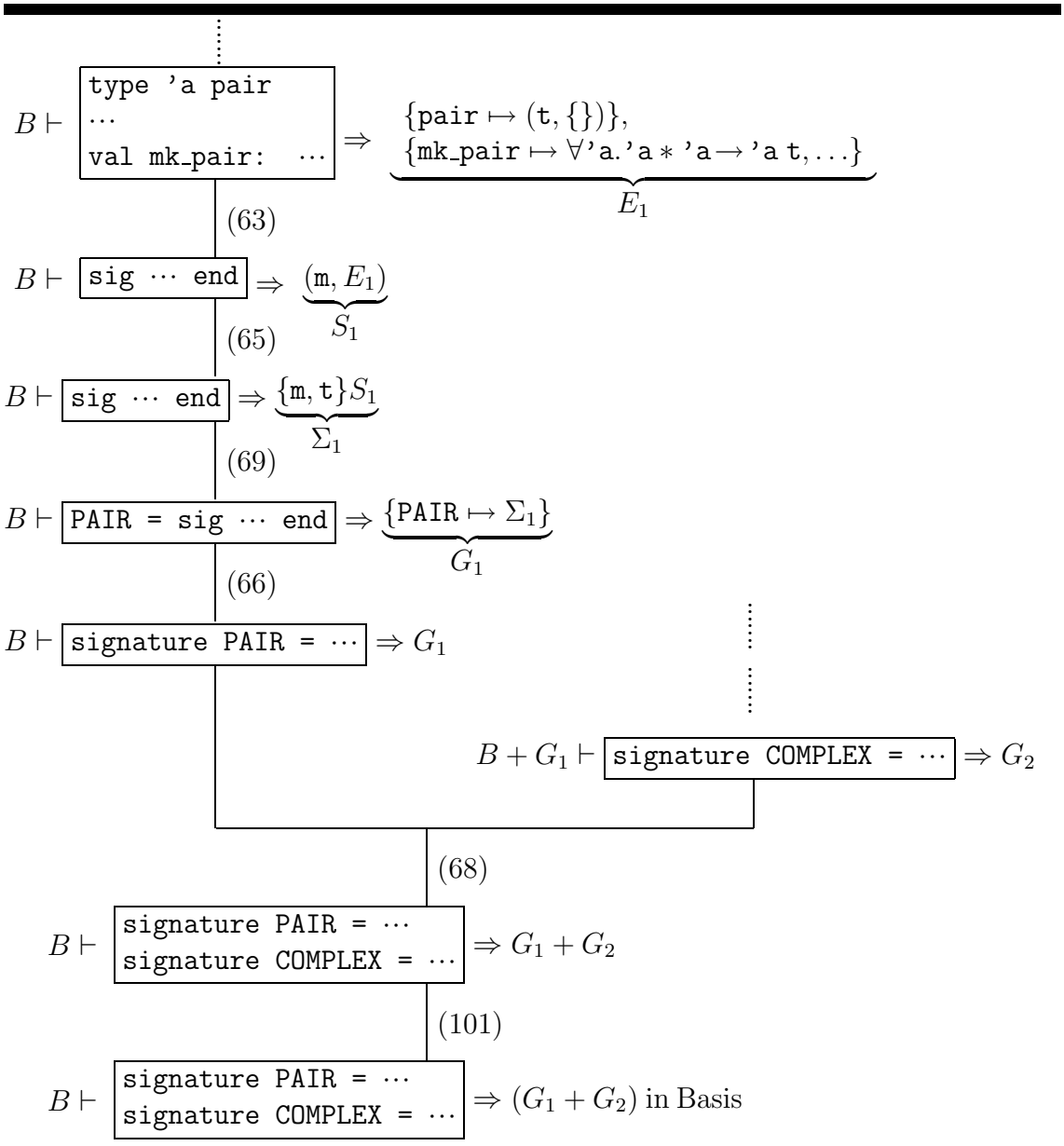

Figure 11: Elaboration of a signature declaration

The sharing constraint forces the structure and type names of `P` to be instantiated to the corresponding names and type functions in `Pair`. Thus `m` was instantiated to `m1` and `t` was instantiated to the type function Λ`'a.'a` $*$ `'a`. Notice that this instantiation gives more specific type schemes for `fst`, `snd` and `mk_pair` than those actually inferred for `Pair` itself.

Sharing which is specified with an existing structure (or type, see below) is said to be *external*. Sharing can also be *internal* to the signature, as in the following (which might be part of an ML implementation):

```
signature FRONT_END  =
  sig
    structure Parser:
      sig structure TV: TYVAR (* type variables *)
            ......
      end
    structure Elaborator:
      sig structure TV: TYVAR
            (* the Elaborator also knows about type variables*)
            ......
      end
    sharing Parser.TV = Elaborator.TV
  end}
```

Assuming that `TYVAR` is bound to the signature $(N_0)S_0$, `FRONT_END` becomes

$$(N)\Big(\texttt{m1}, \{\texttt{Parser} \mapsto (\texttt{m2}, \{\texttt{TV} \mapsto S_0, \ldots\}, \ldots),\\ \texttt{Elaborator} \mapsto (\texttt{m3}, \{\texttt{TV} \mapsto S_0, \ldots\}, \ldots)\}\Big)$$

where N is $N_0 \cup \{\texttt{m1}, \texttt{m2}, \texttt{m3}\}$, assuming that $N_0 \cap \{\texttt{m1}, \texttt{m2}, \texttt{m3}\} = \emptyset$. Note that the sharing is represented by the two occurrences of the structure S_0.

> The expression for `FRONT_END` suggests a different approach to sharing, taken in other languages, namely *sharing by parameterisation*. For example, in the above signature we could introduce a structure variable `s` of type S_0 and form the parameterised signature
>
> $$\Big[M, T, \texttt{s} : S_0\Big]\Big(\texttt{m1}, \{\texttt{Parser} \mapsto (\texttt{m2}, \{\texttt{TV} \mapsto \texttt{s}\}),\\ \texttt{Elaborator} \mapsto (\texttt{m3}, \{\texttt{TV} \mapsto \texttt{s}\})\}\Big)$$
>
> where the square brackets indicate a new binding operator.

Finally, one can have sharing equations of type constructors, as in the following example:

```
signature FRONT_END  =
  sig
    structure Parser:
      sig type tyvar (* type variables *)
            ......
      end
    structure Elaborator:
      sig type tyvar
            (* the Elaborator also knows about type variables*)
            ......
      end
    sharing type Parser.tyvar = Elaborator.tyvar
  end
```

Sharing can be specified between structures or between types, but not between values or exceptions, since such sharing cannot be decided at elaboration time.

One might expect to find a mechanism for type abbreviation in signatures, just as *declarations* like

```
type 'a t = int * 'a list
```

provide schematic abbreviation for all types of the form `int *` *ty* `list` . But to allow such abbreviatory specifications, even using a type constructor of arity 0, causes embarrassment in the presence of sharing. Consider

```
type 'a t
type u = int t
sharing type u = int
```

The effect is to demand that `t` be bound to a type function θ such that `int` θ = `int` . But there are two type functions θ which will work, namely Λ`'a.'a` and Λ`'a.int`; which should be chosen? (Similar problems are discussed at the end of Section 9.1 to do with consistency, and in Section 7.7 to do with type explication; in each case the attempt to find a "most general" meaning for a type constructor would involve second-order unification.) This difficulty also arises if, instead, we allow sharing equations to contain arbitrary type expressions. Indeed, if we did so then abbreviatory type specifications would be derived forms; for example, the above case would be equivalent to

```
type 'a t
type u  sharing type u = int t
sharing type u = int
```

This explains why type-sharing equations have been limited to type constructors.

Nonetheless there is some incentive, in an extension of ML, to allow a restricted class of type expressions to occur in sharing equations. One possibility is to admit a non-atomic type expression in a sharing equation only on condition that it contains no *flexible* type constructors, i.e. no type constructor which is specified at an earlier point in the current signature expression. (Thus `int t` above violates the condition, because `t` is specified just previously in the same signature.) If this extension were found to be sufficiently useful it would still need careful checking, to ensure that it does not invalidate the proof of principal signatures in Appendix A.

6.4 Coercive signature constraints

It is possible to constrain an existing structure by a signature, provided the structure matches the signature, and the effect is to obtain a special view of the structure where only the components specified in the signature are visible. (In Section 3.3 we called this the *curtailment* effect of a signature, when ascribed to a structure declaration.) Moreover, the semantic names in the (perhaps restricted) view will be exactly the names of the corresponding components in the existing structure, i.e. the restricted view shares with the un-restricted view.

Example 6.3 Consider the following program

```
signature VIEW1 = sig type t val x: t and y: t end
      and VIEW2 = sig type t val p: t * t  end

structure A =
  struct
    datatype t = BLUE | RED
    val x = BLUE and y = RED
    val p = (x,y)
  end

structure A1: VIEW1 = A
      and A2: VIEW2 = A;
```

As a result of the constraint of `A1` to `VIEW1`, only the components specified in `VIEW1` are accessible via `A1`. Hence neither `A1.p` nor `A1.RED` is accessible. However, the structure and type components that are in view maintain their structure and type names. For instance `A.t` and `A2.t` are bound to different type structures, because the former has two constructors and the latter none, but both type structures have the same type name. Hence, when used in type expressions, `A.t` , `A1.t` and `A2.t` can all be used interchangably. For example, `A.RED: A1.t` elaborates, despite the fact that `RED` is not in `VIEW1`.

Similarly,

```
if true then A.RED else A1.y
```

elaborates; in fact even `if A.RED = A1.y then 5 else 7` elaborates, for `A.t` admits equality and equality is an attribute of type names, so `A1.t` admits equality.

Exercise 6.2 Guess the structure environment resulting from elaborating the structure declarations in the above example.

6.5 Principal Signatures

Whenever a signature expression *sigexp* elaborates to a signature $(N)S$ in B, $(N)S$ is going to be what is called *equality-principal* for *sigexp* in B. The definitions of equality principality and the more primitive notion of *principality* are discussed in Chapter 11. Suffice it to say that if $(N)S$ is principal for *sigexp* in B then S has exactly the components and sharing which any structure which matches *sigexp* must have and moreover N is as large as possible, so that a name in S is flexible if it can be. Thus, a principal signature is as general as possible in the sense that it is not going to exclude any real structures that match *sigexp*. But at the same time, a principal signature is as informative as possible in the sense that it contains all the components and sharing which any structure which matches *sigexp* must have. To illustrate the latter point, consider the following *sigexp*:

```
sig
  structure A : sig structure C: sig end end
        and B : sig structure C: sig end end
  sharing A = B
end
```

The signature $(N)S$ which is principal for *sigexp* is such that $S($`A`$)$ and $S($`B`$)$ share and moreover $S($`A.C`$)$ and $S($`B.C`$)$ share, since this must be the case in any real structure which matches *sigexp*.

6.6 Summary

A structure may match different signatures of varying generality. For example, the structure

```
struct type t = int; val x = 3 end
```

matches all of the following signatures, which are listed in the order of strictly descending generality:

(a) `sig end`
(b) `sig type t end`
(c) `sig type t val x: t end`
(d) `sig type t val x: int end`
(e) `sig type t sharing type t = int val x: t end`

Exercise 6.3 For each of (b)–(e) above, give an example of a structure which does not match the specification, but matches the previous one.

Conversely, a signature may of course be matched by many structures. Also, a (semantic) structure S may be obtained from many different structure expressions and a (semantic) signature Σ may be obtained from many different signature expressions. Sometimes, a semantic structure or signature may even contain free names that are not reachable by looking up a long structure identifier or type constructor in the current basis, in which case printing a sensible textual representation of the semantic object becomes harder.

However it can be proved that, given a basis B and a structure expression *strexp*, there is at most one semantic structure S, such that *strexp* elaborates to S in B, up to renaming of generative names in S.

Moreover, given a basis B and a signature expression *sigexp*, there is at most one semantic signature $(N)S$, such that *sigexp* elaborates to $(N)S$ in B, up to renaming of the bound names N, namely the equality-principal signature for *sigexp* in B.

7 Signature Matching

In this chapter we discuss what it is for a structure to *match* a signature. We first describe how matching is a combination of *instantiation* and *enrichment*. In doing so, we shall introduce the crucial concept of a *realisation*. Finally, we prove a theorem which states that, informally speaking, for every structure S and signature Σ, if Σ is what is called *type-explicit*, then there is at most one way in which S can match Σ.

7.1 Matching

A structure S *matches* a signature Σ if, for some S^-, S enriches S^- and S^- is an instance of Σ [Sec 5.12, p 35]. This of course leaves us with defining enrichment and instantiation. Write Σ in the form $(N_1)S_1$. Recall that the names in N_1 are flexible and that the free names are rigid. Roughly speaking, instantiation is concerned with mapping the flexible names in Σ to the actual names in S, and enrichment is concerned with the fact that S may have more components and more polymorphism than specified in Σ.

7.2 Realisation

In order to be able to instantiate flexible names to rigid names, we need the notion of *realisation*. A *type realisation* is a map φ_{Ty} mapping type names to type functions such that t and $\varphi_{\mathrm{Ty}}t$ have the same arity, and if t admits equality then so does $\varphi_{\mathrm{Ty}}t$ [Sec 5.6, p 33]. We can say that a *structure realisation* is a map φ_{Str} from structure names to structure names. Then a *realisation*, φ, is a pair $(\varphi_{\mathrm{Ty}}, \varphi_{\mathrm{Str}})$ of a type realisation and a structure realisation [Sec 5.7, p 33]. In instantiating a signature $(N_1)S_1$, we want realisation to act on the bound names only. For any realisation φ we define the *support of* φ, $\mathrm{Supp}\,\varphi$, to be the set of names n for which $\varphi n \neq n$ [Sec 5.7, p 33].

7.3 Instantiation

Then we say that S^- is an *instance* of $(N_1)S_1$, written $(N_1)S_1 \geq S^-$, if there exists a realisation φ which affects flexible names only (i.e. $\mathrm{Supp}\,\varphi \subseteq N_1$) such that $\varphi(S_1) = S^-$ [Sec 5.9, p 34].

Notice that two different names in S_1 may be mapped to the same name (or type function) in S^-. Thus there may be sharing in S^- which was not specified in S_1. On the other hand, different occurrences of the same name in S_1 will of course be mapped to the same name (or type function) by φ, so S^- must have *at least* as much sharing as specified by S_1. Also note that, by the definition of type realisation, if a flexible type name in the signature admits equality then it can only be realised by a type function that admits equality. Finally, since φ may

change names, but never affects the domain of any kind of environment, $(N_1)S_1$ and S^- have exactly the same "shape", i.e. for any kind of long identifier, *longid*, $S^-(longid)$ exists if and only if $S_1(longid)$ exists.

Exercise 7.1 Recall the declarations in Example 6.3. What is the signature to which `VIEW1` elaborates? Give a structure to which `A` elaborates. Does there exist an instance of `VIEW1` which agrees with the structure and type names used in `A`?

To digress from matching for a moment, note that in [Sec 5.9] the relation of instantiation between two signatures, $\Sigma_1 \geq \Sigma_2$, is defined in terms of instantiation between a signature and a structure. It is easy to show that $\Sigma_1 \geq \Sigma_2$ and $\Sigma_2 \geq \Sigma_1$ if and only if Σ_1 and Σ_2 are identical after renaming of bound names, and removal of any unused bound names from their prefixes. We shall say in this case that Σ_1 and Σ_2 are identical. This is relevant to the results about principal signatures in Chapter 11.

7.4 Enrichment

The other half of signature matching, *enrichment*, allows the actual structure S to have more components and more polymorphism than the signature instance S^-, but S and S^- must agree on the names of structure and type components that are present in S^-. The precise definition of the enrichment relation $S_1 \succ S_2$ is found in [Sec 5.11, p 34].

Several points are worth noting about this definition.

1. It depends, via the requirement $VE_1(id) \succ VE_2(id)$ for example, upon the relation of generalisation between type schemes [Sec 4.5, p 19], which is also denoted by $\succ$.

2. In the clause dealing with exception environments you might expect to see the condition $EE_1(excon) \succ EE_2(excon)$ instead of $EE_1(excon) = EE_2(excon)$. But recall that $EE(tycon)$ is always a *type*, not an arbitrary *type scheme*; between types, enrichment ($\succ$) coincides with equality.

3. For two different reasons, in the clause dealing with constructor environments you may be surprised at the strictness of the condition that $CE_1 = CE_2$ or $CE_2 = \{\}$. First, why not $\mathrm{Dom}(CE_1) \supseteq \mathrm{Dom}(CE_2)$ and ..., as in the earlier clauses? The reason is that when a datatype is specified by a type structure (θ_2, CE_2) for which $CE_2 \neq \{\}$, for example in the argument signature of a functor, then we wish to retain the validity of the check for *exhaustiveness* [Sec 4.11, p 30] of any *match* which occurs in the functor

body; this check would be worthless if a datatype with extra constructors were allowed to match the specifying type structure.[25]

4. Second, you may still be surprised that we have not merely required that $CE_1(con) \succ CE_2(con)$ for each *con*, i.e. *generalisation* rather than *equality* of type schemes. However, the type schemes which appear in a constructor environment are so special that enrichment between two of them actually *implies* equality, as the following exercise illustrates.

Exercise 7.2 Let $\sigma_1 \succ \sigma_2$, where $\sigma_1 = \forall$'a'b.$\tau_1 \to$ ('a, 'b)t_1 and $\sigma_2 = \forall$'a'b.$\tau_2 \to$ ('a, 'b)t_2 , and where 'a and 'b are the only type variables occurring in τ_1 and τ_2 (see the third restriction in [Sec 2.9, p 9]). Prove that $\sigma_1 = \sigma_2$.

The two conditions of enrichment – more components, more polymorphism – are really closely related. For we can think of a polymorphic function $f : \sigma$ as a family of functions $f : \tau$, one for each monotype τ such that $\sigma \succ \tau$. Now if $\sigma \succ \sigma'$, then each member of the family $f : \sigma'$ is also a member of the family $f : \sigma$; this is just another case of "more components".

7.5 Discussion of matching

The definition of matching in terms of enrichment and instantiation is, formally:

$$S \textit{ matches } (N_1)S_1 \text{ if, for some } S^-,\ (N_1)S_1 \geq S^- \prec S$$

Exercise 7.3 Recall the structures `Pair` and `Complex` from Section 6.1 and the signatures `PAIR` and `COMPLEX` from Section 6.2. Prove that the structures match the signatures, by exhibiting the required realisations. Are these realisations uniquely determined by the structures and the signatures? Are there value components which are more polymorphic in the actual structure than in the signature instance?

Exercise 7.4 In the following, does structure `A` match `SIG`? Does `B`?

```
structure A = struct datatype t = C | D of  int end
structure B = struct datatype 'a t = C | D of 'a end
signature SIG  = sig datatype 'a t = C | D of int end
```

[25] The reader may like to reflect upon the two levels of matching in ML. There is some analogy between them, but large differences. Matching a structure to a signature is done entirely at elaboration time, and no alternatives are provided; matching a value to a pattern is done at evaluation time, and a set of alternative patterns is provided (see Section 2.5). But in the latter case there is still certainty, before evaluation time, that the value will *fit* the *match*; this is provided partly by the type discipline and partly by the exhaustiveness check (which ensures that the value will fit at least *one* pattern in the *match*).

As an example of the use of matching in the inference rules, consider rule 62 [p 39] concerning structure bindings, omitting the second option:

$$\frac{B \vdash \mathit{strexp} \Rightarrow S \qquad \langle B \vdash \mathit{sigexp} \Rightarrow \Sigma \ , \ \Sigma \geq S' \prec S \rangle}{B \vdash \mathit{strid}\ \langle \texttt{:}\ \mathit{sigexp} \rangle\ \texttt{=}\ \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto S\langle' \rangle\}}$$

Read operationally, this rule says that we first elaborate the structure expression to get S and then elaborate the signature expression (assuming it is present) to get a signature Σ and then match S against Σ. If the match is successful, it is the signature instance S' which becomes the view bound to the structure identifier.

By simply dropping $\langle' \rangle$ from rule 62, we would lose the *coerciveness* of signature matching, as described in Section 6.4; the signature in a structure binding would become merely a *condition* to be satisfied by the structure – that it should possess at least the components (and at least the sharing) mentioned in the *sigexp*. There is a perfectly sound alternative semantics for ML in which signatures are uniformly treated as conditions, not as coercions. (We also discussed this in the context of dynamic semantics, Section 3.5, using the word "curtailment" in place of "coercion".) Remarkably few – but important – other changes have to be made to the present static semantics; for example, rule 99 [p 44] for functor bindings needs a similar adjustment. Perhaps we can look forward to a future language in which signatures are used *both* as coercions *and* as conditions; for the present, it is important to find out exactly when discomfort is caused by only having coercions.

7.6 Equality type specifications

To specify a type constructor `t` which has arity k and admits equality, one writes specifications such as

`eqtype t`	(if $k = 0$)
`eqtype 'a t`	(if $k = 1$)
`eqtype ('a,'b,'c)t`	(if $k = 3$)

By the definition of type realisation [Sec 5.6, p 33], such a type can only be realised by a type function which also has arity k and admits equality.

Example 7.1 In the following, `List1` matches `LIST`, but `List2` does not:

```
signature LIST =
  sig
    eqtype item
    val is_in: item * item list -> bool
  end
structure List1 =
  struct
    datatype item = A | B of item
```

```
    fun is_in(_,[ ]) = false
      | is_in(i, hd::tl) = i=hd orelse is_in(i,tl)
  end
structure List2 =
  struct
    type item = int -> int
    fun is_in(_,[ ]) = false
      | is_in(i, hd::tl) = i=hd orelse is_in(i,tl)
  end
```

As a point of language design, it is interesting to consider an alternative to `eqtype`. The specification `eqtype 'a t` does two different things; it specifies that a (unary) type constructor `t` should exist, and it specifies that `t` should admit equality. The first of these things is already done by `type 'a t` . One can therefore imagine a different primitive, say

```
equality t
```

which does only the second thing; like `sharing`, it specifies an additional property for entities which exist already. Then `eqtype 'a t` would be a derived form, equivalent to

```
type 'a t equality t
```

Moreover `equality` would be convenient in such a situation as

```
sig structure S: SIG equality S.t end
```

effectively strengthening `SIG` by an extra requirement. This can be done using `eqtype`, but only messily, as follows:

```
sig structure S: SIG
    local eqtype 'a u in sharing type S.t=u end
end
```

7.7 Type explication

Given a structure S and a signature $(N_1)S_1$, we would like it to be the case that there is at most one structure S^- such that $(N_1)S_1 \geq S^- \prec S$, for S^- is supposed to be *the* result of "cutting down" S by $(N_1)S_1$. For the purpose of functor application (explained later), we actually want a stronger property, namely that there exist at most one realisation φ such that $\operatorname{Supp}\varphi \subseteq N_1$ and $\varphi(S_1) \prec S$.

Unless special care is taken this will not hold, as the following example shows:

Example 7.2 Consider the signature

```
signature SIG  = sig  type 'a t  val x: int t  type t  end
```

and the structure

```
structure Str  = struct  val x = (3,4)  type t = bool  end
```

The second specification of `t` in `SIG` overwrites the first, so `SIG` elaborates to

$$\{\texttt{m1}, \texttt{t1}, \texttt{t2}\}(\texttt{m1},\ \{\texttt{t} \mapsto (\texttt{t2}, \{\})\},\ \{\texttt{x} \mapsto \texttt{int t1}\})$$

whereas `Str` elaborates to

$$(\texttt{m3},\ \{\texttt{t} \mapsto (\texttt{bool}, \{\})\},\ \{\texttt{x} \mapsto \texttt{int} * \texttt{int}\})$$

Now there are four different realisations φ of `t1` which satisfy $(\texttt{int})(\varphi(\texttt{t1})) = \texttt{int} * \texttt{int}$, namely $\theta = \Lambda\texttt{'a.'a} * \texttt{'a}$, $\theta = \Lambda\texttt{'a.'a} * \texttt{int}$, $\theta = \Lambda\texttt{'a.int} * \texttt{'a}$, $\theta = \Lambda\texttt{'a.int} * \texttt{int}$. The problem is that, by comparing the type functions of the specified type constructors with the type functions in the actual structure, we can determine the value of $\varphi(\texttt{t2})$ but not of $\varphi(\texttt{t1})$.

We therefore define that a signature $(N_1)S_1$ is *type-explicit* if, whenever $t \in N_1$ occurs free in S_1, then some substructure of S_1 contains a type environment TE such that $TE(tycon) = (t, CE)$ for some $tycon$ and some CE [Sec 5.8, p 33]. Before a signature is accepted as the result of an elaboration, i.e. at the point of application of rule 65, it is checked whether the signature really is type-explicit. This will rule out the declaration of `SIG` above.

For any structure S and signature $(N_1)S_1$ and realisation φ, we say that S *matches* $(N_1)S_1$ *via* φ, if $\mathrm{Supp}\,\varphi \subseteq N_1$ and $\varphi S_1 \prec S$ [Sec 5.12, p 35]. In the statement of Theorem 7.1 we refer to the notion of well-formed signature, which will be discussed in Section 9.2. However, the only property of well-formed signatures that is used in the proof is that if $(N)S$ is well-formed then $N \subseteq \mathrm{names}\, S$.

Theorem 7.1 (Type Explication) *For any structure S and signature Σ, if Σ is type-explicit and well-formed, then there is at most one realisation φ such that S matches Σ via φ.*

Proof Write Σ in the form $(N_1)S_1$. Let S match Σ via two realisations φ and φ', and let $n \in N_1$. Since Σ is well-formed, we have $n \in \mathrm{names}\, S_1$. Therefore if n is a structure name, S_1 contains a substructure (n, E) for some E; on the other hand if n is a type name then, since Σ is type-explicit, S_1 contains a type structure (n, CE) for some CE. In either case $\varphi n = \varphi' n$, by the definition of enrichment, since S enriches both φS_1 and $\varphi' S_1$. Hence φ and φ' agree for all names in N_1. But $\mathrm{Supp}\,\varphi, \mathrm{Supp}\,\varphi' \subseteq N_1$, so $\varphi = \varphi'$.

8 Elaboration of Functors

In this chapter we shall comment on the static semantics of functors. In Section 8.1 we review the concept of functor itself; see also Section 3.4. In Sections 8.2 and 8.3 we then comment on the rules for functor declaration and application, respectively. In Section 8.4 we look at some alternative meanings for signature ascriptions. Finally, in Section 8.5 we comment on the possibility of admitting higher-order functors.

8.1 Discussion

The static semantic object corresponding to a functor is called a *functor signature* [Fig 11, p 31]. As a result of elaborating a functor declaration

$$\texttt{functor}\ \mathit{funid}\ \texttt{(}\ \mathit{strid}\texttt{:}\ \mathit{sigexp}\ \texttt{)}\langle\texttt{:}\ \mathit{sigexp}'\rangle\ \texttt{=}\ \mathit{strexp}$$

a functor signature is bound to *funid* and entered into the basis. A functor signature is analogous to a signature, but has two parts (argument and result) in place of one; it takes the form $(N)(S, (N')S')$, where $(N)S$ represents the argument signature while $(N')S'$ – loosely speaking – is a signature for the result structure. A functor is elaborated just once, and the elaboration takes place at the time of declaration; the principle of static scoping for functors makes this possible.

The elaboration of an application of *funid* consists of first checking that the actual argument structure matches $(N)S$, and then deriving the result from $(N')S'$ in a manner that will be described later. Elaboration of a functor application does *not* require the re-elaboration of the functor body, despite the fact that the argument signature may specify types, whose true identity does not become known until the functor is applied.

When a functor declaration is elaborated, the body of the functor is elaborated in a basis in which *strid* has been bound to a formal structure which matches *sigexp* and has all the sharing, the polymorphism, the equality attributes and the components that any structure which matches *sigexp* must have, but no more. This is achieved by elaborating *sigexp* to an equality-principal signature $(N)S$ and then using S as the formal structure. We emphasise the practical significance of principality at this point; if there were no natural choice of "best" elaboration for the argument signature, then one would indeed be faced with either recording some (possibly infinite!) set of "good" results, one for each "good" elaboration of *sigexp*, or else requiring re-elaboration of the functor body for each functor application. It is helpful here to recall the dynamic semantics of functors from Section 3.4, and then to compare both with the treatment of functions in the Core. In the dynamic semantics, function and functor closures both contain code to be evaluated at each application; in the static semantics, both function types and functor signatures are truly abstract objects which summarise all possible applications.

In writing the functor body *strexp*, one can consider *strid* as a kind of abstract structure or abstract datatype. If the body relies on properties of the argument that are not specified in the formal argument signature, the elaboration of the body will fail. Conversely, if a functor declaration does elaborate, then one is certain that the body of the functor would also elaborate if one were to replace the formal structure by an actual structure, provided of course that the actual structure matches the formal argument signature and is cut down to contain just the components mentioned in the signature.

Functors need not be closed [Sec 3.6, p 14]; besides referring to the formal argument structure and its components, the body and the result signature can also refer to identifiers declared outside the functor. Of course, if a functor is closed – except perhaps for non-local signature identifiers, and reference to standard types and values – then the functor declaration can be executed (i.e. both elaborated and evaluated) in the initial basis, perhaps augmented just by a signature environment. This leads to the possibility of independent compilation of functors. But in a body of a functor f one often needs to refer non-locally to another functor g say; one cannot abstract this reference by making g a parameter of f because functors cannot take functor parameters (see Section 8.5 for the possibility of relaxing this restriction). This is why a scheme is proposed in [Sec 3.6, p 15] in which functors could refer non-locally only to signatures and to other functors; a separately compilable module would then include specifications of the free functor identifiers, in the form of *functor signature expressions*, whose semantics is given by rule 95 [p 43]. These suggested closure restrictions are very near those in MacQueen's original proposal for Modules; but it was encouraging to find no semantic complication at all in treating the fully general case in the Definition.

In the Core, the most common reason for abstracting an expression to form a function is perhaps that one wishes to apply the function at least twice. Some functors are written with the purpose of being applied more than once, but the most important use of functors is probably to act as an abstraction mechanism. In that case, the formal argument signature serves as an *interface*, and the elaboration of the functor body is such that the body can only use what the interface specifies. Furthermore, the compiler or elaborator checks that any actual structure to which the functor is applied really does satisfy the interface. The reader will recall from Section 3.3 that the term *interface* has an exact meaning in the dynamic semantics; there, it represents all that is relevant at run-time about an argument signature, i.e. simply the names of a structure's components.

In some ways, functors are less general than functions. Functors can only be declared at top level; since they cannot be declared inside structures, they cannot be returned as results from functors. Also, as we saw above, functors cannot be parameterised on functors. Functors are not recursive, neither by themselves nor by mutual recursion. Last but not least, a functor's result structure cannot be made to depend in an arbitrary way upon the *values* in its argument structure; in particular, there is no conditional form at the level of structures which would

allow one to write

$$\texttt{if A.x = 0 then } \mathit{strexp}_1 \texttt{ else } \mathit{strexp}_2$$

8.2 Functor declaration

Consider a functor declaration *fundec* of the form

$$\texttt{functor } \mathit{funid} \texttt{ (} \mathit{strid} \texttt{ :} \mathit{sigexp} \texttt{) = } \mathit{strexp}$$

(Empty and sequential functor declarations present no added difficulty, nor do functor bindings that bind more than one functor identifier. Result signatures are more interesting and will be treated separately below.) We can see the effect of such a functor declaration more clearly if we combine rules 96 [p 43] and 99 [p 44] into a single rule

$$\frac{\begin{array}{ll}(\mathrm{P1}) & B \vdash \mathit{sigexp} \Rightarrow (N)S \\ (\mathrm{P2}) & B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ (\mathrm{P3}) & N' = \mathrm{names}\, S' \setminus ((N \text{ of } B) \cup N)\end{array}}{B \vdash \mathit{fundec} \Rightarrow \{\mathit{funid} \mapsto (N)(S, (N')S')\}}$$

The result of the elaboration is a *functor environment*, i.e. a finite map from functor identifiers to functor signatures [Fig 11, p 31]. In the functor signature $(N)(S, (N')S')$, N and S are obtained by elaborating the argument signature, see (P1), and S' is obtained by elaborating the body of the functor, see (P2); N' is the set of names occurring in S' that are "new", i.e. that stem neither from the basis B nor from the formal argument structure S (in fact they stem from generative structure expressions or datatype declarations occurring in the body, or else occurring in the body of another functor invoked by the body). These names are called the *generative* names of the functor signature.

Now let us consider the three premises one at a time. At (P1) we elaborate *sigexp* to a signature. This can only happen via rule 65, so we know that $(N)S$ is equality-principal for *sigexp* in B. Hence S contains as many components and as much sharing and equality as any structure which matches *sigexp* must contain. Moreover, according to the definition of principality [Sec 5.14, p 35] we can assume that the bound names N have been chosen to be disjoint from the names occurring free in B.

At (P2) we then bind S to *strid* and add the binding to the basis before elaborating the functor body. Since N is chosen disjoint from B, we do not thereby assume any sharing between S and B which does not already exist between $(N)S$ and B. By dropping the name prefix (N) from $(N)S$ within the body, we have in effect introduced the names in N as new constant type and structure names whose scope is the functor body. That is why $\oplus$ is used: all names in S should be treated as rigid within the body. In terms of Section 10.1, the functor body will

be elaborated in a *rigid* basis. In particular therefore, the elaboration of the body cannot retrospectively identify names that are different in S.

Example 8.1 Consider the following functors:

```
functor F(S: sig type u and t
                 val x: u and y: t
             end) =
  struct
    fun choice b = if b then S.x else S.y
  end

functor G(S: sig type t val x:t and y:t end)=
  struct
    val b = (S.x = S.y)
  end

functor H(S: sig datatype t = C | D
                 val x: t
             end) =
  struct
    val b = (S.x = S.C)
  end
```

The declaration of `F` will *not* elaborate, for within the functor body, `t` and `u` are different rigid types, just as different as `t` and `int`, say. Similarly, the declaration of `G` will *not* elaborate, because `t` was not specified as an `eqtype`. However, `H` will elaborate, because the type name for `t` in the equality-principal signature will admit equality. Moreover, `G` would also elaborate if `eqtype` were to replace `type` in its argument signature.

Finally, at (P3) we take N' to be the set of names that were freshly generated by the elaboration of the body. The reason for giving these names a special status is that each time the functor is applied, they must be replaced by names that are fresh *with respect to the basis in which the application takes place*. For example, every time the functor

```
functor J(S: sig type t end) =
struct
  datatype u = C of S.t | D
end
```

is applied, `u` will be given a fresh type name, as indeed is necessary, since `t` may vary from application to application.

We now see why a functor signature is of the form $(N)(S, (N')S')$ and not, say, $((N)S, (N')S')$ or $(N)(S, S')$ or $(N \cup N')(S, S')$. The generative names N' pertain strictly to the body of the functor, so the scope of the binding (N') should be S', rather than (S, S'). On the other hand, the result of the functor may well share with the formal argument S, as indeed in functor `J` above, where the type of `C` in the result signature contains the type name of `S.t`. Therefore, (N) binds over both S and $(N')S'$. This implies that if we rename the names in N we will in general have to rename names that occur free in $(N')S'$. More importantly, if we instantiate the names in N to names in an actual structure to which the functor is applied then we must also instantiate names that are free in $(N')S'$ and occur in N.

Exercise 8.1 What is the functor signature of `J`?

Now let us consider the case of a functor declaration *fundec with* a result signature:

`functor` *funid* `(` *strid* `:`*sigexp* `)` `:`*sigexp'* `=` *strexp*

for which we can combine rules 96 and 99 into the single rule

$$\frac{\begin{array}{ll}\text{(P1)} & B \vdash \mathit{sigexp} \Rightarrow (N)S \\ \text{(P2)} & B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ \text{(P3)} & B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow \Sigma' \\ \text{(P4)} & \Sigma' \geq S'' \prec S' \\ \text{(P5)} & N' = \operatorname{names} S'' \setminus ((N \text{ of } B) \cup N)\end{array}}{B \vdash \mathit{fundec} \Rightarrow \{\mathit{funid} \mapsto (N)(S, (N')S'')\}}$$

Here (P1) and (P2) are as in the case of no result signature described above. At (P3) we elaborate the result signature in the same basis as we elaborate the functor body. In particular, the result signature can refer to the argument structure, but no sharing specification in the result signature can retrospectively impose sharing which was not met in $B \oplus \{\mathit{strid} \mapsto S\}$. For example, the following functor declaration does *not* elaborate

```
functor K(S: sig type t and u end):
  sig sharing type S.t = S.u end
  = struct end
```

Next, at (P4) it is checked whether the body matches the result signature, i.e. whether there exists a structure S'' such that $\Sigma' \geq S'' \prec S'$. Finally, at (P5), N' is taken to be the generative names of S''. Since $S'' \prec S'$, all names that occur in S'' also occur in S'.[26]

[26]The careful reader will have noticed that the definition of N' in (P5) differs slightly from the one in rule 99. There is a small mistake in the latter, whose side-condition should have been $N' = \operatorname{names} S'\langle' \rangle \setminus ((N \text{ of } B) \cup N)$. If one omits the optional prime, then $(N')S''$ may fail to be well-formed, since we do not necessarily have $N' \subseteq \operatorname{names} S''$.

In the conclusion, the crucial point is that it is S'', not S', which is used in the functor signature, i.e. the result structure S' is "cut down" to the view given by the result signature. In particular, S'' may have less components and less polymorphism that S' but since S' enriches S'', the result signature does not hide the identity of any type or structure that is not cut away by the result signature.

Exercise 8.2 Let us say that two phrases $phrase_1$ and $phrase_2$ are *equivalent* if for all semantic objects A and A',

$$A \vdash phrase_1 \Rightarrow A' \quad \textit{iff} \quad A \vdash phrase_2 \Rightarrow A'$$

Prove that $fundec_1$ is equivalent to $fundec_2$, where

$fundec_1 \equiv$ `functor` *funid*`(`*strid* `:`*sigexp*`)` `:`*sigexp*$'$ `=` *strexp*

$fundec_2 \equiv$
```
functor funid(strid :sigexp) =
    let structure strid' :sigexp' = strexp
    in
      strid'
    end
```

for all $strid'$. In other words, functors with result signatures can be expressed as a derived form of functors without result signatures.

8.3 Functor application

We shall now comment on the rule for functor application, rule 55 [p 37], which more explicitly can be stated

$$\begin{array}{ll} \text{(P1)} & B(funid) = (N_1)(S_1, (N_1')S_1') \\ \text{(P2)} & B \vdash strexp \Rightarrow S \\ \text{(P3)} & S \text{ matches } (N_1)S_1 \text{ via } \varphi \\ \text{(P4)} & \dfrac{(N')S' = \varphi((N_1')S_1') \quad (N \text{ of } B) \cap N' = \emptyset}{B \vdash funid\texttt{(}strexp\texttt{)} \Rightarrow S'} \end{array}$$

First, at (P1), we look up the functor signature for *funid* in the basis. Then, at (P2), we elaborate *strexp* to give S, the actual argument structure. Then, at (P3), we check whether the actual argument matches the formal argument signature $(N_1)S_1$ via some realisation φ. (As we saw in Section 7.7, in all normal circumstances there is at most one such φ.)

There may be sharing between the argument and result in the functor signature; more precisely, there can be names in N_1 that occur free in $(N_1')S_1'$ besides occurring free in S_1. (An example is the functor `J` in the previous section.) Upon applying the functor, we wish to carry the identity of the types and structures from the actual argument through to the result of the application. For example, after the application

```
structure A = J(struct type t = int end)
```

the constructor `A.C` has type `int -> A.u`.

This propagation of sharing is achieved by applying φ to $(N_1')S_1'$, see (P4). (Recall that φ is the instantiation of the flexible names (N) to the names in the actual argument.)

By the second condition at (P4), in applying φ to $(N_1')S_1'$ we must pick the bound names (N') so that they are also disjoint from the names already in use in the basis. It is this condition which ensures that generative structure expressions and datatype declarations in the functor body are given fresh names upon each application.

To sum up, a functor application does not hide the identity of structures or types in the argument. However, note that the resulting structure S' has no more components than were present in the result part of the functor signature. In particular, components of the actual argument that were not specified in the argument signature are not propagated through to the result of the application. In the dynamic semantics, this is reflected by cutting down the actual argument to the shape of the argument signature, before evaluating the functor body.

> Looking at the rule for functor application, one naturally asks whether it really has to be that complicated. One might have expected something like
>
> $$\frac{B(\mathit{funid}) = (S, (N')S') \quad B \vdash \mathit{strexp} \Rightarrow S \quad (N \text{ of } B) \cap N' = \emptyset}{B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)} \Rightarrow S'}$$
>
> which looks more like the rule for ordinary function application. However, in order to avoid re-elaboration of structure expressions, we want the result S of the elaboration of *strexp* to be as specific as possible (see the discussion at the end of Section 6.2). But then by the above rule, *funid* could *only* be applied to S, and that is clearly not what is intended. Therefore, the functor signature for *funid* is *polymorphic*, in the sense that it can be instantiated each time the functor is applied. Hence the concept of *functor signature instance* [Sec 5.10, p 34] used in rule 55.

Renaming of bound names always preserves attributes (i.e. equality and arity) [Sec 5.1, p 31].[27] In particular, the equality properties of datatypes occurring in a functor body are determined once and for all at the time the functor is *declared*. Thus, if such a datatype is found not to admit equality, it will not admit equality in any application of that functor, no matter how much the types in the actual argument admit equality. For example, `A.u` is the above example does *not* admit equality.

One may be tempted to think of a functor as a kind of *macro*; is it perhaps the case that applying a functor is the same as elaborating the functor body after substitution of the actual argument for the formal parameter? Strictly speaking, this question is not entirely meaningful as it stands, for by substituting a structure

[27] There is a small mistake on [p 31], line 12 from the bottom: delete "imperative," — type names do not have an imperative attribute.

expression for a structure identifier one does not always obtain a grammatically correct phrase. However, the following exercise states the question accurately.

Exercise 8.3 The question is whether the following phrases would be equivalent, if we were to introduce local functor declarations:

$$\begin{aligned} \mathit{strexp}_1 \equiv\ & \texttt{let structure } \mathit{strid}\ \texttt{:}\mathit{sigexp}\ \texttt{=}\ \mathit{strexp} \\ & \texttt{in } \mathit{strexp}' \\ & \texttt{end} \end{aligned}$$

$$\begin{aligned} \mathit{strexp}_2 \equiv\ & \texttt{let functor } \mathit{funid}\texttt{(}\mathit{strid}\ \texttt{:}\mathit{sigexp}\texttt{) = } \mathit{strexp}' \\ & \texttt{in } \mathit{funid}\texttt{(}\mathit{strexp}\texttt{)} \\ & \texttt{end} \end{aligned}$$

assuming that *funid* does not occur free in *strexp*. Use the above observations about equality to show that this equivalence does *not* hold in general. (There are other reasons why the equivalence does not hold.)

Functor application is deterministic up to the choice of generative names. More precisely, let $(N_1)(S_1, (N_1')S_1')$ be $B(\mathit{funid})$. Let us assume that the functor application is part of an elaboration that starts out in the initial basis. Then $(N_1)S_1$ is type-explicit; for there are no functor signatures in the initial basis and whenever a functor signature is created, it is via rule 99 which in turn employs rule 65, which explicitly requires that the signature be type-explicit. Given that $B \vdash \mathit{strexp} \Rightarrow S$, there is at most one φ such that S matches $(N_1)S_1$ via φ, by Theorem 7.1. Hence there is at most one $(S'', (N')S')$ such that (P2) is satisfied, namely $(\varphi(S_1), \varphi((N_1')S_1'))$. Hence S' is fully determined by $B(\mathit{funid})$ and S, up to the choice of the bound names in $\varphi((N_1')S_1')$.

At the end of Section 3.4 on the dynamic semantics of functors, we noticed that the *evaluation* effect of signature ascription ": *sigexp*" in a structure declaration (at least at top-level) can be achieved just as well by applying a "curtailing" functor

$$\texttt{functor F(X: } \mathit{sigexp}\texttt{) = X}$$

This is also true for the *elaboration* effect; that is, after defining `F` as above, the two declarations

$$\texttt{structure A: } \mathit{sigexp}\ \texttt{ = }\ \mathit{strexp}$$

and

$$\texttt{structure A = F(}\mathit{strexp}\texttt{)}$$

either both fail or bind A to exactly the same structure.

Exercise 8.4 Prove this fact, as follows. Assume that elaboration begins in some B, and that *sigexp* elaborates in B to the signature $(N)S^*$. What functor signature gets bound to `F`? Suppose then that *strexp* elaborates to S. In

each structure declaration, what is the condition under which elaboration will be successful, and then what S' gets bound to `A`? Show that the answers are the same in each case.

With the notion of equivalence defined in Exercise 8.2, and with local functor declarations as suggested in Exercise 8.3, we would then expect to prove the following phrases equivalent:

`structure` *strid* `:` *sigexp* `=` *strexp*

and

`structure` *strid* `= let functor F(X:` *sigexp*`)=X in F(`*strexp*`) end`

provided that `F` does not occur in *strexp*. Thus all signature ascription can be explained in terms of functor argument signatures.

8.4 Variations and extensions

In Section 3.5, and again in Section 7.5, we briefly explored an alternative meaning for signature ascription, namely that it should have no curtailing effect, but should act simply as a conformity test; in our current terminology, this means just checking that a structure matches the signature. One thing we noticed was that, if this meaning is adopted for the argument signature of a functor, then there is difficulty with the present form of the `open` declaration. Let us now keep the meaning of functor argument signatures unchanged, and look at variations in the meaning of ascribing signatures in structure bindings, and to the results in functor bindings. (In fact, these are the only places where a signature is ascribed essentially to a structure *expression*.)

For the variation in which a matching check is done, without curtailment, let us use "`>:`" in place of the colon (by "`>`" we are trying to suggest that the structure may be *bigger* than the signature). Thus we introduce two new binding forms (multiple bindings omitted for brevity):

strbind ::= *strid* `>:` *sigexp* `=` *strexp*
funbind ::= *funid* `(` *strid* `:` *sigexp* `)` `>:` *sigexp*$'$ `=` *strexp*

Exercise 8.5 Give modified forms of the rules 55, 62 and 99 for these non-curtailing bindings.

In Section 3.5 we commented on another variation, which avoids curtailment *everywhere*, even on functor arguments; this indeed has some interest, despite the difficulty with `open`, but in that case we have to reformulate the semantics in a

non-trivial way, even changing the notion of realisation, and this takes us beyond the scope of our Commentary.[28]

A final variation is essentially the `abstraction` declaration offered by David MacQueen in his original Modules proposal, as an alternative to the `structure` declaration. Under the present semantics, a signature ascription may conceal components of a structure, but it will never conceal sharing – and other properties – of components that remain in view. This can even be misleading; consider the following example:

```
functor F(X: sig type T .. end): sig type T .. end
    = struct type T = X.T .. end
```

Perhaps the user expects that his result signature here conceals the nature of the result type `T` – i.e. conceals the fact that it is the same type as `X.T`, and has all the attributes that `X.T` enjoys. After all, *within* the functor body he knows no attributes possessed by the `T`-component of any structure to which `F` may be applied. But now suppose he applies the functor:

```
structure S2 = F(S1)
```

Then he will find that, if for example `S1.T` is a datatype with associated constructors, or admits equality, then he can take advantage of these attributes when handling objects of the new type `S2.T`.

This does not imply that in ML one cannot hide the implementation of a specified type. If one adopts the style of writing modules in the form of functors, then one can use the concealment provided by the argument signature of a functor. The modular style suggested in [Sec 3.6, p 15] would provide the same sort of concealment. All the same, as a matter of convenience at least, it may be desirable to add MacQueen's notion of *abstraction*. MacQueen originally proposed it for structure declarations only, using the keyword `abstraction` in place of `structure`; but, as we saw above, one may also wish to conceal the nature of a functor result, and we are therefore led to consider a third pair of binding forms. This time we use "`<:`" in place of the colon (the "`<`" now suggests that the structure is *diminished* even further than in the case of ":"; no doubt there are better notations!):

strbind ::= *strid* `<:` *sigexp* = *strexp*
funbind ::= *funid* `(` *strid* `:` *sigexp* `)` `<:` *sigexp′* `=` *strexp*

Exercise 8.6 Write down an inference rule which gives the semantics of the abstract structure binding. Then similarly define the semantics of the abstract

[28]The interested reader can find details of this semantics in Mads Tofte's PhD Thesis, *Operational semantics and polymorphic type inference*, CST-52-88, Computer Science Department, Edinburgh University, 1988.

functor binding. Finally express the abstract functor binding as a derived form of the normal functor binding and abstract structure binding.

Thus we have seen how to accommodate three very different meanings for signature ascription within the existing semantic framework. The abstraction version is used as the default in the specification system Extended ML of Sannella and Tarlecki[29].

8.5 Higher-order functors

As remarked earlier, functors cannot take functors as arguments, nor can they return functors as results. Now the question arises: is this limitation fundamental to the semantic theory built so far, or is it possible to extend the theory smoothly to cover the higher-order case?

If functors could be declared within structures (and specified in signatures), then functors, being mappings from structures to structures, would essentially be higher-order. The purpose of the present section is to outline a possible way to admit functors inside structures. This raises many interesting questions, some of which will be mentioned below. These questions have not yet been answered, so this section is purely speculative; it is not a proposal of any kind.

First let us recall from the Definition [Sec 5.15, p 45] the notion of functor signature matching, since matching a structure against a signature in the extended scheme must involve matching functor signatures against functor signatures. For reasons that will become apparent later on, we shall actually consider this concept as belonging to enrichment rather than to matching. Assume that

$$\Phi_1 \;=\; (N_1)(S_1, (N_1')S_1')$$

and

$$\Phi_2 \;=\; (N_2)(S_2, (N_2')S_2')$$

(Φ_1 can be thought of as the *specified* functor signature and Φ_2 as the *declared* functor signature). We then say that Φ_2 *enriches* Φ_1, written $\Phi_2 \succ \Phi_1$, if there exists a realisation φ such that

1. $(N_1)S_1$ matches $(N_2)S_2$ via φ, and
2. $\varphi((N_2')S_2')$ matches $(N_1')S_1'$.

The first condition ensures that the declared functor signature Φ_2 requires the argument of any application to have no more sharing, and no more richness, than was predicted by the specified signature Φ_1. The second condition ensures that the

[29] D. Sannella and A. Tarlecki, *Toward formal development of ML programs: foundations and methodology.* Report ECS-LFCS-89-71, LFCS, Dept. of Computer Science, Univ. of Edinburgh (1989); extended abstract in Proc. Joint Conf. on Theory and Practice of Software Development (TAPSOFT), Barcelona, Springer LNCS 352, 375–389 (1989)

declared functor signature Φ_2, instantiated to $(\varphi S_2, \varphi((N_2')S_2'))$, provides in the result of the application no less sharing, and no less richness, than was predicted by the specified signature Φ_1.

Note the contravariance in the argument position; that is, notice that matching goes the opposite way to the way it goes in the result position. Also note that this definition of functor signature enrichment is in terms of what it means for one signature to match another signature via a realisation [Sec 5.12, p 35]. In the Definition, enrichment and instantiation are defined independently and matching is then defined in terms of these. What is suggested now is to make all three concepts mutually recursive. An environment $E_1 = (F_1, SE_1, TE_1, VE_1, EE_1)$ then *enriches* another environment $E_2 = (F_2, SE_2, TE_2, VE_2, EE_2)$ if, in addition to the requirements already listed in [Sec 5.11, p 34]

$$\mathrm{Dom}\, F_1 \supseteq \mathrm{Dom}\, F_2, \text{ and } F_1(\mathit{funid}) \succ F_2(\mathit{funid}) \text{ for all } \mathit{funid} \in \mathrm{Dom}\, F_2$$

where F as usual ranges over functor environments.

Several notions generalise naturally to the extension. Since structures now contain functor environments, we shall consider two structures equal if they are identical up to the renaming of bound names. Assuming that one does not introduce new sharing equations, it seems reasonable to keep the notion of consistency exactly as it is. The definition of substructure [p 31] still applies; in particular, an occurrence of a structure S' in the functor environment of a structure S is not counted as a substructure of S. The definitions concerning admissibility [Sec 5.3 – Sec 5.5], signature instance and signature instantiation [Sec 5.9, p 34], functor signature instantiation [Sec 5.10] and signature matching [Sec 5.12, p 35] are not affected.

It is crucial that in all normal circumstances, for all signatures Σ and structures S, there is at most one φ such that S matches Σ via φ. At present, type-explication is sufficient to ensure this; but a stronger condition is necessary when signatures can contain functor signatures due to the contravariance mentioned above.

Finally, something interesting happens to the definition of principality. Note that in [Sec 5.13, p 35], principality is defined in terms of inferences of the form $B \vdash \mathit{sigexp} \Rightarrow S$. None of the rules by which one proves $B \vdash \mathit{sigexp} \Rightarrow S$ mentions the concept of principality. But if *sigexp* can contain functor specifications then some rules, for example rule 95 [p 43], will rely on rule 65, which *does* mention principality. Thus the definition of principality becomes mutally recursive with the definition of the inference rules. This will of course affect several of the theorems stated in this Commentary and give rise to interesting problems which we have not faced before.

Thus it appears that most of the theory scales well, but that some parts require careful rethinking before functors can be introduced as components of structures.

9 Admissible Semantic Objects and Proofs

We are now ready to develop the theory of semantic objects on which the static semantics of Modules relies.

Recall the relationship of elaboration between phrases and the semantic objects that model them:

$$\begin{array}{rcl} \text{Structure expressions} & \Longrightarrow & \text{Structures} \\ \text{Signature expressions} & \Longrightarrow & \text{Signatures} \end{array}$$

While every valid phrase elaborates to (at least) one semantic object, there are many semantic objects that we could call *monsters*, because they are useless even though the elaboration rules may permit them. For example, by elaborating a signature expression containing the sharing constraint `sharing A = A.Loopy`, one can obtain the signature

$$\{\mathtt{m}\}\,(\mathtt{m}, \{\mathtt{Loopy} \mapsto (\mathtt{m}, \{\})\})$$

which is a monster because it cannot be matched by any real structure (no real structure can have itself as a proper substructure). One could argue that this monster is harmless, because the mistake will be detected when an actual structure is matched against the signature. However, this might happen very late in the development process, especially if one uses functors extensively. A functor whose structure parameter is specified with a monster signature can never be applied to a real structure. To accept such functors as valid would be cruel, if we can think of a tractable criterion for excluding them.

We therefore define a notion of *admissibility* of semantic objects (and in certain cases of entire elaboration trees). We require that every object which occurs in an elaboration tree be admissible, and this is crucial in the reading of the rules. Technically, admissibility is a conjunction of three properties [p 33], namely *consistency*, *well-formedness* and *cycle-freedom*. These are discussed in the following sections.

In [Sec 5.5, p 32] it is stated that all semantic objects *mentioned* thereafter in the Definition are assumed to be admissible. We do not make this assumption in this Commentary, since we have to discuss the very question of when admissibility is needed.

9.1 Consistency

The semantic theory must allow that two structures (m_1, E_1) and (m_2, E_2) can share without being identical, i.e. $m_1 = m_2$ need not imply $E_1 = E_2$. The main reason for this is that signature constraints are coercive and preserve sharing, see Section 6.4. A signature constraint can restrict the view of a structure, but it always preserves sharing.

Structures that share need not be identical, neither statically, nor dynamically. However, if two structures S_1 and S_2 share and both contain a value component `x` say, then dynamically $S_1(\texttt{x})$ and $S_2(\texttt{x})$ are the same value, even if `x` denotes a reference. Statically a similar, albeit weaker, notion of consistency applies [Sec 5.2, p 32]:

First, an assembly A of type structures is said to be *consistent* if, for all (θ_1, CE_1) and (θ_2, CE_2) in A that share (i.e. $\theta_1 = \theta_2$), we have

$$CE_1 \text{ is empty or } CE_2 \text{ is empty or } \operatorname{Dom} CE_1 = \operatorname{Dom} CE_2$$

(The reason we do not require $CE_1 = CE_2$ in this definition will be explained below.)

Second, an assembly A of structures is *consistent*, if the set of all type structures occurring in A is consistent, and also for all structures S_1 and S_2 that occur in A, if S_1 and S_2 share then

(a) For any *strid*, if $S_1(\mathit{strid})$ and $S_2(\mathit{strid})$ both exist, then they share
(b) For any *tycon*, if $S_1(\mathit{tycon})$ and $S_2(\mathit{tycon})$ both exist, then they share

Exercise 9.1 Is the following structure environment consistent?

$$\begin{Bmatrix} \texttt{A} \mapsto \big(\texttt{m},\ \{\texttt{tree} \mapsto (\texttt{t}, \{\texttt{L} \mapsto \texttt{int} \to \texttt{t},\ \texttt{N} \mapsto \texttt{int} * \texttt{t} * \texttt{t} \to \texttt{t}\})\}, \\ \qquad\qquad \{\texttt{x} \mapsto \texttt{t}\}\big)\ , \\ \texttt{B} \mapsto \big(\texttt{m1},\ \{\texttt{A} \mapsto (\texttt{m2}, \{\texttt{tree} \mapsto (\texttt{t2}, \{\texttt{N} \mapsto \texttt{t2 list} \to \texttt{t2}\})\})\ , \\ \qquad\qquad \texttt{B} \mapsto (\texttt{m}, \{\texttt{tree} \mapsto (\texttt{t}, \{\})\})\} \big) \end{Bmatrix}$$

The ways in which structures are created preserve consistency. For example, starting from the initial basis, there is no structure binding that elaborates to the inconsistent structure environment

$$\begin{Bmatrix} \texttt{A} \mapsto \big(\texttt{m}, \{\texttt{C} \mapsto (\texttt{m1}, \{\})\}\big)\ , \\ \texttt{B} \mapsto \big(\texttt{m}, \{\texttt{C} \mapsto (\texttt{m2}, \{\})\}\big) \end{Bmatrix}$$

Hence one never has to check consistency of structures that result from elaborating structure expressions.

However, the rules for signature expressions and specifications do not automatically preserve consistency. For example, compare rule 53 [p 37] concerning generative structure expressions

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad m \notin (N \text{ of } B) \cup \operatorname{names} E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (m, E)}$$

with rule 63 concerning generative signature expressions

$$\frac{B \vdash \mathit{spec} \Rightarrow E}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)}$$

The first rule demands that m be chosen fresh, so that the new structure trivially is consistent with any structure whose name is already used (i.e. in N of B or in names E), whereas the second rule apparently admits any choice of m. The second rule is deliberately liberal, so that one is free to "guess" a name which will satisfy sharing constraints in some enclosing specification. However, such a guess must be made without violating consistency of the entire elaboration tree which proves $B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)$ [Sec 5.5, p 33].

Exercise 9.2 Guess the result of elaborating the following declaration:

```
signature SIG =
  sig
    structure A: sig  structure C: sig end  end
    structure B: sig  structure C: sig end  end
    sharing A = B
  end
```

After doing this exercise, you may object that there are other solutions which may be inconsistent, but which we should not regard as monster signatures because they can indeed be matched by real structures. An example is

$$\begin{aligned}\Sigma \;=\; \{\texttt{m1},\texttt{m2},\texttt{m3},\texttt{m4}\}\,\big(\texttt{m1},\ \{\ \ &\texttt{A} \mapsto (\texttt{m2}, \{\texttt{C} \mapsto (\texttt{m3}, \{\})\})\ \},\\ &\texttt{B} \mapsto (\texttt{m2}, \{\texttt{C} \mapsto (\texttt{m4}, \{\})\})\ \}\ \big)\end{aligned}$$

since, as will be clear in Chapter 7, Σ can be matched by any structure with `A` and `B` components which share and which both possess a `C` component. So why reject Σ? The reason is that Σ is a kind of *relative* monster; there is no real structure which matches Σ unless it *also* matches the consistent structure which is given as the answer to the above exercise. Thus consistency provides the user with more information about what a signature actually specifies.

The next example emphasises that in an elaboration $B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow S$, the result S must not only be *self*-consistent, but must also be consistent with the basis B in which the elaboration occurs:

Example 9.1

```
structure A = struct type t = int type u = bool end
structure A1: sig type t end = A
signature SIG =
```

```
sig
  structure B: sig type t type u end
  sharing B = A1
end
```

In the above declarations, `B` is specified to share with `A1` and the signature for `B` specifies `u`, which in fact is invisible in `A1`. Is the declaration of `SIG` legal? Yes, in fact it elaborates to precisely

$$\{\mathtt{m}\}\,(\mathtt{m}, \{\mathtt{B} \mapsto S_{\mathtt{A}}\})$$

where $S_{\mathtt{A}}$ is the structure bound to `A` in the basis. The point is that consistency is required not just between `B` and `A1`, but between `B` and `A` as well.

Any intersection of consistent sets is consistent. However, the union of two consistent sets need not be consistent. Indeed, there exist structures S_1, S_2, and S_3 for which S_1 and S_2 are consistent, S_2 and S_3 are consistent but S_1 and S_3 are not consistent.

Exercise 9.3 Give an example to show that consistency is not transitive.

Note that consistency places absolutely no demands on the types of constructors, variables or exception constructors. For example, the following signature

```
signature SIG =
  sig
    datatype t = C of int
    datatype u = C of bool
    sharing type t = u
  end
```

is perfectly legal although it cannot be matched by any real structure.

The reason consistency does not place demands on the types of constructors, variables and exceptions, thus admitting certain monster signatures, is that requiring equality of type schemes would make it impossible to find a single principal signature for every valid signature expression (the definition of principal signature [Sec 5.13, p 35] is discussed in Chapter 11). Consider for example the signature expression

```
sig
  structure A: sig type 'a t val x:  int t end
  structure B: sig val x:  int * int end
  sharing A = B
end
```

The problem now is to choose a type function θ for `t` such that θ applied to `int` is `int*int`. As it happens, there are four different solutions to this problem, namely $\theta = \Lambda$`'a.'a * 'a`, $\theta = \Lambda$`'a.'a * int`, $\theta = \Lambda$`'a.int * 'a`, $\theta = \Lambda$`'a.int * int`. No matter which one is chosen, it prevents perfectly good structures from matching the signature. With the actual definition of consistency, we simply bind `t` to a unary type name; the instantiation of `t` can then happen each time a structure is matched against the signature.

9.2 Well-formed signatures

A free occurrence of a structure or type name n in a signature $\Sigma = (N)S$ signifies sharing with one or more real structures. It is therefore reasonable to require of Σ that for every structure (m, E) which occurs in S, if m is free (i.e. $m \notin N$) then all the structure and type names occurring in E are free as well. Another natural property is that $N \subseteq \text{names}\, S$, i.e. that there are no spurious bound names. Taken together, these properties make up the well-formedness property defined in [Sec 5.3, p 32]. Although well-formedness is required of all signatures that partake in elaboration, it suffices to check for well-formedness at one single point, namely in rule 65, [p 39], where new signatures can be introduced. (See Examples 11.2 and 11.3 for signatures that will be caught by this check.)

There is also a notion of well-formed functor signature, see [Sec 5.3, p 32]. An assembly A is well-formed if every type structure [Sec 4.9, p 21], signature and functor signature occurring in A is well-formed [Sec 5.3, p 32].

Although well-formedness is explicitly required of all objects in an elaboration, it is not clear how necessary this is; for to a considerable extent well-formedness of a signature is ensured by other conditions which are imposed when it is built. For example, consider

$$\Sigma \;=\; \{\texttt{m2}\}\Big(\texttt{m1}, \{\texttt{A} \mapsto (\texttt{m2}, \{\})\}\Big)$$

It is ill-formed because `m1` is free but `m2` bound. Now if Σ is the result of an elaboration in basis B, and if B contains any structure S with name `m1` which has an `A` component, then Σ must be inconsistent with B (because the bound name `m2` differs from the free name of the `A` component of S, by the convention about renaming in the definition of consistency [Sec 5.2, p 32]). Thus Σ can only be consistent with B if B contains *no* structure named `m1` and having an `A` component. But then there is no way of ever building a real structure which can match Σ! (Any structure withan `A` component will have to be newly generated, and hence will possess a name different from `m1`.) In other words, Σ is a monster. In fact, the requirement that a signature be *covered* by the basis in which it is built [Sec 5.13, p 35] is designed to reject such monsters. Therefore consistency and covering together ensure that a signature cannot be ill-formed in the sense that Σ above is ill-formed.

All the same, there is another sense in which well-formedness is *not* guaranteed by other required properties (this point is taken further in Chapter 11). We are not sure whether this other kind of ill-formedness is harmful; for the present, we prefer ML to wear both belt (consistency and covering) and braces (well-formedness).

9.3 Cycle-freedom

The requirement of cycle-freedom [Sec 5.4, p 32] is imposed to bar cyclic monster signatures, such as would result from the following declaration:

```
signature SIG =
  sig
    structure A: sig  structure Loopy: sig end  end
    sharing  A = A.Loopy
  end
```

We discussed such monsters at the beginning of the chapter. Like consistency, cycle-freedom is a property of an assembly of semantic objects rather than a single object. The larger the assembly, the harder it is to satisfy consistency and cycle-freedom. The check for cycle-freedom is a part of the admissification process discussed ion the following section. One never has to check for cycle-freedom in structures obtained from elaborating structure expressions.

9.4 Admissibility

An object or assembly A is admissible if it is consistent, well-formed and cycle-free [Sec 5.5, p 33]. It is easy to prove that if some assembly A of objects is admissible and $A' \subseteq A$, then A' is admissible. On the other hand, the union of admissible sets need not be consistent, or cycle-free, although it will be well-formed.

It is important to note that realisation does not preserve admissibility; one can easily find φ and admissible S such that φS is neither consistent, nor well-formed, nor cycle-free. However, *inverse* realisation is better-behaved:

Theorem 9.1 (Inverse Realisation) Let φ be a realisation, and A be any assembly. Then if φA is well-formed, so is A; and if φA is cycle-free, so is A.

We shall need this result in Section 10.4.

It is unnecessarily restrictive to require that the assembly of all semantic objects occurring in the elaboration tree of an entire program be admissible. We do, however, impose this requirement on the elaboration tree for a signature expression which proves a sentence of the form $B \vdash \mathit{sigexp} \Rightarrow S$ [Sec 5.5, p 33]. This is

stronger than merely requiring that the *conclusion* of the tree – i.e. the assembly (B, S) – be admissible. Without the stronger assumption we are unable to prove the existence of principal signatures (Chapter 11).

Readers who are interested in how and when one checks admissibility during elaboration are referred to Section 10.5.

Exercise 9.4 Does the following signature declaration elaborate?

```
signature SIG =
  sig
    structure A: sig
      structure B: sig
        structure C: sig end end end
    structure D: sig end sharing D = A
    structure E: sig end sharing E = A.B.C
    sharing D = E
  end
```

10 Elaboration of Signature Expressions

In this chapter and the next we comment on the inference rules for signature expressions and specifications. We begin in Section 10.2 with a discussion of all the rules except one, namely rule 65, the one which infers the *principal* signature for a signature expression; this crucial rule is treated separately in Chapter 11. In Section 10.3 we prove a theorem which states that, informally speaking, elaborations which use the rules discussed in Section 10.2 are closed under realisation. In Section 10.4 we prove a theorem that determines, for any sharing specification, the *most general* realisation which meets the demands of that specification, as long as they can be met without violating admissibility. Finally, in Section 10.5 we summarise exactly where admissibility must be checked during elaboration.

First, we review the notion of a *basis*.

10.1 The basis

Almost all Modules phrases are elaborated in a *basis*

$$B = (N, F, G, E)$$

where N as usual is a name set (M, T) [Fig 11, p 31], and E as usual is an environment [Fig 10, p 17]. F is a *functor environment*, i.e. a finite map from functor identifiers to functor signatures [Fig 11, p 31] and G is a *signature environment*, i.e. a finite map from signature identifiers to signatures [Fig 11, p 31]. When the elaboration of a Modules phrase involves the elaboration of a Core phrase, one can extract a context, C of B, from B as follows:

$$C \text{ of } B = (T \text{ of } N, \emptyset, E)$$

i.e. one ignores M, F and G and inserts an empty set of explicit type variables [Sec 5.1, p 31]. The structure name set M in a basis B plays a rôle in Modules elaboration similar to that played by T in a context C in Core elaboration; M is the set of *rigid* structure names, i.e. names of real structures, just as T is the set of names of rigid type names.

The beginning of [Sec 5.14, p 37] states that it is "intended" that every basis B in which a *topdec* is elaborated has the property that $\text{names}\, B \subseteq N$ of B. Let us say that B is *rigid* if it has this property. In fact, we can easily prove that this intention is satisfied for the execution of programs, assuming that this starts in the initial basis B_0 [App C,D]. For first note that B_0 – now meaning the initial *static* basis – is certainly rigid, by inspection of [App C]. Now suppose that $\mathit{program}_0 = \mathit{topdec}$; $\mathit{program}_1$ and that $\mathit{program}_0$ is elaborated in B_0. Then we can see from rule 196 [p 64] that *topdec* is elaborated in (rigid) B_0, yielding a basis B say; then $\mathit{program}_1$ is elaborated in $B_1 = B_0 \oplus B$, which is in turn rigid due to the use of $\oplus$. Hence, by iterating this argument, every *topdec* in a program is elaborated in a rigid basis.

10.2 The rules

In this section we shall treat rules 63, 64 and 70–90; together with rules 47–52 (for types) they constitute the set of rules – which we shall call $\mathcal{R}_{\texttt{sig}}$ – which are involved in any elaboration concluding with a sentence of the form $B \vdash \mathit{sigexp} \Rightarrow S$. Actually, it is revealing to notice that these rules use neither N nor F; that is, the elaborations which are possible do not depend upon these components. (In particular, there are no side-conditions which reject the choice of certain names because they appear in N of B.) In fact – though it was not worth the trouble – the rules could have been formulated in terms of a cut-down basis containing only (G, E) of B, just as a context C is a cut-down basis containing all that is needed for Core elaborations. By contrast however, N plays an important role in the elaboration of *principal* signatures discussed in Chapter 11. Also common to the rules $\mathcal{R}_{\texttt{sig}}$ is that the results of elaboration contain no bound structure names or type names. For example, signature expressions elaborate to structures (!) and specifications elaborate to environments.

The rules are *liberal* in the sense that when they prove $B \vdash \mathit{phrase} \Rightarrow A$, the semantic object A is often not uniquely determined by B and *phrase*. The freedom lies only in the choice of names and type functions in A. We have already seen that rule 63

$$\frac{B \vdash \mathit{spec} \Rightarrow E}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)}$$

allows any choice of m, within the limits of admissibility, of course. Similarly, rule 64 for signature expressions

$$\frac{B(\mathit{sigid}) \geq S}{B \vdash \mathit{sigid} \Rightarrow S}$$

allows freedom in choosing S, as long as S really is an instance of $B(\mathit{sigid})$ and (B, S) is admissible. The three other rules that permit freedom of choice are rule 79 for `include` and rules 83 and 84 for type and datatype descriptions. This freedom allows one to choose names and type functions that will satisfy sharing constraints elsewhere in the signature. Thus, when elaborating signature expressions and specifications, one has to guess structure names and type functions, much as one has to guess types when elaborating phrases in the Core. As we shall see later, there is a systematic way of finding structure names and type functions that satisfy the sharing constraints in question, if the constraints can be satisfied at all, using a particular kind of unification algorithm. However, for the purpose of the inference rules, the guessing relieves us of the burden of spelling out the details of unification and rules 88 and 89 for sharing equations [p 42] become remarkably simple:

$$\frac{m \text{ of } B(\mathit{longstrid}_1) = \cdots = m \text{ of } B(\mathit{longstrid}_n)}{B \vdash \mathit{longstrid}_1 = \cdots = \mathit{longstrid}_n \Rightarrow \{\}}$$

$$\frac{\theta \text{ of } B(\mathit{longtycon}_1) = \cdots = \theta \text{ of } B(\mathit{longtycon}_n)}{B \vdash \texttt{type}\ \mathit{longtycon}_1 = \cdots = \mathit{longtycon}_n \Rightarrow \{\}}$$

The side-conditions on these two rules, and on rules 72 and 83, are the only direct constraints upon the freedom of choice which we have just discussed. However, the requirement of admissibility is an indirect constraint; when two structures share, it will force any structure components or type components which they have in common to share as well. The fact that rule 88 only requires equality of names, rather than equality of structures, allows different but consistent views to coexist. Similarly, a type structure with an empty constructor environment can share with a type structure with a non-empty constructor environment.

Notice that rule 72 forces the elaboration to choose equality type names for type constructors that are specified with `eqtype`. The rules for `datatype` and `type` specifications are not explicit about whether or not the chosen type functions should admit equality; that can depend on sharing specifications present elsewhere in the signature expression (see Chapter 11 for a more detailed discussion of the equality attribute).

Exercise 10.1 Elaborate the signature expression below to a structure. Which of the types `t`, `u` and `v` will admit equality?

```
sig
  type t
  eqtype u
  datatype v = C | D
  sharing type t = u = v
end
```

Exercise 10.2 The signature declaration

```
signature SIG =
  sig
    structure A: sig end
    structure B: sig end
  end
```

yields the signature

$$\{\mathtt{m1}, \mathtt{m2}, \mathtt{m3}\}\ \Big(\mathtt{m1}, \{\mathtt{A} \mapsto (\mathtt{m2}, \{\}), \mathtt{B} \mapsto (\mathtt{m3}, \{\})\}\Big)$$

Does the expression below elaborate? (Hint: the semantics of `include` is defined by rule 79 [p 41].)

```
sig
  include SIG
  sharing A = B
end
```

Exercise 10.3 Given that `Str` is declared by

```
structure Str =
  struct
    structure A = struct end
    structure B = struct end
  end
```

does the signature expression below elaborate?

```
sig
  open Str
  sharing A = B
end
```

10.3 The realisation theorem

As mentioned earlier, there exists a systematic way of finding structure names and type functions that satisfy sharing constraints. The basic idea is to choose fresh names whenever possible and then identify different names when it is found from sharing specifications that they ought to be identical. The reason one can proceed in this manner is that elaboration of signature expressions is closed under realisation; the remainder of this section is devoted to expressing this fact precisely as a theorem, and proving the theorem.

From now on we shall use ∇ to range over inference trees; recall from Section 1.1 that such a tree is built from rule instances whose side-conditions are satisfied. This does not require anything to be admissible. We now define an *admissible* inference tree to be one which satisfies the admissibility conditions in [Sec 5.5, p 33]. We shall refer to admissible inference trees also as *proofs*. Let $A \vdash \mathit{phrase} \Rightarrow A'$ be the root sentence of a proof ∇ (and let $\mathcal{R}$ be a set of rules including all those used in ∇); then we say that ∇ *proves* $A \vdash \mathit{phrase} \Rightarrow A'$ (*from* $\mathcal{R}$). We shall sometimes write for example "Assume $A \vdash \mathit{phrase} \Rightarrow A'$" to mean "Assume that ∇ proves $A \vdash \mathit{phrase} \Rightarrow A'$ for some ∇". ($\mathcal{R}$ will be clear from the context.)

The first thing we have to do is to define how to apply a realisation φ to a proof ∇. Recall from Section 7.2 that a realisation $\varphi = (\varphi_{\mathrm{Ty}}, \varphi_{\mathrm{Str}})$ acts upon type names and structure names. It is extended naturally to act upon other semantic objects, for example $\varphi(m, E) = (\varphi m, \varphi E)$, but two cases need special treatment. For an object with bound names, such as a signature $\Sigma = (N)S$, the bound name set N is first changed if necessary to become disjoint from $\mathrm{Supp}\,\varphi \cup \mathrm{Yield}\,\varphi$; then $\varphi\Sigma = (N)(\varphi S)$ [Sec 5.7, p 33]. Thus bound names are immune to φ. Also for the typename set T in a context, and the name set N in a basis, we define

$$\begin{aligned} \varphi T &= \bigcup\{\mathrm{names}(\varphi t)\ ;\ t \in T\} \\ \varphi N &= \bigcup\{\mathrm{names}(\varphi n)\ ;\ n \in N\} \end{aligned}$$

This ensures, for example, that $\varphi B = \varphi N, \varphi F, \varphi G, \varphi E$ is indeed a basis.

Now, to apply φ to a proof ∇, one simply applies φ to every semantic object in ∇. While this does yield a tree $\varphi\nabla$, the tree may not be a proof. For one thing, it may well contain inadmissible objects; see also the discussion preceding Theorem 9.1. But even if it is admissible, it may still not be a proof. To see this, consider any proof ∇ involving rule 53 [p 37] for generative structure expressions; the side-condition requiring m to be new may be satisfied in ∇ but violated in $\varphi\nabla$.

However, the following theorem shows that the latter phenomenon – that $\varphi\nabla$ is admissible but still not a proof – cannot arise in elaborating signatures. Such elaboration only involves the rules in $\mathcal{R}_{\mathtt{sig}}$, and for these we can prove:

Theorem 10.1 (Realisation) *Assume that* ∇ *proves* $A \vdash \mathit{phrase} \Rightarrow A'$ *from* $\mathcal{R}_{\mathtt{sig}}$*, and let* $\varphi\nabla$ *be admissible. Then* $\varphi\nabla$ *proves* $\varphi A \vdash \mathit{phrase} \Rightarrow \varphi A'$ *from* $\mathcal{R}_{\mathtt{sig}}$*.*

To this end, we need the following lemma:

Lemma 10.2 *For any signature* Σ*, structure* S *and realisation* φ*, if* $\Sigma \geq S$ *then* $\varphi\Sigma \geq \varphi S$*.*

Proof Write Σ as $(N)S'$, assuming w.l.o.g. that $(\mathrm{Supp}\,\varphi \cup \mathrm{Yield}\,\varphi) \cap N = \emptyset$. Then $\varphi\Sigma = \varphi((N)S') = (N)\varphi S'$. So it suffices to find a realisation ψ such that $\mathrm{Supp}\,\psi \subseteq N$ and $\psi\varphi S' = \varphi S$.

Now since $\Sigma \geq S$ there exists ψ' such that $\mathrm{Supp}\,\psi' \subseteq N$ and $\psi' S' = S$. Define ψ to be the restriction of $\varphi \circ \psi'$ to N, i.e. $\psi n = \varphi(\psi' n)$ if $n \in N$ and $\psi n = n$ if $n \notin N$.

To prove $\psi\varphi S' = \varphi S$, it is enough to show that $\psi\varphi n = \varphi\psi' n$ for every $n \in \mathrm{names}\,S'$; this is now straightforward, considering the two cases $n \in N$ and $n \notin N$ separately.

Proof (of Theorem 10.1)

By induction on the depth of inference. Rules 47–52 are straightforward; the only interesting case is rule 49 [p 29], where one uses that type realisation and type function application commute, i.e. that $\varphi(\tau^{(k)}\theta) = (\varphi\tau^{(k)})(\varphi\theta)$.

Rule 63 [p 39] is a trivial inductive case and the case for rule 64 follows directly from Lemma 10.2.

In the case for rule 70 [p 40], we use that closure commutes with realisation: for any closed type scheme $\sigma = \forall\alpha^{(k)}.\tau$, we have $\varphi\sigma = \forall\alpha^{(k)}.\varphi\tau$, which is equivalent to the closure of $\varphi\tau$ although $\forall\alpha^{(k)}.\varphi\tau$ may contain spurious bound type variables.

The case for rule 71 is a straightforward inductive case.

For rule 72, we need to note that if θ admits equality then $\varphi\theta$ will admit equality as well, by the definition of type realisation.

Rules 73–82 are easy, using Lemma 10.2 at rule 79. For rule 83 we need

to note that the arity of $\varphi\theta$ is the same as the arity of θ, since $\varphi(\Lambda\alpha^{(k)}.\tau) = \Lambda\alpha^{(k)}.\varphi\tau$, although not all the type variables in $\alpha^{(k)}$ need occur in $\varphi\tau$.

Rule 84 is more interesting [p 42]. *CE* is non-empty, by rule 85. Moreover, $\varphi(t, \mathrm{Clos}CE) = (\varphi t, \mathrm{Clos}(\varphi CE))$. Since $\varphi\nabla$ is admissible, $(\varphi t, \mathrm{Clos}(\varphi CE))$ must be well-formed. Since $\mathrm{Clos}(\varphi CE)$ is non-empty, this implies that φt must be a type name, cf. [Sec 4.9, p 21]. Thus $\varphi(C, \alpha^{(k)}t)$ is of the form $(\varphi C, \alpha^{(k)}t')$, for some t', and the result follows by induction.

Rule 85 is straightforward. For rule 86 we need to note that $\mathrm{tyvars}(\varphi\tau) \subseteq \mathrm{tyvars}\,\tau$. The remaining rules, 87–90, offer no resistance.

It is instructive to compare this theorem with Theorem 5.1, which states that Core elaboration is closed under substitution for type variables.

10.4 Admissification

The topic of this section is a process called *admissification*, which is a kind of unification of structures and type structures. The existence of so-called principal admissifiers is crucial to the existence of principal signatures, so implementers need to know what they are and how to compute them. We shall not give any program that computes principal admissifiers, but we give a constructive mathematical proof that they exist under certain natural conditions.

The need for admissification arises in elaborating sharing specifications. For example, consider the signature expression

```
sig
  structure A: sig structure C: sig end
                   structure D: sig end
               end
  structure B: sig structure D: sig end
                   structure E: sig end
               end
  sharing A = B
end
```

Having elaborated the specifications of `A` and `B`, just before dealing with the sharing specification, we have obtained the following structure environment:

$$\{\ \mathtt{A} \mapsto (\mathtt{m1}, \{\mathtt{C} \mapsto (\mathtt{m2}, \{\}), \mathtt{D} \mapsto (\mathtt{m3}, \{\})\}),\\ \mathtt{B} \mapsto (\mathtt{m4}, \{\mathtt{D} \mapsto (\mathtt{m5}, \{\}), \mathtt{E} \mapsto (\mathtt{m6}, \{\})\})\ \}$$

Because of the sharing specification, we now wish to identify `m1` and `m4`. However, in order to do that without violating consistency, we must also identify `m3` and `m5`. By admissification one can obtain the realisation $\varphi^* = \{\mathtt{m1} \mapsto \mathtt{m4},\ \mathtt{m3} \mapsto \mathtt{m5}\}$ which,

when applied to the structure environment yields

$$\begin{array}{ll}\{ & \mathtt{A} \mapsto (\mathtt{m4}, \{\mathtt{C} \mapsto (\mathtt{m2}, \{\}), \mathtt{D} \mapsto (\mathtt{m5}, \{\})\}), \\ & \mathtt{B} \mapsto (\mathtt{m4}, \{\mathtt{D} \mapsto (\mathtt{m5}, \{\}), \mathtt{E} \mapsto (\mathtt{m6}, \{\})\}) \ \}\end{array}$$

The above example might give the false impression that in order to check a sharing constraint, one simply looks up the structure (or type) identifiers in question and unifies them by recursive descent along common paths. The reason this impression is false is that the unification also has to take into account every structure (or type structure) in the entire elaboration tree that shares with one of the structures (or type structures) mentioned in the sharing constraint. Indeed, admissification is an operation on an entire assembly of objects, where one attempts to turn a inadmissible assembly into an admissible assembly by applying realisations. In the above example, we would start out from the inadmissible assembly

$$\begin{array}{ll}(B, A, & (\mathtt{m4}, \{\mathtt{C} \mapsto (\mathtt{m2}, \{\}), \mathtt{D} \mapsto (\mathtt{m3}, \{\}), \\ & (\mathtt{m4}, \{\mathtt{D} \mapsto (\mathtt{m5}, \{\}), \mathtt{E} \mapsto (\mathtt{m6}, \{\}) \ )\end{array}$$

where B is the basis, and A is an assembly containing all the semantic objects present in the elaboration tree so far. In this example, the above φ^* is an admissifier (assuming that (B,A) is already admissible), because there is no sharing between (B,A) and the other two components. An example of admissification involving objects in the basis other than the ones referred to directly in the sharing equations was given in Section 9.1, Example 9.1.

In seeking to admissify an assembly when computing a signature in a basis B, one seeks an admissifier φ whose support is disjoint from N of B, since the names in B are to be kept fixed. In general, for any name set N we say that φ is *fixed on* N if $\mathrm{Supp}\,\varphi \cap N = \emptyset$.

Now let A be an assembly, not necessarily admissible. A realisation φ is said to be an *admissifier for* A *under* N if φA is admissible and φ is fixed on N. An admissifier φ^* for A under N is said to be *most general* or *principal* if, for every admissifier φ for A under N, there exists a realisation φ' fixed on N such that $\varphi A = \varphi'\varphi^* A$. (Henceforward we shall write e.g. $\varphi'\varphi^* A$ for $\varphi'(\varphi^*(A))$ to avoid too many parentheses.) Intuitively, a realisation is principal if it identifies as few structure names and type functions as necessary.

For given A and N, although there may be an admissifier for A under N, there may not be a principal one. Take for example

$$A \ = \ \Big\{ \big(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{int}, \{\})\}\big), \big(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{int}\,\mathtt{t}, \{\})\}\big) \Big\}$$

(where `t` has arity 1) and $N = \emptyset$. There are essentially only two admissifiers: $\{\mathtt{t} \mapsto \Lambda\mathtt{'a.'a}\}$ and $\{\mathtt{t} \mapsto \Lambda\mathtt{'a.int}\}$, and neither is principal. (This embarrassment can arise if we allow a richer class of sharing equations, as we discussed at the end of Section 6.3.) The problem is that A contains a type structure $(\theta, CE) = (\mathtt{int}\,\mathtt{t}, \{\})$ in which θ is not simply a type name, but contains a type name not

in N. Motivated by such examples, we say that A is *grounded in* N if (after disjoining the bound names in A from N) for every occurrence of a type structure (θ, CE) in A not within a functor signature[30], either θ is simply a type name or tynames$(\theta) \subseteq N$. Further, we say that a basis B is *grounded* if it is grounded in N of B.

The following theorem ensures that, if an assembly is indeed grounded in some N, and has an admissifier, then it will have a principal one. The language is such that elaboration need only involve grounded semantic objects,[31] and hence the groundedness condition of the theorem will always be satisfied when required in the proof of the principality theorem, Theorem A.2 in Appendix A.

Theorem 10.3 (Admissification) *Let A be grounded in N, and let φA be admissible for some φ fixed on N. Then there exists a principal admissifier φ^* for A under N. Moreover, $\varphi^* A$ is grounded in N.*

Proof Let A,N and φ satisfy the assumptions. In the proof we shall refer to names in N as *rigid*, and other names as *flexible*. We are going to define φ^* by first defining an equivalence relation $\sim$ on strnames $A \cup$ tyfuns$A$, partitioned into equivalence relations $\sim_{\mathrm{Str}}$ on strnames A and $\sim_{\mathrm{Ty}}$ on tyfuns$A$. Here tyfuns$A$ means the set of type functions that occur free in A (the meaning of this should be clear, given that A is grounded in N). Then we make φ^* map each name to a suitable member of its equivalence class.

We obtain $\sim$ as a limit, by iterating a function F over such partitioned equivalences. F is defined as follows: $F(\equiv)$ is the least equivalence $\equiv'$ such that, whenever A contains two structures (m_1, E_1) and (m_2, E_2) such that $m_1 \equiv m_2$, then

(i) If $\mathit{strid} \in \mathrm{Dom}(E_1) \cap \mathrm{Dom}(E_2)$, then m of $E_1(\mathit{strid}) \equiv' m$ of $E_2(\mathit{strid})$
(ii) If $\mathit{tycon} \in \mathrm{Dom}(E_1) \cap \mathrm{Dom}(E_2)$, then θ of $E_1(\mathit{tycon}) \equiv' \theta$ of $E_2(\mathit{tycon})$

It is easy to see that F is monotonic, in fact continuous, with respect to inclusion of relations.

Let $\sim$ be the least fixed point of F. (It is also the least *post* fixed point, i.e. least such that $F(\sim) \subseteq \sim$.) Starting from the identity equivalence Id, it can be computed as the least upper bound of the chain

$$\mathrm{Id} \subseteq F(\mathrm{Id}) \subseteq \cdots \subseteq F^n(\mathrm{Id}) \subseteq \cdots$$

Now define the partitioned equivalence $\equiv_\varphi$ as follows:

$$m \equiv_\varphi m' \Leftrightarrow \varphi m = \varphi m' \quad \text{and} \quad \theta \equiv_\varphi \theta' \Leftrightarrow \varphi\theta = \varphi\theta'$$

[30] No constraint is necessary upon functor signatures.

[31] This is partly because type-sharing equations are confined to type constructors, and partly because consistency places no constraint upon variable and exception environments.

Since φA is consistent, we have $F(\equiv_\varphi) \subseteq \equiv_\varphi$, i.e. $\equiv_\varphi$ is a post fixed point for F. But $\sim$ is the least post fixed point, so we have $\sim \subseteq \equiv_\varphi$. That is,

$$m \sim m' \Rightarrow \varphi m = \varphi m' \text{ and } \theta \sim \theta' \Rightarrow \varphi\theta = \varphi\theta'$$

To define φ^*_{Str}, consider any equivalence class $[m]$ of $\sim_{\mathrm{Str}}$. We claim that there is at most one $m_0 \in [m]$ which is not flexible. For if there are two such, say $m_0 \sim m_1$, then we have $\varphi m_0 = \varphi m_1$. But $\varphi m_0 = m_0$ and $\varphi m_1 = m_1$ since φ is fixed on N; hence $m_0 = m_1$.

Take this m_0 if it exists; otherwise let m_0 be any member of $[m]$. Set $\varphi^*_{\mathrm{Str}}(m') = m_0$ for all $m' \in [m]$. Then φ^*_{Str} is fixed on N as required.

For φ^*_{Ty}, take any equivalence class $[\theta]$ of $\sim_{\mathrm{Ty}}$. Because $\sim \subseteq \equiv_\varphi$ and because φA is admissible, all members of $[\theta]$ have the same arity. We claim that there is at most one $\theta_0 \in [\theta]$ such that θ_0 is not both a type name and flexible. For if there are two such, say $\theta_0 \sim \theta_1$, then we have $\varphi\theta_0 = \varphi\theta_1$; but also $\mathrm{tynames}\,\theta_i \subseteq N$ $(i = 0, 1)$ because A is grounded in N; hence $\varphi\theta_0 = \theta_0$ and $\varphi\theta_1 = \theta_1$ since φ is fixed on N; hence $\theta_0 = \theta_1$.

Assume such a θ_0 exists. If θ_0 does not admit equality then, since $\varphi\theta_0 = \theta_0$, no member of $[\theta_0]$ admits equality. Now set $\varphi^* t = \theta_0$ for all $t \in [\theta_0] \setminus \{\theta_0\}$; this makes sense, because all members of $[\theta_0]$ except θ_0 are flexible type names. Thus $\varphi^*\theta' = \theta_0$, for all $\theta' \in [\theta_0]$.

Otherwise, i.e. if no such θ_0 exists, every member of $[\theta]$ is a flexible type-name. Let t_0 be a member which admits equality, if one exists; otherwise, let t_0 be any member. Set $\varphi^* t = t_0$, for all $t \in [\theta]$.

Clearly, φ^*_{Ty} is a type realisation, fixed on N (and incidentally idempotent). Moreover φ^* is clearly independent of φ, and we can define φ' such that $\varphi'\varphi^* A = \varphi A$ simply by setting $\varphi' = \varphi$; for it is easy to verify that $\varphi\varphi^* n = \varphi n$ for all $n \in \mathrm{names}\, A$.

We now wish to prove that $\varphi^* A$ is admissible. First, it is well-formed and cycle-free by Theorem 9.1. Moreover, $\varphi^* A$ is consistent: conditions 1 and 2 of the definition of consistency [Sec 5.2, p 32] are satisfied because $\sim$ is a post fixed point of F; condition 3, that the set of type structures in $\varphi^* A$ is consistent, follows from the fact that φA is consistent and $\sim \sqsubseteq \equiv_\varphi$. This proves that $\varphi^* A$ is admissible, as required.

Finally $\mathrm{tyfuns}(\varphi^* A) \subseteq \mathrm{tyfuns} A$, so $\varphi^* A$ is grounded in N as desired.

For implementation purposes, one is also interested in discovering that *no* admissifier exists, in order that an error may be reported. The following exercise concerns failing admissifications.

Exercise 10.4 Assume that A is grounded in N, but that it is not known whether an admissifier for A under N exists. By a careful inspection of the proof of Theorem 10.3, find all the places where explicit checks are necessary to ensure that φ^* is well defined and is an admissifier.

10.5 Checking admissibility

We shall now summarise in terms of the inference rules precisely where in practice one needs to check admissibility in order to satisfy the global admissibility requirements [Sec 5.5, p 33].

We claim without proof that all rules that are not in $\mathcal{R}_{\texttt{sig}} \cup \{\text{rule } 65\}$ preserve admissibility and therefore require no check for admissibility.

Let us therefore consider rule 65 and the members of $\mathcal{R}_{\texttt{sig}}$. Well-formedness must be checked at rule 65, see Section 11.3. It suffices to check consistency, cycle-freedom and well-formedness of type structures once for each sharing specification. It is not in general true to the Definition to delay the checking of consistency, cycle-freedom and well-formedness of type structures till the entire signature has been elaborated, since inadmissibility can be introduced locally, as in

`sig local` *spec* `in end end`

where *spec* specifies an inconsistent structure. (Recall from the end of Section 9.4 that we require the assembly of all semantic objects in a signature elaboration to be admissible.)

Exercise 10.4 is concerned with exactly how admissibility is checked during admissification.

11 Principal Signatures

We shall now motivate and analyse the definitions in [Sec 5.13, p 35–36] concerning principal and equality-principal signatures. The proofs of Theorems 11.1 and 11.2, the key theorems leading to the existence of principal signatures, are deferred to Appendix A.

We saw in Chapter 10 that the rules for inferring sentences of the form $B \vdash \mathit{sigexp} \Rightarrow S$ allow one to guess names that will satisfy sharing constraints later in the elaboration. Yet, whenever we wish to infer a signature $(N)S$ for *sigexp*, we would like the sharing in S to be just enough to satisfy the sharing specified in *sigexp*, without identifying names unnecessarily, so that $(N)S$ really is matched by any structure which satisfies the sharing specified in *sigexp*. Informally, $(N)S$ is principal for *sigexp* in B, if S is the result of elaborating *sigexp* in B choosing names in such a way that two name-occurrences in the signature are identical only if they have to be, and every name is flexible if it can be.

Recall from the beginning of Chapter 9 that in this Commentary we depart from the convention introduced in [Sec 5.5, p 33] that all semantic objects which are mentioned are admissible. In particular, we allow the notation $(N)S$ for a signature which is not necessarily admissible. The admissibility conditions on *proofs* (Section 10.3) still apply, however.

11.1 Bare principality

We shall begin with a slightly simpler notion than principality, which we shall call *bare principality*. A signature (not necessarily admissible) $(N)S$ is *barely principal for sigexp in* B if, choosing N so that $(N \text{ of } B) \cap N = \emptyset$,

1. $B \vdash \mathit{sigexp} \Rightarrow S$
2. Whenever $B \vdash \mathit{sigexp} \Rightarrow S'$, then $(N)S \geq S'$

As for the converse of the implication in condition 2, suppose that $(N)S \geq S'$ via φ and names $B \cap N = \emptyset$. Then $\varphi B = B$ and $\varphi S = S'$, so if ∇ proves $B \vdash \mathit{sigexp} \Rightarrow S$ and $\varphi\nabla$ is admissible then $\varphi\nabla$ proves $B \vdash \mathit{sigexp} \Rightarrow S'$, by Theorem 10.1.

Note that condition 1 ensures that $(N)S$ will be both consistent and cycle-free, but not necessarily well-formed.

Example 11.1 The signature expression

```
sig structure A: sig end; structure B: sig end end
```

elaborates to the structure $S =$

$$\Big(\mathtt{m1}, \{\mathtt{A} \mapsto (\mathtt{m2}, \{\}), \mathtt{B} \mapsto (\mathtt{m2}, \{\})\}\Big)$$

but $\{\mathtt{m1}, \mathtt{m2}\}S$ is not barely principal for *sigexp* since it does not generalise $S' =$

$$\Big(\mathtt{m1}, \{\mathtt{A} \mapsto (\mathtt{m2}, \{\}), \mathtt{B} \mapsto (\mathtt{m3}, \{\})\}\Big)$$

which also is a possible result of the elaboration. However, $\{\mathtt{m1}, \mathtt{m2}, \mathtt{m3}\}S'$ *is* barely principal.

It easily follows from the definition that for given B and *sigexp* there can be at most one barely principal signature for *sigexp* in B.[32] Far more important is the following theorem, which states that if *any* signature can be found then a barely principal signature exists. This property, simple to state and perhaps not surprising, has been one of the main goals of the design and definition of ML Modules. The difficulty has been to ensure that none of the rich variety of ML's features has deprived it of the property, since it is crucial to both intuitive understanding and tractable implementation of the Modules.

We now state two theorems, both of which follow from the principality theorem, Theorem A.2 in Appendix A. The first theorem asserts that barely principal signatures exist, under a certain condition upon the basis. The second theorem asserts that the top-level basis will always satisfy this condition; it can then be shown that the condition will always be satisfied when principality is required. The theorems are deduced quite briefly as Corollaries A.6 and A.7 in Appendix A.

Theorem 11.1 (Barely Principal Signatures) *Let B be rigid and grounded, and let $B \vdash sigexp \Rightarrow S$ for some S. Then there exists a barely principal signature for sigexp in B.*

Theorem 11.2 (Bases) *Let $B \vdash topdec \Rightarrow B'$ occur in the elaboration of some program in the initial basis B_0. Then B is rigid and grounded.*

11.2 Defective signatures

All signatures we have seen in previous chapters have been barely principal, and they have always been well-formed and type-explicit. They have even been *principal* in the full sense; see Section 11.3 below. In general, however, several things can be wrong with a barely principal signature which should disqualify it from entering the elaboration tree. In particular, it may not be well-formed; it may not be type-explicit; it may not respect equality; it may not maximise equality; it may have all these virtues but still be an unsatisfiable monster. The following examples illustrate these various pathologies.

Example 11.2 A barely principal signature which is not well-formed.

[32] As pointed out at the end of Section 7.3, two signatures are considered to be identical if they can be made so by renaming their bound names and removing any unused bound names from their prefixes.

```
structure Empty = struct end
signature SIG  =
  sig
    structure E: sig type T end
    sharing E = Empty
  end
```

The barely principal signature for `SIG` is

$$\{\mathtt{m},\mathtt{t}\}\Big(\mathtt{m},\{E\mapsto(\mathtt{m0},\{\mathtt{T}\mapsto(\mathtt{t},\{\})\})\}\Big)$$

which is not well-formed because the bound typename `t` of a type structure occurs in a structure with the free name `m0`, the name of `Empty`.

Example 11.3 Another barely principal signature which is not well-formed.

Assume `Empty` is as above. Here the ill-formedness arises from a bound typename in the type scheme of a *value component* of a structure whose name is free:

```
sig
  type T
  structure E: sig val x: T  end
  sharing E = Empty
end
```

The barely principal signature is

$$\{\mathtt{m},\mathtt{t}\}\Big(\mathtt{m},\ \begin{array}{l}\{\mathtt{E}\mapsto(\mathtt{m0},\{\mathtt{x}\mapsto\mathtt{t}\})\},\\ \{\mathtt{T}\mapsto(\mathtt{t},\{\})\}\end{array}\Big)$$

Example 11.4 A barely principal signature which is not type-explicit.

```
sig type T  val x: T   type T end
```

The barely principal signature is

$$\{\mathtt{m},\mathtt{t1},\mathtt{t2}\}\Big(\mathtt{m},\ \begin{array}{l}\{\mathtt{T}\mapsto(\mathtt{t2},\{\})\},\\ \{\mathtt{x}\mapsto\mathtt{t1}\}\end{array}\Big)$$

which is not type-explicit, since the typename `t1` ascribed to `x` occurs in no type structure in the signature.

Example 11.5 A barely principal signature which does not respect equality.

```
sig
  datatype T = C of int -> int
  eqtype U
  sharing type T = U
end
```

The barely principal signature is

$$\{\mathtt{m},\mathtt{t}\}\Big(\mathtt{m},\ \{\ \mathtt{T}\mapsto(\mathtt{t},\{\mathtt{C}\mapsto(\mathtt{int}\to\mathtt{int})\to\mathtt{t}\}),\ \mathtt{U}\mapsto(\mathtt{t},\{\})\ \},\ \{\ \mathtt{C}\mapsto(\mathtt{int}\to\mathtt{int})\to\mathtt{t}\ \}\Big)$$

where `t` possesses the equality attribute; the type environment fails to respect equality [Sec 4.9, p 21] because the type `int -> int` does not admit equality. (It is the `eqtype` specification that forces `t` to admit equality.)

Example 11.6 A barely principal signature which does not maximise equality.

```
sig
  datatype T = C
end
```

The barely principal signature is

$$\{\mathtt{m},\mathtt{t}\}\Big(\mathtt{m},\ \{\ \mathtt{T}\mapsto(\mathtt{t},\{\mathtt{C}\mapsto\mathtt{t}\})\ \},\ \{\ \mathtt{C}\mapsto\mathtt{t}\ \}\Big)$$

where `t` does *not* possess the equality attribute; the type environment fails to maximise equality [Sec 4.9, p 21] because the signature would still respect equality if the typename `t` were awarded the equality attribute.

Example 11.7 A barely principal signature which is unmatchable.

```
structure C1 = struct end
structure C2 = struct end
signature SIG =
  sig
    structure A: sig structure B: sig end end
    sharing A = C1 and A.B = C2
  end
```

The barely principal signature for the large signature expression is

$$\{\mathtt{m0}\}\Big(\mathtt{m0},\ \{\mathtt{A}\mapsto(\mathtt{m1},\{\mathtt{B}\mapsto(\mathtt{m2},\{\})\})\}\ \Big)$$

where the free names `m1` and `m2` are those of the structures `C1` and `C2`. The signature has all the virtues lacked by the preceding examples; in particular it is well-formed (in contrast to Example 11.2). But it can never be matched, because the only real structure with name `m1` which will ever exist is the empty structure, and this fails to satisfy the signature's demand for a `B` component named `m2`.

We shall now embark upon remedying the defects in the above signatures in various ways. First, Examples 11.2, 11.3 and 11.4 are excluded by the simple device of requiring that all signatures in proofs be both well-formed and type-explicit; the first is a condition of admissibility (see Sections 9.2 and 9.4), while the second is a side-condition of rule 65 [p 39] (see Section 7.7). Second, the notion of *cover* is introduced in the next section to exclude such monsters as Example 11.7. Then in Section 11.4 we deal with equality in such a way that Example 11.5 will be rejected, while that of Example 11.6 will undergo slight transformation.

11.3 Covering and principality

We claimed in Example 11.7 that no structure will ever exist with name `m1` and with a `B` component. In fact the following theorem can be proved:

Theorem 11.3 *Let the sentence $B \vdash strexp \Rightarrow S$ occur in the elaboration of a program in the initial basis B_0. Then if (m, E) is a substructure of S such that $m \in N$ of B, and id is a structure identifier or type constructor such that E has an id component, then some structure (m, E') occurring in B also has an id component.*

In other words, every structure component or type component of a substructure of S with a rigid name stems from the basis. So clearly no structure will ever match `SIG` in Example 11.7.

A similar theorem does not hold for signature expressions, as indeed Example 11.7 illustrates. The concept of *covering* [Sec 5.13, p 35] is concerned with imposing the same property on sentences of the form $B \vdash sigexp \Rightarrow S$, to rule out some unmatchable signatures. *B covers S* means that if (m, E) is a substructure of S such that $m \in N$ of B, and *id* is a structure identifier or type constructor such that E has an *id* component, then some structure (m, E') occurring in B must also have an *id* component. (This structure may occur in any of the parts of B, for example in F of B or G of B.)

Before turning to principality, we prove a simple property of covering which we shall need.

Theorem 11.4 (Covering) *Let B cover S, and $\varphi S' = S$ where* Supp $\varphi \cap$ names $B = \emptyset$. *Then B covers S'.*

Proof Suppose that S' contains some (m, E') with $m \in N$ of B, and E' has an *id* component. Now $\varphi(m, E') = (m, \varphi E')$ since $\varphi m = m$, and $\varphi E'$ has an *id* component; but B covers S, so B must contain a structure (m, E) with an *id* component; hence B covers S'.

Now, to define principality, we simply add covering to bare principality. Precisely, $(N)S$ is *principal for sigexp in* B if it is barely principal and also B covers S – having chosen N so that $(N \text{ of } B) \cap N = \emptyset$. Now we justify the unqualified term "principal" as follows. Given B and S, among all those S such that $B \vdash \mathit{sigexp} \Rightarrow S$ we are only concerned with the ones covered by B, since only these can yield matchable signatures; and in fact the principal signature will possess every such S as instance. We can derive this now as a sharpened form of Theorem 11.1:

Theorem 11.5 (Principal Signatures) *Let B be rigid and grounded, and let $B \vdash \mathit{sigexp} \Rightarrow S$ for some S covered by B. Then there exists a principal signature for sigexp in B.*

Proof From Theorem 11.1 we have a unique signature $\Sigma = (N)S'$ barely principal for *sigexp* in B. Hence $\Sigma \geq S$, i.e. $\varphi S' = S$ with $\mathrm{Supp}\,\varphi \subseteq N$. But $(N \text{ of } B) \cap N = \emptyset$, so $\mathrm{Supp}\,\varphi \cap \mathrm{names}\,B = \emptyset$ since B is fixed. Hence B covers S' by Theorem 11.4, so Σ is principal.

Note that the principal signature, when it exists, is also the barely principal signature. Were one to replace the notion of principality in the Definition by bare principality, the sole result would be to admit certain unmatchable signatures; if a tree is a proof according to the present definition it would also be a proof according to the modified definition.

Cover was introduced not only to ban unmatchable signatures, but also in an attempt to ensure that a signature which is principal is automatically well-formed. It almost achieves this, but not quite. To see that the covering condition rejects certain kinds of ill-formedness, look again at Example 11.2. The signature there is not covered by the basis, since the only structure named `m0` in the basis is the empty structure and has no `T` component. But the ill-formed signature in Example 11.3 does *not* violate the covering condition, because the latter places no constraint upon type names occurring in the *VE* or *EE* components of a structure.

In fact the claim in the penultimate paragraph of [p 35], although it simply asserts that principal signatures exist (and Theorem 11.5 proves the claim!), is not fully accurate in the terms of the Definition, simply because the principal signature may not be well-formed (and in [Sec 5.5, p 33] it was assumed that all semantic objects mentioned thereafter would be admissible). We now see that it remains necessary, in applying rule 65 [p 39], to check that the signature is well-formed.

Perhaps the definition of well-formedness is unnecessarily strong. Taking the hint from consistency and cover – neither of which places any constraint on the *VE* or *EE* components of a structure – we might try replacing the definition in [Sec 5.3, p 32] with the following weaker definition: $(N)S$ is *well-formed* if $N \subseteq \text{names}\, S$ and also, whenever (m, E) is a substructure of S and $m \notin N$, then for every structure (m', E') in E we have $m' \notin N$ and for every type structure (θ, CE) in E we have $\text{tynames}\, \theta \cap N = \emptyset$. In this case it is not hard to see that if Σ is principal for *sigexp* in B then (due to consistency and cover together) Σ is well-formed.

However, we do not fully understand the consequences of weakening well-formedness in this way.

There are also stronger forms of covering which could be imposed (although they would still not ensure well-formedness of signatures). For example, one could demand the covering condition for *every* component *id*, not just for structure identifiers and type constructors. Even more strongly, one could adopt the following definition: *B strongly covers S* means that if (m, E) is a substructure of S such that $m \in N$ of B, then there is a structure (m, E') in B such that $\text{Dom}\, E' \supseteq \text{Dom}\, E$. It is likely that the signatures excluded by these stronger form of covering are also unmatchable.

11.4 Equality-principal signatures

Even if the principal signature is well-formed and type-explicit, it may still be defective in its treatment of the equality attribute, as illustrated by Examples 11.5 and 11.6. If (as in Example 11.5) it fails to respect equality then it makes a contradictory claim that some datatype admits equality.[33]

Signatures like the one in Example 11.6, however,

$$\{\mathtt{m}, \mathtt{t}\}\Big(\mathtt{m},\ \{\ \mathtt{T} \mapsto (\mathtt{t}, \{\mathtt{C} \mapsto \mathtt{t}\})\ \},\ \{\ \mathtt{C} \mapsto \mathtt{t}\ \}\Big)$$

(where `t` does not admit equality) are uninformative rather than contradictory; they fail to express the fact that a certain datatype in *every* matching structure will indeed admit equality. Let us see why `t` does not admit equality in the above (principal) signature. Recall that a type realisation can map a non-equality type name to an equality type name, but not the other way around. Therefore, since we are free to choose a non-equality type name for `T` during the elaboration of the signature expression, the principal signature must use a non-equality type name. This is no problem if the signature is simply used to constrain an existing structure, for the signature constraint will not affect the fact that the real datatype `T` admits equality. However, if the signature is used as a specification of an unknown

[33] Example 11.5 will be rejected, because rule 65 of the Definition will be inapplicable. In general, the equality-principal signature as defined in [Sec 5.13, p 36] may exist even if the principal signature does not respect equality; but this is not the case for Example 11.5.

structure, it would be intolerable if `T` were not to admit equality, given that any actual datatype which matches the specification will admit equality. The critical case is when the signature represents the formal parameter of a functor, for then it would prevent valid uses of the equality predicate within the body of the functor. This is a case where, if we retain a signature which is a relative monster (in the sense of Section 9.1, meaning that there exists a more specific signature which is matched by exactly the same real structures), then we exclude useful programs.

We therefore define what it means for a signature $\Sigma = (N)S$ to maximise equality, by analogy with the concept for type environments [Sec 4.9, p 21]. First, let T be the set of type names $t \in N$ such that (t, CE) occurs in S for some $CE \neq \{\}$; we call these the *bound datatype names* of Σ. Then we say that Σ *maximises equality* if (a) Σ respects equality, and also (b) if any larger subset of T were to admit equality (without any change in the equality attribute of any type names not in T) then Σ would cease to respect equality.[34]

Now let $\Sigma_0 = (N_0)S_0$ be principal for *sigexp* in B, and let T_0 be its bound datatype names. Then Σ is *equality-principal for sigexp in B* if it maximises equality, and is obtained from Σ_0 merely by possibly making more members of T_0 admit equality. (This is equivalent to the definition given in [Sec 5.13, p 36].)

If Σ_0 respects equality then such a Σ certainly exists; it may exist in any case. Also, Σ is uniquely determined by Σ_0, when Σ exists. (With modifications, one can reuse the argument from Section 5.2 where we proved that, for any type environment *TE*, the side-condition "*TE* maximises equality" uniquely determines the equality attributes of every type structure in *TE*.)

Now assume that Σ_0 is well-formed and type-explicit. Then Σ, if it exists, is also well-formed and type-explicit. Also, since $B \vdash sigexp \Rightarrow S_0$ and $N_0 \cap \text{names}\, B = \emptyset$ we get $B \vdash sigexp \Rightarrow S$, by Theorem 10.1, the realisation theorem.[35] Thus if Σ exists all the premises of rule 65,

$$\frac{\begin{array}{ll} B \vdash sigexp \Rightarrow S & (N)S \text{ equality-principal for } sigexp \text{ in } B \\ & (N)S \text{ type-explicit} \end{array}}{B \vdash sigexp \Rightarrow (N)S}$$

are met. (There is no need for the rule to require well-formedness of $(N)S$ explicitly, since it is required implicitly by the fact that $(N)S$ occurs in the rule.)

[34] This definition is almost word-for-word the same as for type environments. But it is not quite true that Σ maximises equality iff all type environments in Σ maximise equality. The reason is to do with possible sharing between two datatypes; see Exercise 11.8. The exercise also explains why maximising equality is separated from principality.

[35] Strictly speaking, the terminology "to change the equality attribute of a type name" is a bit loose, since every type name from the outset has an unchangeable equality attribute. What is meant is, of course, that one can pick fresh type names all of which do admit equality and are disjoint from the names in the proof of $B \vdash sigexp \Rightarrow S_0$, and this amounts to creating a bijective type realisation φ from the type names one wishes to change to the fresh type names; such a φ cannot destroy the admissibility of a proof, so one can indeed apply Theorem 10.1.

Conversely, if Σ_0 is not well-formed or not type-explicit then rule 65 cannot be applied, for either Σ does not exist, or else it exists but is not well-formed or not type-explicit. Therefore it is valid to check that a principal signature has these properties, before attempting to transform it into an equality-principal signature.

Thus, rule 65 can be implemented by the following steps:

Step 1 Find a barely principal signature $\Sigma_0 = (N_0)S_0$ for *sigexp* in B; FAIL, if none exists

Step 2 Check that B covers S_0, so that Σ_0 is principal; otherwise FAIL

Step 3 Check that Σ_0 is well-formed and type-explicit; otherwise FAIL

Step 4 Compute and return the equality-principal signature Σ; FAIL, if none exists

During step 1, one might have to choose type names that admit equality in order to satisfy sharing constraints (with `eqtype`s or external types). In step 4 as many further bound datatype names as possible are made to admit equality; failure only occurs when it is impossible to make Σ respect equality. (This eliminates examples like 11.5 above.) During step 1, one should not attempt to determine the final equality attribute while processing a `datatype` specification in isolation, for it can depend on specifications elsewhere in the signature. By contrast, when elaborating `datatype` or `abstype` *declarations*, the equality attributes can be determined separately for each declaration.

The reader may wonder why type explication is not part of admissibility, so that no explicit check for type explication would be needed in rule 65. There is no strong reason; the definition could probably be reformulated to include type explication in admissibility. As it stands, however, type explication must not be imposed everywhere; in particular, in a functor signature $(N)(S, (N')S')$, the signature $(N')S'$ need not be type-explicit. For example, in

```
functor F(S: sig end) =
  local datatype T = C in val x = C end
```

$(N')S'$ is $\{\mathtt{t}\}(\{\mathtt{x} \mapsto \mathtt{t}\})$, which is not type-explicit.

Exercise 11.1 Does rule 65 apply successfully to the signature expression below? If so, what is the result of the elaboration?

```
sig datatype T = C; type U sharing type T = U end
```

Exercise 11.2 Same question, for

```
sig type T; datatype U = C end
```

Exercise 11.3 Same question, for

```
sig type T; datatype U = C of T end
```

Exercise 11.4 Same question, for

```
sig eqtype T; datatype U = C of T end
```

Exercise 11.5 Same question, for

```
sig
  datatype T = C
  datatype U = C of int -> int
  sharing type T = U
end
```

Exercise 11.6 Same question, for

```
sig
  eqtype T
  datatype U = C of int -> int
  sharing type T = U
end
```

The next exercise is concerned with introducing type abbreviations in signatures, an idea which was discussed in the small print at the end of Section 6.3.

Exercise 11.7 (For specialists) In Section 6.3 we suggested a syntactic condition under which type abbreviations in signatures might be introduced, namely that a non-atomic type expression be permitted in a sharing equation only on condition that it contains no flexible type constructors. This side-condition is intended to avoid problems with second-order unification, see Section 10.4. Instead one might consider the following natural inference rule, to allow a type abbreviation in a specification, imposing a *semantic* rather than a *syntactic* side-condition:

$$\frac{B \vdash ty \Rightarrow \tau \quad (\Lambda\alpha^{(k)}.\tau, \{\}) \text{ grounded in } N \text{ of } B \quad tyvarseq = \alpha^{(k)}}{B \vdash \texttt{type}\ \ tyvarseq\ tycon\ \texttt{=}\ ty \Rightarrow \{tycon \Rightarrow (\Lambda\alpha^{(k)}.\tau, \{\})\}}$$

This rule would be much more permissive than the syntactic condition. Prove that if one introduces the above rule, the principality theorem (Theorem 11.1) no longer holds!

Exercise 11.8 (For specialists) The reader might expect that, if we were able to confine our attention to signatures which maximise equality, then among them we could find one which possesses all the others as instances, and which

could therefore be called *equality-principal*. In fact this is false. To see why, consider all the signatures to which the following signature expression could elaborate and which maximise equality. (For this purpose, relax rule 65 by removing the principality requirement.) Show that none of them is principal in the new sense.

```
signature datatype T = A  datatype U = A of int->int end
```

A Appendix: Proof of Principality

The purpose of this appendix is to prove the central theorem in the static semantics of Modules, implying the existence of principal signatures (referred to as the *principality theorem*, for short). It states that, roughly speaking, if a signature *sigexp* elaborates at all in some basis B, then it elaborates to a signature Σ in B such that the structures to which *sigexp* elaborates in B are precisely the structures that are instances of Σ.

This property is crucial for several reasons. First, it is the reason why we never have to elaborate any signature expression more than once. If, for example, we declare `SIG` by

`signature SIG =` *sigexp*

then *sigexp* can be elaborated to a principal signature Σ, if it can be elaborated at all; whenever we subsequently want to find out whether some structure S matches `SIG`, it suffices to check whether S matches Σ in the sense of [Sec 5.12, p 35].

Second, the principality theorem is the reason why the body of a functor can be elaborated once when the functor is declared, and never has to be re-elaborated when the functor is applied. Recall that in the rule for functor bindings [rule 99, p 44], the signature $(N)S$ obtained from the functor parameter signature expression *sigexp* is principal[36] and hence unique up to the choice of the names in N. Thus, the structure S which we bind to *strid*, the functor parameter, before elaborating the functor body, is uniquely determined by B and *sigexp*. Were there distinct (maximal) signatures, which structure would one bind to *strid*? Also, S has precisely as much sharing as any structure which satisfies *sigexp* must have. It can be proved that as a consequence, for any actual structure S' which matches $(N)S$, the functor body would elaborate, were one to bind S', rather than S, to *strid*.

Thus principality is not merely a property of theoretical interest but also one which very much affects how the language can be implemented. Indeed, our proof of the principality theorem demonstrates the existence of principal signatures by actually constructing them. This construction will hopefully be of use to implementers, not least because the proof indicates that one can write a single function which will handle 18 out of the 26 inference rules involved.

In Section A.1 we shall define *structural contractions* which are used in the principality proof. Then, in Section A.2 we state and prove the principality theorem. Finally, in Section A.3 we deduce Theorem 11.1 about the existence of barely principal signatures, from which the existence of principal signatures, Theorem 11.5, is deduced in Chapter 11.

[36] Actually it is equality-principal, but that is not important for the present discussion.

A.1 Structural contractions

Let $\mathcal{E}$ be a map over semantics objects, with domain $\mathrm{Dom}\,\mathcal{E}$. (A typical example is $\mathcal{E} = + : \mathrm{Env} \times \mathrm{Env} \to \mathrm{Env}$; then $\mathrm{Dom}\,\mathcal{E} = \mathrm{Env} \times \mathrm{Env}$.)

We say that $\mathcal{E}$ is *structural* if, for all realisations φ and objects $P \in \mathrm{Dom}\,\mathcal{E}$,

$$\varphi(\mathcal{E}(P)) = \mathcal{E}(\varphi(P))$$

For any admissible assembly A and structure (m, E), we can say an occurrence of (m, E) in A is *free* if m is free. Moreover, we say that an occurrence of a type structure (θ, CE) is *free* in A if every type name in θ is free. Note that if A is grounded in N, and (θ, CE) occurs in A but not within a functor signature, then either this is a free occurrence and $\mathrm{tynames}(\theta) \subseteq N$, or else θ is simply a type name – free or bound.

We now say that $\mathcal{E}$ is a *contraction* if, for all $P \in \mathrm{Dom}\,\mathcal{E}$, if P is admissible and grounded in N then

1. $(P, \mathcal{E}(P))$ is admissible and grounded in N
2. For every structure (m, E') occurring free in $\mathcal{E}(P)$, there exists an E such that (m, E) occurs free in P and $\mathrm{Dom}\,SE\mathrm{of}\,E' \subseteq \mathrm{Dom}\,SE\mathrm{of}\,E$ and $\mathrm{Dom}\,TE\mathrm{of}\,E' \subseteq \mathrm{Dom}\,TE$ of E
3. For every type structure (θ, CE') which occurs free in $\mathcal{E}(P)$, there exists a CE such that (θ, CE) occurs free in P

Intuitively, (2) and (3) just express that, as far as consistency is concerned [Sec 5.2, p 32], $\mathcal{E}$ produces nothing which does not stem from the argument.

The reason that it is most useful to know about some $\mathcal{E}$ that it is a contraction is that one then knows that it preserves admissibility, not just with respect to the argument P but also with respect to any assembly:

Lemma A.1 *Let $\mathcal{E}$ be a contraction, N a name set and A an assembly. Then for all $P \in \mathrm{Dom}\,\mathcal{E}$, if (P, A) is admissible and grounded in N then $(\mathcal{E}(P), P, A)$ is admissible and grounded in N.*

Proof Immediate from the definition of admissibility [Sec 5.5, p 33].

The point of the above definitions is that many inference the rules are (ignoring side-conditions) of the form

$$\frac{\mathcal{E}_1(P) \vdash \mathit{phrase}_1 \Rightarrow Q_1 \quad \cdots \quad \mathcal{E}_k(P, Q_1, \ldots, Q_{k-1}) \vdash \mathit{phrase}_k \Rightarrow Q_k}{P \vdash \mathit{phrase} \Rightarrow \mathcal{E}(P, Q_1, \ldots, Q_k)}$$

for $k \geq 0$, where $\mathcal{E}_1, \ldots, \mathcal{E}_k$ and $\mathcal{E}$ are structural contractions and every premise is inferred by one of the rules listed above. For brevity, we shall refer to such a rule

as a *structural contraction*, provided also that its side-conditions depend only on *phrase*. (Rule 86 provides an example of such a side-condition.)

Some rules contain side-conditions purely for notational convenience and are equivalent to structural contractions without side-conditions. This applies to rules 47, 49, 74 and 78.

We now assert that the eighteen rules 47–52, 70, 74–78, 80–82, 86, 87 and 90 are all structural contractions. (In rule 49, use that realisation commutes with type function application; in rule 70, use that realisation commutes with closure.)

A.2 The principality theorem

The principality theorem is only concerned with proofs ∇ that prove sentences of the form $B \vdash \mathit{sigexp} \Rightarrow S$, i.e. proofs from $\mathcal{R}_{\mathtt{sig}}$. For such proofs, recalling Section 9.4, admissibility of ∇ means that the assembly of all semantic objects occurring in ∇ is admissible. Similarly, if ∇ is a proof from $\mathcal{R}_{\mathtt{sig}}$ and A is an assembly, we say that the pair (∇, A) is *admissible* if the assembly of all objects occurring in ∇ or in A is admissible.

The principality theorem is

Theorem A.2 (Principal Elaborations) *Let phrase be one of sigexp, spec, ty, tyrow, valdesc, exdesc, strdesc or shareq. Further, let P and Q be semantic objects, A an assembly, N a name set, φ a realisation fixed on N, and ∇ a proof such that*

(P, A) *is admissible and grounded in* N
$(\nabla, \varphi A)$ *is admissible*
∇ *proves* $\varphi P \vdash \mathit{phrase} \Rightarrow Q$ *from* $\mathcal{R}_{\mathtt{sig}}$

Then there exist φ^*, ∇^*, Q^* depending only on P, A, N and phrase, such that φ^* is fixed on N and

$(\nabla^*, \varphi^* A)$ *is admissible and grounded in* N
∇^* *proves* $\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*$

Moreover, for some ψ fixed on N,

$$\psi(\nabla^*, \varphi^* A) = (\nabla, \varphi A)$$

A few informal comments may be in order. Let us take sentences of the form $B \vdash \mathit{sigexp} \Rightarrow S$ as an example. One of the assumptions is that for some ∇ and φ, ∇ proves $\varphi B \vdash \mathit{sigexp} \Rightarrow S$. One of the conclusions is that there exist φ^*, ∇^* and S^* such that ∇^* proves $\varphi^* B \vdash \mathit{sigexp} \Rightarrow S^*$. Here φ^*, ∇^* and S^* depend only on B, N, A and *sigexp*; one can think of φ^*, ∇^* and S^* as being the results of running a "modules elaborator" (or "signature checker") on the input B, N, A and *sigexp*.

It may be the case that *sigexp* does not elaborate to any S in B, but that it elaborates to some S in φB, for some realisation φ. This is the case if *sigexp* fails to elaborate because some sharing constraint in *sigexp* is not met in B, but

is met in φB because φ has identified the offending names or type functions. The theorem therefore answers a question not just about the elaborations of *sigexp* in B, but about the elaborations of *sigexp* in φB, for any φ. The reader may now object that this is not the question we want answered, since we do not want the elaboration of signature expressions to identify different rigid names, i.e. names of "real" structures and types. But this is the point of the parameter N of the theorem; by taking $N = \mathrm{names}\, B$ we ensure that $\varphi B = B$. This is how we specialise the theorem in Section A.3.

Another conclusion of the theorem is that φ^*, ∇^* and S^* are *most general* (or *universal*) in the sense that for any φ, ∇ and S satisfying that ∇ proves $\varphi B \vdash \mathit{sigexp} \Rightarrow S$, there exists a realisation ψ such that $\psi(\nabla^*) = \nabla$; in particular, $\psi(\varphi^* B) = \varphi(B)$ and $\psi S^* = S$.

It turns out that the only phrases that contribute to the construction of φ^* are sharing equations. In particular, if *sigexp* contains no sharing equations, φ^* will be Id.

As we have seen earlier, admissibility is not a transitive property. This is why the theorem is concerned with the admissibility not of a single object or proof but with the admissibility of that object or proof together with a whole assembly. One can think of A at the beginning of the elaboration as just being B; as elaboration progresses, A is "knocked down" by the realisations that result from the elaboration of sharing equations; moreover, structures and type structures that are chosen to have fresh names during the elaboration are accumulated in A.

Finally, the requirements that (P, A) be grounded in N and that φ be fixed on N are needed to avoid dealing with second-order unification of type functions, as explained in Section 10.4. This concludes our comments on the statement of the theorem.

In the proof of the principality theorem we shall use the following lemmas, which are easily proved by induction on the structure of *ty*. First, realisation and inverse realisation preserve the elaboration of type expressions:

Lemma A.3 *Let C and φC be admissible. Then*

(1) If ∇ proves $C \vdash ty \Rightarrow \tau$ then $\varphi\nabla$ proves $\varphi C \vdash ty \Rightarrow \varphi\tau$

(2) If ∇' proves $\varphi C \vdash ty \Rightarrow \tau'$, then there exist ∇ and τ such that $\nabla' = \varphi(\nabla)$ and ∇ proves $C \vdash ty \Rightarrow \tau$.

Second, the elaboration of type expressions is uniquely determined by the context:

Lemma A.4 *For every C and ty, there exists at most one (∇, τ) such that ∇ proves $C \vdash ty \Rightarrow \tau$. Moreover, whenever ∇ proves $C \vdash ty \Rightarrow \tau$ and C' is an admissible context such that*

(1) $\mathrm{Dom}(TE) = \mathrm{Dom}(TE')$

(2) θ of $TE(tycon) = \theta$ of $TE'(tycon)$, *for all* $tycon \in \mathrm{Dom}\, TE$,

where $TE = TE$ of C *and* $TE' = TE$ of C', *then there exists a* ∇' *which proves* $C' \vdash ty \Rightarrow \tau$.

Third, in elaborating a type expression in C, no semantic object outside C is encoutered:

Lemma A.5 *For every* C, ty, τ *and* ∇, *if* ∇ *proves* $C \vdash ty \Rightarrow \tau$ *then for all assemblies* A *and name sets* N, (∇, A) *is admissible and grounded in* N *if and only if* (C, A) *is admissible and grounded in* N.

We now proceed to the proof of the principality theorem.

Proof We prove the theorem by induction on the depth of ∇. The proof ∇ concludes with one of the rules 47–52, 63, 64, 70–81, 82, 86, 87, 88–90. Thus our inductive proof potentially has twenty-six cases to consider. However, we have seen that eighteen of these are structural contractions, and this dramatically shortens the proof. A further rule, namely rule 79 for `include`, may also be ignored; for one can show that the form `include` *sigid* elaborates to exactly the same results as

`local structure S:` *sigid* `in open S end`

in any basis, and this translation generalises easily to the case of several signature identifiers.

We now assume the hypotheses of the theorem, and proceed by case analysis of the final rule used in the proof ∇. Note in particular that φ is assumed to be grounded in N. In each case it is easy to demonstrate that the φ^* and ψ constructed are also grounded in N, and we shall not usually mention this explicitly.

First we treat the general case of a structural contraction rule; then we treat the remaining seven cases individually.

Structural Contractions Without loss of generality it will be sufficient to deal with the structural contraction rules just in the case $k = 2$. Therefore, consider the case in which ∇ concludes with an instance

$$\frac{\mathcal{E}_1(\varphi P) \vdash phrase_1 \Rightarrow Q_1 \qquad \mathcal{E}_2(\varphi P, Q_1) \vdash phrase_2 \Rightarrow Q_2}{\varphi P \vdash phrase \Rightarrow \mathcal{E}(\varphi P, Q_1, Q_2)} \tag{1}$$

where $\mathcal{E}_1, \mathcal{E}_2$ and $\mathcal{E}$ are structural contractions, φ is fixed on N, and $Q = \mathcal{E}(\varphi P, Q_1, Q_2)$.

Then by assumption

$$(P, A) \text{ is admissible and grounded in } N \tag{2}$$
$$(\nabla, \varphi A) \text{ is admissible} \tag{3}$$
$$\nabla \text{ proves } \varphi P \vdash \mathit{phrase} \Rightarrow Q \tag{4}$$

By (4) there exist ∇_1 and ∇_2 such that

$$\nabla = \frac{\nabla_1 \qquad \nabla_2}{\varphi P \vdash \mathit{phrase} \Rightarrow Q} \tag{5}$$

and

$$\nabla_1 \text{ proves } \mathcal{E}_1(\varphi P) \vdash \mathit{phrase}_1 \Rightarrow Q_1 \tag{6}$$
$$\nabla_2 \text{ proves } \mathcal{E}_2(\varphi P, Q_1) \vdash \mathit{phrase}_2 \Rightarrow Q_2 \tag{7}$$

Using induction, first time: We now wish to use induction on ∇_1. Let $P_1 = \mathcal{E}_1(P)$, $\varphi_1 = \varphi$ and $A_1 = (P, A)$. Since $\mathcal{E}_1$ is a contraction and (P, A) is admissible and grounded in N we get

$$(P_1, A_1) \text{ is admissible and grounded in } N$$

by Lemma A.1. Also, $(\nabla_1, \varphi_1 A_1) = (\nabla_1, (\varphi_1 P, \varphi_1 A))$ so by (3),

$$(\nabla_1, \varphi_1 A_1) \text{ is admissible}$$

Also, from (6) and the fact that $\mathcal{E}_1$ is structural, we get

$$\nabla_1 \text{ proves } \varphi_1 P_1 \vdash \mathit{phrase}_1 \Rightarrow Q_1$$

Thus, by induction there exist φ_1^*, ∇_1^* and Q_1^*, depending only on P_1, A_1, N and phrase_1, such that

$$(\nabla_1^*, \varphi_1^* A_1) \text{ is admissible and grounded in } N \tag{8}$$
$$\nabla_1^* \text{ proves } \varphi_1^* P_1 \vdash \mathit{phrase}_1 \Rightarrow Q_1^* \tag{9}$$

Moreover, for some φ_2 fixed on N

$$\varphi_2(\nabla_1^*, \varphi_1^* A_1) = (\nabla_1, \varphi_1 A_1) \tag{10}$$

Using induction, second time: We now wish to apply induction to ∇_2. Let $P_2 = \mathcal{E}_2(\varphi_1^* P, Q_1^*)$ and let $A_2 = (\nabla_1^*, \varphi_1^* A_1)$. Then

$$A_2 = (\nabla_1^*, \varphi_1^*(P, A)) = (\nabla_1^*, (\varphi_1^* P, \varphi_1^* A)) \tag{11}$$

so $((\varphi_1^* P, Q_1^*), A_2)$ is admissible and grounded in N by (8), noting that Q_1^* occurs in ∇_1^*. Since $\mathcal{E}_2$ is a contraction, we therefore have

$$(P_2, A_2) \text{ is admissible and grounded in } N$$

using Lemma A.1. Moreover

$$\begin{array}{lcll}(\nabla_2, \varphi_2 A_2) & = & (\nabla_2, (\nabla_1, \varphi_1 A_1)) & \text{by (10)} \\ & = & (\nabla_2, (\nabla_1, (\varphi P, \varphi A))) & \end{array}$$

so by (3) and (5),

$(\nabla_2, \varphi_2 A_2)$ is admissible

Also,

$$\begin{array}{lcll}\mathcal{E}_2(\varphi P, Q_1) & = & \mathcal{E}_2(\varphi_2 \varphi_1^* P, \varphi_2 Q_1^*) & \text{by (10)} \\ & = & \varphi_2(\mathcal{E}_2(\varphi_1^* P, Q_1^*)) & \text{as } \mathcal{E}_2 \text{ is structural} \\ & = & \varphi_2 P_2 & \end{array}$$

so by (7),

∇_2 proves $\varphi_2 P_2 \vdash \mathit{phrase}_2 \Rightarrow Q_2$

Thus, by induction there exist φ_2^*, ∇_2^* and Q_2^*, depending only on P_2, A_2, N and phrase_2, such that

$$(\nabla_2^*, \varphi_2^* A_2) \text{ is admissible and grounded in } N \tag{12}$$
$$\nabla_2^* \text{ proves } \varphi_2^* P_2 \vdash \mathit{phrase}_2 \Rightarrow Q_2^* \tag{13}$$

Moreover, for some ψ,

$$\psi(\nabla_2^*, \varphi_2^* A_2) = (\nabla_2, \varphi_2 A_2) \tag{14}$$

Collecting the results: We shall now prove that if we define $\varphi^* = \varphi_2^* \circ \varphi_1^*$, $Q^* = \mathcal{E}(\varphi^* P, \varphi_2^* Q_1^*, Q_2^*)$ and

$$\nabla^* = \frac{\varphi_2^* \nabla_1^* \qquad \nabla_2^*}{\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*}$$

then φ^*, ∇^* and Q^* have the desired properties.

Note that φ^*, ∇^* and Q^* only depend on P, A, N and *phrase*. Moreover, written out in full, the assembly from (12) – let us call it A_3 – is

$$\begin{array}{lcll} A_3 & = & (\nabla_2^*, \varphi_2^* A_2) & \\ & = & (\nabla_2^*, (\varphi_2^* \nabla_1^*, (\varphi_2^* \varphi_1^* P, \varphi_2^* \varphi_1^* A))) & \text{by (11)} \end{array}$$

i.e.

$$A_3 = (\nabla_2^*, (\varphi_2^* \nabla_1^*, (\varphi^* P, \varphi^* A))) \tag{15}$$

Hence $((\varphi^* P, \varphi_2^* Q_1^*, Q_2^*), A_3)$ is admissible and grounded in N. Thus, by Lemma A.1, $(\mathcal{E}(\varphi^* P, \varphi_2^* Q_1^*, Q_2^*), A_3)$, i.e. (Q^*, A_3), is admissible and grounded in N. But then, by (12), (15) and the definition of ∇^*,

$(\nabla^*, \varphi^* A)$ is admissible and grounded in N

as required.

We now have to check that ∇^* really proves its conclusion, $\varphi^* P \vdash \mathit{phrase} \Rightarrow Q^*$. We have already established that ∇^* is admissible. Also, the side-conditions on the last step of ∇^*, if present, are satisfied because they depend on *phrase* only and were satisfied at (4). But is the conclusion of ∇^* obtained by an instance of the same rule as yielded (1)?

To see that it is, first note that by (9), ∇_1^* proves $\varphi_1^* P_1 \vdash \mathit{phrase}_1 \Rightarrow Q_1^*$ and since we know that the tree $\varphi_2^* \nabla_1^*$ really is admissible, we get that

$$\varphi_2^* \nabla_1^* \text{ proves } \varphi^* P_1 \vdash \mathit{phrase}_1 \Rightarrow \varphi_2^* Q_1^*$$

by Theorem 10.1. Since $\mathcal{E}_1$ is structural and $P_1 = \mathcal{E}_1(P)$, this is equivalent to

$$\varphi_2^* \nabla_1^* \text{ proves } \mathcal{E}_1(\varphi^* P) \vdash \mathit{phrase}_1 \Rightarrow \varphi_2^* Q_1^* \tag{16}$$

Similarly, because $\mathcal{E}_2$ is structural and $P_2 = \mathcal{E}_2(\varphi_1^* P, Q_1^*)$, (13) is equivalent to

$$\nabla_2^* \text{ proves } \mathcal{E}_2(\varphi^* P, \varphi_2^* Q_1^*) \vdash \mathit{phrase}_2 \Rightarrow Q_2^* \tag{17}$$

But from (16) and (17) we see that ∇^* really does conclude with an instance of the structural contraction rule under consideration, so ∇^* is a proof, as required.

It remains to exhibit a ψ with the desired properties. Take any ψ satisfying (14). Let us expand the left- and right-hand sides of the equation in (14):

$$\begin{array}{lcll} \psi(A_3) & = & (\psi\nabla_2^*, (\psi\varphi_2^*\nabla_1^*, (\psi\varphi^* P, \psi\varphi^* A))) & \text{by (15)} \end{array}$$

and

$$\begin{array}{lcll} (\nabla_2, \varphi_2 A_2) & = & (\nabla_2, (\nabla_1, \varphi_1 A_1)) & \text{by (10)} \\ & = & (\nabla_2, (\nabla_1, (\varphi P, \varphi A))) & \end{array}$$

so (15) gives $\psi\nabla_2^* = \nabla_2$, $\psi\varphi_2^*\nabla_1^* = \nabla_1$, $\psi\varphi^* P = \varphi P$ and $\psi\varphi^* A = \varphi A$. But then

$$\begin{array}{lcl} \psi Q^* & = & \psi(\mathcal{E}(\varphi^* P, \varphi_2^* Q_1^*, Q_2^*)) \\ & = & \mathcal{E}(\psi\varphi^* P, \psi\varphi_2^* Q_1^*, \psi Q_2^*) \\ & = & \mathcal{E}(\varphi P, Q_1, Q_2) \\ & = & Q \end{array}$$

so that

$$\begin{array}{lcl} \psi(\nabla^*, \varphi^* A) & = & \Big(\dfrac{\psi\varphi_2^*\nabla_1^* \qquad \psi\nabla_2^*}{\psi\varphi^* P \vdash \mathit{phrase} \Rightarrow \psi Q^*}\,, \psi\varphi^* A\Big) \\ & = & \Big(\dfrac{\nabla_1 \qquad \nabla_2}{\varphi P \vdash \mathit{phrase} \Rightarrow \psi Q^*}\,, \varphi A\Big) \\ & = & (\nabla, \varphi A) \end{array}$$

as required.

Rule 63, *sigexp* ≡ `sig` *spec* `end` This rule is not a structural contraction, due to the structure name m in the conclusion. By assumption (with $P = B$, $Q = S$)

$$(B, A) \text{ is admissible and grounded in } N \tag{1}$$
$$(\nabla, \varphi A) \text{ is admissible} \tag{2}$$
$$\nabla \text{ proves } \varphi B \vdash \mathit{sigexp} \Rightarrow S \tag{3}$$

Then there exist ∇_1, m and E such that

$$\nabla = \frac{\nabla_1}{\varphi B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)} \tag{4}$$
$$(\nabla_1, \varphi A) \text{ is admissible} \tag{5}$$
$$\nabla_1 \text{ proves } \varphi B \vdash \mathit{spec} \Rightarrow E \tag{6}$$

By induction on (1), (5) and (6) there exist φ_1^*, ∇_1^* and E^* depending only on B, A, N and *spec*, such that

$$(\nabla_1^*, \varphi_1^* A) \text{ is admissible and grounded in } N \tag{7}$$
$$\nabla_1^* \text{ proves } \varphi_1^* B \vdash \mathit{spec} \Rightarrow E^* \tag{8}$$

Moreover, for some ψ_1,

$$\psi_1(\nabla_1^*, \varphi_1^* A) = (\nabla_1, \varphi A) \tag{9}$$

Let m^* be a structure name such that

$$m^* \notin \text{names}(\nabla_1^*, \varphi_1^* A) \tag{10}$$

Let $\varphi^* = \varphi_1^*$, $S^* = (m^*, E^*)$ and let

$$\nabla^* = \frac{\nabla_1^*}{\varphi^* B \vdash \mathit{sigexp} \Rightarrow S^*} \tag{11}$$

These depend only on B, A, N and *sigexp*. Now $(\nabla^*, \varphi^* A)$ is admissible and grounded in N by (7) and (10). Since also ∇^* is an instance of rule 63, ∇^* proves $\varphi^* B \vdash \mathit{sigexp} \Rightarrow S^*$. Let $\psi = \psi_1 + \{m^* \mapsto m\}$. Then

$$\begin{aligned}
\psi(\nabla^*, \varphi^* A) &= \Big(\frac{\psi \nabla_1^*}{\psi\varphi^* B \vdash \mathit{sigexp} \Rightarrow \psi S^*}\ ,\ \psi\varphi^* A\Big) \\
&= \Big(\frac{\psi_1 \nabla_1^*}{\psi_1\varphi_1^* B \vdash \mathit{sigexp} \Rightarrow (m, \psi_1 E^*)}\ ,\ \psi_1\varphi_1^* A\Big) \text{by (10)} \\
&= \Big(\frac{\nabla_1}{\varphi B \vdash \mathit{sigexp} \Rightarrow (m, E)}\ ,\ \varphi A\Big) \qquad \text{by (9)} \\
&= (\nabla, \varphi A)
\end{aligned}$$

as required.

Rule 64, *sigexp* ≡ *sigid* By assumption (with $P = B, Q = S$) conditions (1)–(3) hold as in rule 63, but in this case *sigexp* ≡ *sigid* and ∇^* consists of a single step with $\varphi(B(\mathit{sigid})) \geq S$. Write $B(\mathit{sigid})$ in the form $(N^*)S^*$, where N^* is disjoint from names$(B, A) \cup N$. Let $\varphi^* = \mathrm{Id}$, and let

$$\nabla^* = \frac{}{B \vdash \mathit{sigid} \Rightarrow S^*}$$

These depend only on B, A, N and *sigid*, as required. Since (B, A) by assumption is admissible and grounded in N and S^* is drawn from B, $(\nabla^*, \varphi^* A)$ – i.e. (∇^*, A) – is admissible and grounded in N as required. Also $B(\mathit{sigid}) \geq S^*$, so ∇^* really proves it conclusion, as required.

To exhibit ψ, we note that $\varphi' S^* = S$ for some φ' such that $\mathrm{Supp}\,\varphi' \subseteq N^*$, and it is enough to take $\psi = \varphi + \varphi'$.

Rule 71, *spec* ≡ `type` *typdesc* Rule 71 is a structural contraction, but notice that its premise involves a *typdesc*, so is not inferred by one of the rules we consider in the inductive proof. We wished to avoid proving the theorem for the non-contractive rule 83 for type descriptions – not that this is difficult (the case is very similar to that of rule 63) but simply because we wish to keep the size of the proof to a minimum. The case for rule 71 is easier to deal with by adapting the treatment of rule 72 below, simply by omitting (6) and replacing (10) by

t_i^* does not admit equality, for all $i = 1..n$.

Rule 72, *spec* ≡ `eqtype` *typdesc* Due to its side-condition, rule 72 is not a structural contraction. By assumption (with $P = B$,$Q = E$)

(B, A) is admissible and grounded in N (1)
$(\nabla, \varphi A)$ is admissible (2)
∇ proves $\varphi B \vdash \mathit{spec} \Rightarrow E$ (3)

Then for some TE and ∇_1

$$\nabla = \frac{\nabla_1}{\varphi B \vdash \mathit{spec} \Rightarrow TE \text{ in Env}} \quad (4)$$

∇_1 proves C of $\varphi B \vdash \mathit{typdesc} \Rightarrow TE$ (5)
$\forall(\theta, CE) \in \mathrm{Ran}\, TE$, θ admits equality (6)

Write *typdesc* in the form

$$\alpha^{(k_1)}\mathit{tycon}_1 \texttt{ and } \cdots \texttt{ and } \alpha^{(k_n)}\mathit{tycon}_n, \quad (n \geq 1)$$

where $\mathit{tycon}_1, \ldots, \mathit{tycon}_n$ are distinct by the syntactic constraint in [Sec 3.5, p 12]. By rule 83, there exist $\theta_1, \ldots, \theta_n$ such that

$TE = \{\mathit{tycon}_1 \mapsto (\theta_1, \{\}), \ldots, \mathit{tycon}_n \mapsto (\theta_n, \{\})\}$ (7)
$\mathrm{arity}(\theta_i) = k_i$, for all $i = 1..n$ (8)

Let $t_1^*, \ldots t_n^*$ be distinct type names such that

$$\text{arity}(t_i^*) = k_i, \text{ for all } i = 1..n \tag{9}$$
$$t_i^* \text{ admits equality, for all } i = 1..n \tag{10}$$
$$\{t_1^*, \ldots, t_n^*\} \cap (\text{names}(B, A) \cup T) = \emptyset \tag{11}$$

Let $\varphi^* = \text{Id}$, $TE^* = \{tycon_1 \mapsto (t_1^*, \{\}), \ldots, tycon_n \mapsto (t_n^*, \{\})\}$ and $E^* = TE^*$ in Env. By rule 83 there exists a ∇_1^* such that

$$\nabla_1^* \text{ proves } C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE^* \tag{12}$$
$$(\nabla_1^*, B, A) \text{ is admissible and grounded in } N \tag{13}$$

using (1) and (11). Let

$$\nabla^* = \frac{\nabla_1^*}{B \vdash \mathit{spec} \Rightarrow E^*}$$

By (10) and (13), (∇^*, A) is admissible and grounded in N and ∇^* proves $B \vdash \mathit{spec} \Rightarrow E^*$. Clearly, ∇^* depends only on B, A, N and *spec*. Finally, let

$$\psi = \varphi + \{t_1^* \mapsto \theta_1, \ldots, t_n^* \mapsto \theta_n\}$$

which really is a realisation because of (8), (9) and (10). Then

$$\psi(C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE^*) = C \text{ of } \varphi B \vdash \mathit{typdesc} \Rightarrow TE$$

For every C, *typdesc*, and TE there can at most be one proof proving $C \vdash \mathit{typdesc} \Rightarrow TE$. We therefore have $\psi(\nabla_1^*) = \nabla_1$. Thus $\psi(\nabla^*) = \nabla$. Also, $\psi(\varphi^* A) = \psi(A) = \varphi(A)$ by (11) showing $\psi(\nabla^*, \varphi^* A) = (\nabla, \varphi A)$ as required.

Rule 73, *spec* ≡ `datatype` *datdesc* Rule 73 [p 40] is not a structural contraction, because of the TE occurring on the left in the premise.

Without loss of generality, assume that

$$\begin{aligned} \mathit{datdesc} &\equiv \alpha^{(k)} \mathit{tycon} \texttt{ = } \mathit{condesc} \\ \mathit{condesc} &\equiv \mathit{con} \texttt{ of } \mathit{ty} \end{aligned}$$

(the general case of several type constructors and several value constructors presents no extra difficulty, and neither does the case where "`of` *ty*" is absent).

By assumption (with $P = B$, $Q = E$)

$$(B, A) \text{ is admissible and grounded in } N \tag{1}$$
$$(\nabla, \varphi A) \text{ is admissible} \tag{2}$$
$$\nabla \text{ proves } \varphi B \vdash \mathit{spec} \Rightarrow E \tag{3}$$

Then ∇ is of the form

$$\begin{array}{c}
\nabla_1 \\
\Big| \\
(85) \\
\Big| \\
C \text{ of } \varphi B + TE, \alpha^{(k)} t \vdash \boxed{con \texttt{ of } ty} \Rightarrow \underbrace{\{con \mapsto \tau \to \alpha^{(k)} t\}}_{CE} \\
\Big| \\
(84) \\
\Big| \\
C \text{ of } \varphi B + TE \vdash \boxed{\alpha^{(k)}\ tycon \texttt{ = } condesc} \Rightarrow VE, TE \\
\Big| \\
(73) \\
\Big| \\
\varphi B \vdash \boxed{\texttt{datatype } \alpha^{(k)}\ tycon \texttt{ = } condesc} \Rightarrow (VE, TE) \text{ in Env}
\end{array}$$

where $VE = \text{Clos}CE$, $TE = \{tycon \Rightarrow (t, \text{Clos}CE)\}$ and

$$\nabla_1 \text{ proves } C \text{ of } \varphi B + TE \vdash ty \Rightarrow \tau \tag{4}$$

Now choose $t^* \notin \text{names}(B, A) \cup T$, with arity k, not admitting equality. Let $TE_0^* = \{tycon \mapsto (t^*, \{\})\}$. (Note the empty constructor environment; we shall shortly obtain TE^* by "filling out" the constructor environment of TE_0^*.) Let $\psi = \varphi + \{t^* \mapsto t\}$.

Let $TE_0 = \{tycon \mapsto (t, \{\})\}$. Note that C of $\varphi B + TE_0$ is admissible. Thus we can apply Lemma A.4 (second half) to (4). Hence there exists a ∇_0 which proves C of $\varphi B + TE_0 \vdash ty \Rightarrow \tau$. Note that C of $B + TE_0^*$ is admissible and that $\psi(C \text{ of } B + TE_0^*) = C$ of $\varphi B + TE_0$. Then, by Lemma A.3(2), there exist ∇_0^* and τ^* such that

$$\psi \nabla_0^* = \nabla_0 \tag{5}$$

$$\nabla_0^* \text{ proves } C \text{ of } B + TE_0^* \vdash ty \Rightarrow \tau^* \tag{6}$$

Now let $TE^* = \{tycon \mapsto (t^*, \text{Clos}CE^*)\}$, where $CE^* = \{con \mapsto \tau^* \to \alpha^{(k)} t^*\}$. Note that C of $B + TE^*$ is admissible, since $t^* \notin \text{names}(B, A)$. Thus we can apply Lemma A.4 (second half) to (6), so there exists a ∇_1^* such that

$$\nabla_1^* \text{ proves } C \text{ of } B + TE^* \vdash ty \Rightarrow \tau^* \tag{7}$$

Using that $\psi \tau^* = \tau$ we get

$$\psi(C \text{ of } B + TE^*) = C \text{ of } \varphi B + TE \tag{8}$$

which is admissible. Thus we can apply Lemma A.3(1) to (7); hence

$$\psi \nabla_1^* \text{ proves } C \text{ of } \varphi B + TE \vdash ty \Rightarrow \tau \tag{9}$$

By Lemma A.4 (first half) on (9) and (4) we therefore have

$$\psi \nabla_1^* = \nabla_1 \tag{10}$$

Now let $\varphi^* = \mathrm{Id}$ and $\nabla^* =$

$$\nabla_1^*$$
$$(85)$$
$$C \text{ of } B + TE^*, \alpha^{(k)}t^* \vdash \boxed{con \texttt{ of } ty} \Rightarrow \underbrace{\{con \mapsto \tau^* \to \alpha^{(k)}t^*\}}_{CE^*}$$
$$(84)$$
$$C \text{ of } B + TE^* \vdash \boxed{\alpha^{(k)}\ tycon \texttt{ = } condesc} \Rightarrow VE^*, TE^*$$
$$(73)$$
$$B \vdash \boxed{\texttt{datatype } \alpha^{(k)}\ tycon \texttt{ = } condesc} \Rightarrow E^*$$

where $VE^* = \mathrm{Clos}CE^*$ and $E^* = (VE^*, TE^*)$ in Env (CE^* and TE^* have already been defined). By Lemma A.4 (first half) on (6), τ^* is uniquely determined by C of $B + TE_0^*$ and ty. But then, by Lemma A.4 (first half) on (7), ∇_1^* depends only on C of $B + TE_0^*$ and ty. It follows that φ^*, E^* and ∇^* depend only on B, A, N and *spec*, as required.

Note that $(C \text{ of } B + TE^*, A')$ is admissible and grounded in N, where $A' = (B, A)$. Thus, by Lemma A.5 on (7), (∇_1^*, A') is admissible and grounded in N. It follows that (∇^*, A) is admissible and grounded in N, as required. Then, given (7), one easily checks that ∇ proves $\varphi^* B \vdash spec \Rightarrow E^*$, using rules 73, 84 and 85.

Finally

$$\begin{array}{rcll} \psi(\nabla^*, \varphi^* A) & = & (\psi\nabla^*, \varphi A) & \text{as } t^* \notin \mathrm{names}\, A \\ & = & (\nabla, \varphi A) & \end{array}$$

since $t^* \notin \mathrm{names}(B, A)$, $\psi(t^*, \tau^*) = (t, \tau)$ and (10).

The way ∇_1^* was achieved is summarised in the diagram below, which in the proof is traversed clockwise, starting from ∇_1.

$$\begin{array}{cc} \nabla_0^* \text{ proves } C \text{ of } B + TE_0^* \vdash ty \Rightarrow \tau^* & \nabla_1^* \text{ proves } C \text{ of } B + TE^* \vdash ty \Rightarrow \tau^* \\ \downarrow \psi & \downarrow \psi \\ \nabla_0 \text{ proves } C \text{ of } \varphi B + TE_0 \vdash ty \Rightarrow \tau & \nabla_1 \text{ proves } C \text{ of } \varphi B + TE \vdash ty \Rightarrow \tau \end{array}$$

Rule 88, Structure Sharing | Here *phrase* is a sharing equation

$$shareq \equiv longstrid_1 \texttt{ = } \cdots \texttt{ = } longstrid_k$$

for some $k \geq 2$. By assumption (with $P = B$, $Q = \{\}$)

(B, A) is admissible and grounded in N (1)

$(\nabla, \varphi A)$ is admissible (2)

∇ proves $\varphi B \vdash \mathit{shareq} \Rightarrow \{\}$ (3)

Now by the side-condition on rule 88,

$$m \text{ of } (\varphi B)(\mathit{longstrid}_1) = \cdots = m \text{ of } (\varphi B)(\mathit{longstrid}_k) \tag{4}$$

but the problem is that we do not necessarily have

$$m \text{ of } B(\mathit{longstrid}_1) = \cdots = m \text{ of } B(\mathit{longstrid}_k)$$

(One reason is that whenever we choose names in the proof, e.g. in the cases for rules 64 and 79, we always choose them to be suitably "new". Also, the top-level B may fail to satisfy the sharing equations.) However, let X be any structure identifier, let m be a structure name such that $m \notin \text{names}(B, A)$ and consider the assembly

$$A' = \Big(B, A, (m, \{X \mapsto B(\mathit{longstrid}_1)\}), \ldots, (m, \{X \mapsto B(\mathit{longstrid}_k)\})\Big)$$

Although A' need not be consistent, it is grounded in N. Moreover, letting $\varphi' = \varphi + \{m \mapsto m'\}$, where $m' \notin \text{names}(\varphi B, \varphi A)$, we have that $\varphi' A'$ is admissible, using (2) and (4). Note that φ' is fixed on N.

We can therefore apply the admissification theorem (Theorem 10.3) to A' and φ'. Hence there exists a realisation φ^*, depending only on N and A', such that $\varphi^*(A')$ is admissible and grounded in N, and there exists a realisation ψ fixed on N such that

$$\psi(\varphi^* A') = \varphi' A' \tag{5}$$

Let $Q^* = \{\}$ and $\nabla^* = \dfrac{}{\varphi^* B \vdash \mathit{shareq} \Rightarrow \{\}}$. Note that φ^*, Q^* and ∇^* depend on B, A, N and *phrase* only. Since $\varphi^* A'$ is admissible and grounded in N, we have that $(\nabla^*, \varphi^* A)$ is admissible and grounded in N as required. Moreover, since $\varphi^* A'$ is consistent, we have

$$m \text{ of } (\varphi^* B)(\mathit{longstrid}_1) = \cdots = m \text{ of } (\varphi^* B)(\mathit{longstrid}_k)$$

so ∇^* really proves $\varphi^* B \vdash \mathit{shareq} \Rightarrow \{\}$ as required. Finally,

$$\begin{array}{lcll} \psi(\nabla^*, \varphi^* A) & = & (\nabla, \varphi' A) & \text{by (5)} \\ & = & (\nabla, \varphi A) & \text{since } m \notin \text{names}(B, A) \end{array}$$

as required.

Rule 89, Type Sharing | Here *phrase* is a sharing equation

$$shareq \equiv \texttt{type}\ longtycon_1\ \texttt{=}\ \cdots\ \texttt{=}\ longtycon_k$$

for some $k \geq 2$. The proof is the same as for rule 88, with the following modifications

(a) Replace *longstrid* by *longtycon* throughout
(b) Replace "m of " by "θ of " throughout
(c) X is now a type constructor, not a structure identifier

This concludes the proof of the principality theorem.

A.3 Principal signatures

We are now in a position to prove as corollaries Theorem 11.1 which asserts the existence of barely principal signatures, and Theorem 11.2 which ensures that the conditions for their existence will always be met if we start in the initial basis B_0.

Corollary A.6 (Barely Principal Signatures) *Let B be rigid and grounded, and let $B \vdash sigexp \Rightarrow S$ for some S. Then there exists a barely principal signature for sigexp in B.*

Proof By Theorem A.2, setting $\varphi = \mathrm{Id}$ and $N = N$ of B, we deduce that there exist φ^* fixed on N, ∇^* and S^* depending only on B and *sigexp*, such that

$$(\nabla^*, \varphi^* B) \text{ is admissible and grounded in } N \tag{1}$$
$$\nabla^* \text{ proves } \varphi^* B \vdash sigexp \Rightarrow S^* \tag{2}$$

Moreover, for some ψ fixed on N

$$\psi(\nabla^*, \varphi^* B) = (\nabla, B) \tag{3}$$

But $\varphi^* B = B$ since B is rigid, so from (2) we get

$$\nabla^* \text{ proves } B \vdash sigexp \Rightarrow S^*$$

Now pick $N^* = \mathrm{names}\, S^* \backslash N$. To complete the proof we need to show that $(N^*)S^* \geq S$, i.e. that $\psi' S^* = S$ for some ψ' for which $\mathrm{Supp}\, \psi' \subseteq N^*$. It is enough to take ψ' to be ψ restricted to names S^*; for we have $\psi' S^* = \psi S^* = S$ from (3), and $\mathrm{Supp}\, \psi' \subseteq N^*$ since $N^* = \mathrm{names}\, S^* \backslash N$ and ψ' is fixed on N.

Corollary A.7 (Bases) *Let $B \vdash topdec \Rightarrow B'$ occur in the elaboration of some program in the initial basis B_0. Then B is rigid and grounded.*

Proof (outline) It is easy to check that B_0 is rigid and grounded. We outlined the argument for the rigidity of B in Section 10.1. For groundedness, since a program is just a sequence of top-level declarations, it is enough – combined with a simple induction – to show that if B is rigid and grounded and $B \vdash \mathit{topdec} \Rightarrow B'$ then B' is grounded. Now such an elaboration must conclude with rule 100, 101 or 102. In rules 100 and 102 the result is a rigid basis containing respectively an environment E and a functor environment F, and these are trivially grounded. (Recall that E contains no bound names, and that groundedness imposes no constraint upon functor signatures.)

Therefore we need only deal with rule 101, in which a signature environment G is created. Now let $N = N$ of B. The question then reduces to: If $B \vdash \mathit{sigdec} \Rightarrow G$, is G grounded in N? Tracing back through the elaboration tree, this can be reduced further to: If $B \vdash \mathit{sigexp} \Rightarrow S^*$, with $(N^*)S^*$ principal for *sigexp* in B, then is S^* grounded in N? But S^* here is just the S^* found in Corollary A.6, and from the application of the principality theorem we see that S^* – part of ∇^* – is indeed grounded in $N = N$ of B as we require.

B Appendix: Identifier Status

In this appendix we give rules which determine the status of each occurrence of a long identifier *longid*.

There are nine classes of identifier [Sec 2.4, p 4; Sec 3.2, p 10]. Here we only treat the distinction between value variables Var, value constructors Con and exception constructors ExCon, since there is no problem in assigning an identifier occurrence to any other class. So for this appendix, when we consider an occurrence of *longid*, we assume that it can only be a *longvar*, a *longcon* or an *longexcon*. Also, we ignore derived forms.

We define a *status map* so be a finite map from LongId to {`v`, `c`, `e`}, and we shall use s to range over these three status values. Then, for example, the declaration

```
datatype T = A  val B = A
```

will normally yield the status map which we write {`A`:`c`, `B`:`v`}; this means that, in the scope of this declaration, `A` and `B` will be treated respectively as a value constructor and as a variable (until something else changes their status). But if `B` already possesses `c` or `e` status, then the declaration will instead yield the status map {`A`:`c`} with no entry for `B`, because in that case `val` no longer declares `B` but treats it as a pattern (leading in this case to a failure of elaboration).

We say that a phrase *assigns* *longid*:s if it yields a status map M such that $M(longid) = s$. We assume that status maps compose by modification (+, [p 18]) and that normal scoping rules apply. Thus, if declarations dec_1 and dec_2 yield status maps M_1 and M_2, then the declaration dec_1 `;` dec_2 will yield M_1+M_2, while `local` dec_1 `in` dec_2 `end` will yield just M_2. Also, `abstype` *datbind* `with` ... is treated like `local datatype` *datbind* `in` .... Other phrases that are not mentioned in the rules below yield the empty status map.

Declarations and Specifications

1. A `datatype` declaration or specification assigns *id*:`c` for each *id* which it introduces as a value constructor.

2. An `exception` declaration or specification assigns *id*:`e` for each *id* which it introduces as an exception constructor.

3. A `val` declaration yields the union of the status-maps yielded by its patterns. This must be a well-defined map, because it can only assign `v` status to any *id*. On the other hand, a `val` specification assigns *id*:`v` for every *id* which it contains (ignoring type expressions) [Sec 3.2, p 10].

4. A pattern *pat* assigns *id*:`v` for every *id* which does not already have `c` or `e` status and occurs in *pat* (ignoring type expressions).

5. A `structure` declaration containing the binding *strid* `=` *strexp* assigns the status *strid*`.`*longid* `:`*s*, whenever *longid* `:`*s* is assigned by *strexp*.
   Similarly a `structure` declaration containing the binding *strid* `:` *sigexp* `=` *strexp*, or a `structure` specification containing the description *strid* `:` *sigexp*, assigns the status *strid*`.`*longid* `:`*s*, whenever *longid* `:`*s* is assigned by *sigexp*.

6. An `open` declaration or specification containing the structure identifier *strid* assigns the status *longid* `:`*s*, whenever *strid*`.`*longid* `:`*s* is assigned by the declaration or specification of *strid*.

7. An `include` specification accumulates the status maps yielded by the signature identifiers it contains.

Rules 5–7 depend on how a structure expression *strexp* or a signature expression *sigexp* yields a status map; this will be defined by rules 8–10 below.

Example B.1 Here are some examples of status maps yielded by declarations and specifications, assuming an initially empty status map in each case:

(a)	`datatype T=A val B=A`	{`A`:`c`, `B`:`v`}
(b)	`datatype T=A val A=A`	{`A`:`c`}
(c)	`datatype T=A|B datatype T=B|C`	{`A`:`c`, `B`:`c`, `C`:`c`}
(d)	`local val (A,B)=(2,3) in exception A end`	{`A`:`e`}
(e)	`datatype T=A val A:T`	{`A`:`v`}

In (c), which may be either a declaration or a specification, note that `A` remains a constructor of the *hidden* type `T`. In (d), note that `local` localises `B`'s status, and also that the status `e` can supersede the status `v`. Contrast particularly the declaration (b) with the specification (e); a `val` declaration is sensitive to previous constructor status, but a `val` specification is not.

Note that rule 4 above, for patterns, deals with variables bound in a *match*.

Example B.2 Assume the signature declaration

```
signature SIG = sig type T val A:T end
```

Then the structure declaration

```
structure S:SIG = struct datatype T=A|B end
```

will yield {`S.A`:`v`}. Without "`:SIG`" it would yield {`S.A`:`c`, `S.B`:`c`}. Thereafter, the declaration

```
datatype U=A|C;  open S
```

will yield {`A`:`v`, `C`:`c`}. Note how `open` can override constructor status.

Structure and signature expressions

8. The structure expression `struct` *dec* `end` yields the same status map as *dec*. Similarly the signature expression `sig` *spec* `end` yields the same status map as *spec*.

9. The structure expression $strid_1.\cdots.strid_k.strid$, if *strid* was declared by *strid* $\langle$`:` *sigexp*$\rangle$ `=` *strexp* or specified by *strid*`:` *sigexp* , yields the same status map as was yielded by *sigexp* if present, else as *strexp*. Similarly, the signature expression *sigid* , if *sigid* was declared by *sigid* `=` *sigexp* , yields the same status map as *sigexp*.

10. The structure expression *funid*`(`*strexp*`)` , if *funid* was declared or specified by *funid*`(`*strid*`:` *sigexp*`)` $\langle$`:` *sigexp*$'\rangle$ $\langle\langle$`=` *strexp*$'\rangle\rangle$ yields the same status map as was yielded by *sigexp*$'$ if present, else as *strexp*$'$.

In rule 10, note that *strexp* as a functor argument has no effect on status. Note also that functor and signature declarations have no immediate effect on status; they only take effect via subsequent use of the declared functor identifiers and signature identifiers.

Example B.3 To illustrate some of these points, consider the artificial program

```
exception A and B ;                              (*1*)
signature SIG = sig type T  val A:T end ;        (*2*)
functor F(S: SIG)                                (*3*)
        = struct val C = B  open S end ;         (*4*)
structure R = F(struct datatype T=A end) ;       (*5*)
```

The exception declaration yields the status map $M = \{\mathtt{A}\!:\!\mathtt{e}, \mathtt{B}\!:\!\mathtt{e}\}$, and M applies at all the points `1`, `2` and `4` because signature and functor declarations don't affect status directly. But the signature expression `SIG` yields $M_{\mathtt{SIG}} = \{\mathtt{A}\!:\!\mathtt{v}\}$, so at `3` the status map $M{+}\mathtt{S}.M_{\mathtt{SIG}} = \{\mathtt{A}\!:\!\mathtt{e}, \mathtt{B}\!:\!\mathtt{e}, \mathtt{S.A}\!:\!\mathtt{v}\}$ applies (here `S:SIG` is treated as a specification). Then the functor body yields $M_{\mathtt{F}} = \{\mathtt{A}\!:\!\mathtt{v}, \mathtt{C}\!:\!\mathtt{v}\}$, the structure expression `F(..)` also yields $M_{\mathtt{F}}$, the structure declaration yields $\mathtt{R}.M_{\mathtt{F}} = \{\mathtt{R.A}\!:\!\mathtt{v}, \mathtt{R.C}\!:\!\mathtt{v}\}$, and finally at `5` the status map $M{+}\mathtt{R}.M_{\mathtt{F}} = \{\mathtt{A}\!:\!\mathtt{e}, \mathtt{B}\!:\!\mathtt{e}, \mathtt{R.A}\!:\!\mathtt{v}, \mathtt{R.C}\!:\!\mathtt{v}\}$ applies.

This concludes our rules for assigning identifier status. They are complete; in a program with no unbound identifiers they determine the status of every occurrence of a *longid*. The rules could be formalised, and indeed the status map could even have been combined with the static environment, so that elaboration could be given the task of assigning status. We resisted this in the Definition, to avoid too much complexity at once.

C Appendix: Solutions to Exercises

This Appendix contains the solutions to all the exercises.

1.1 See Figure 12

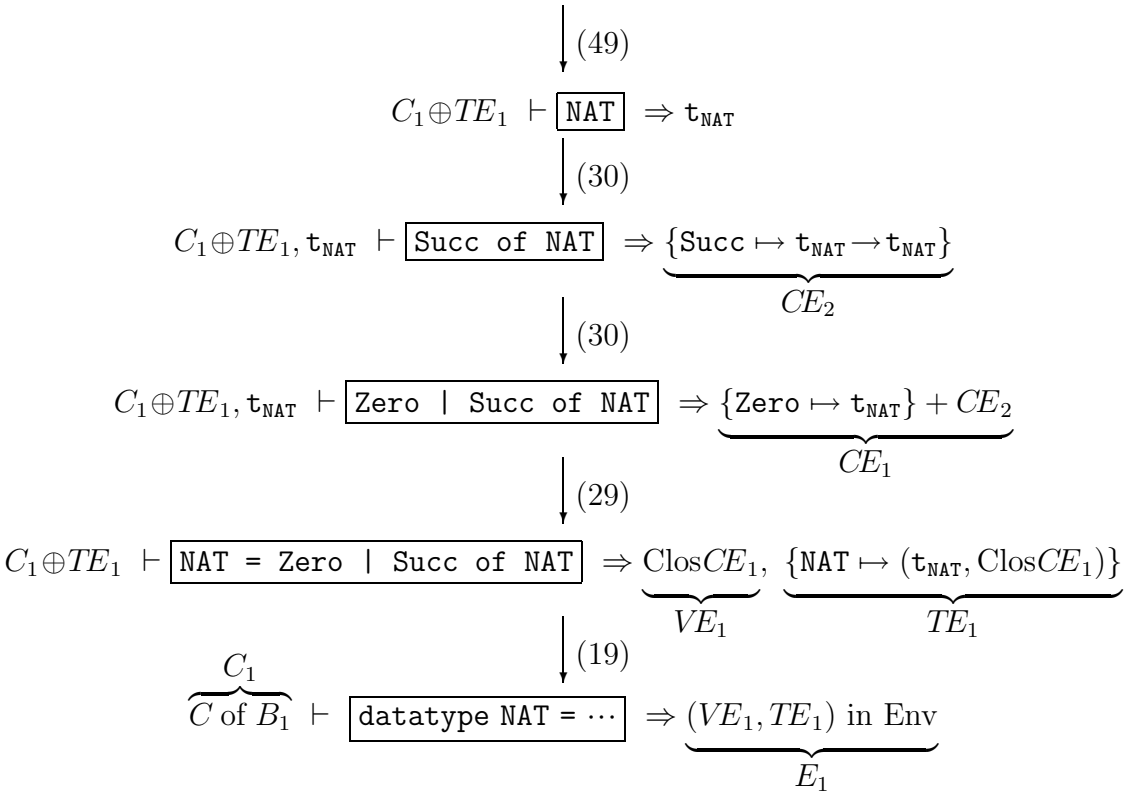

Figure 12: Elaborating a `datatype` declaration

1.2 See Figure 13

3.1 $E = (\{\}, \{\mathtt{A} \mapsto \mathtt{A}, \mathtt{B} \mapsto \mathtt{B}\}, \{\})$. (Without the new rule, E would be empty.) $I = (\{\}, \{\mathtt{A}\}, \emptyset)$. $E \downarrow I = (\{\}, \{\mathtt{A} \mapsto \mathtt{A}\}, \{\})$ is bound to `S`.

3.2 Let $IB(sigid_i) = I_i$, $1 \leq i \leq n$. (If any is undefined then it is easy to show that neither phrase evaluates.) Then by rule 181

$$IB \vdash \mathtt{include}\ sigid_1 \cdots sigid_n \Rightarrow I_1 + \cdots + I_n$$

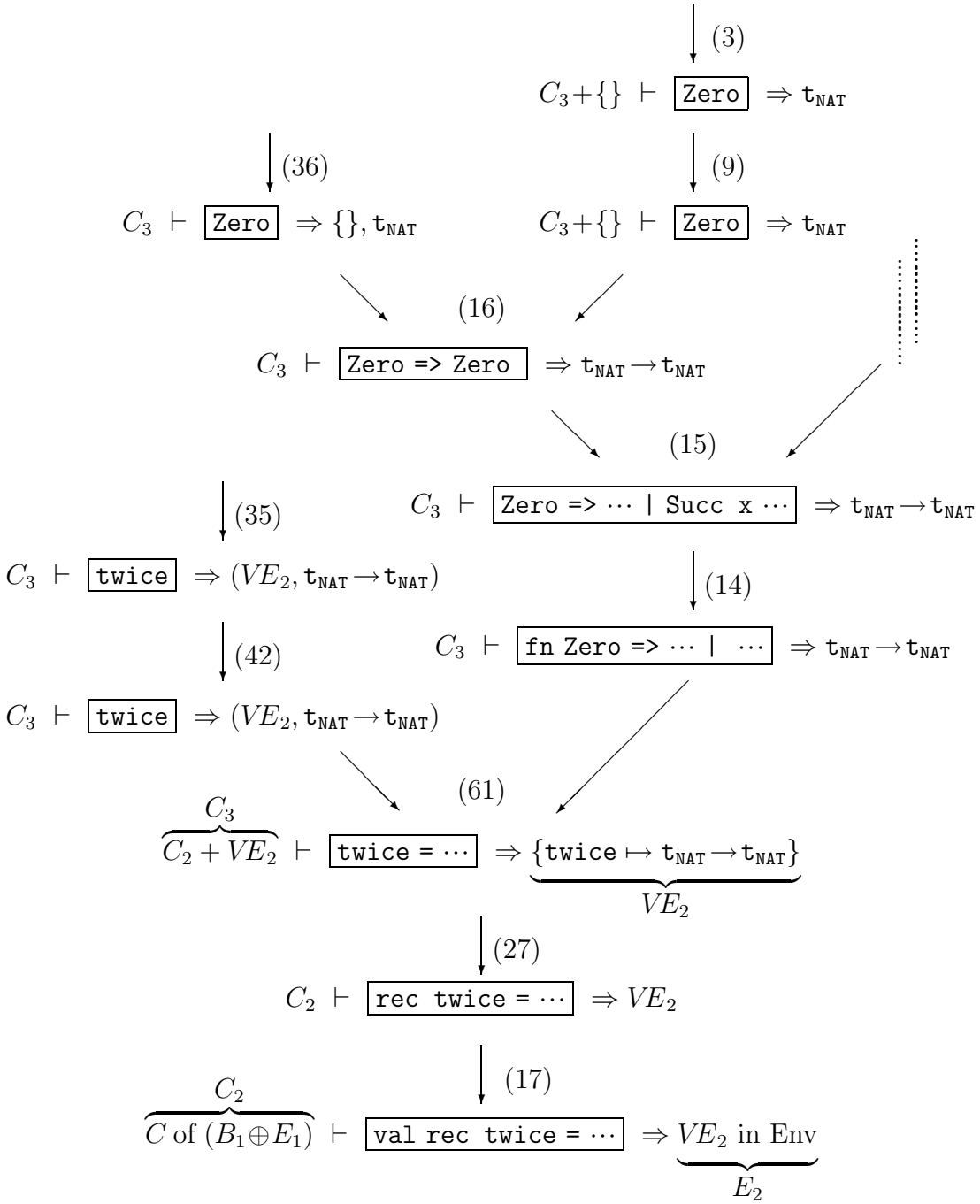

Figure 13: Elaborating a `fun` declaration

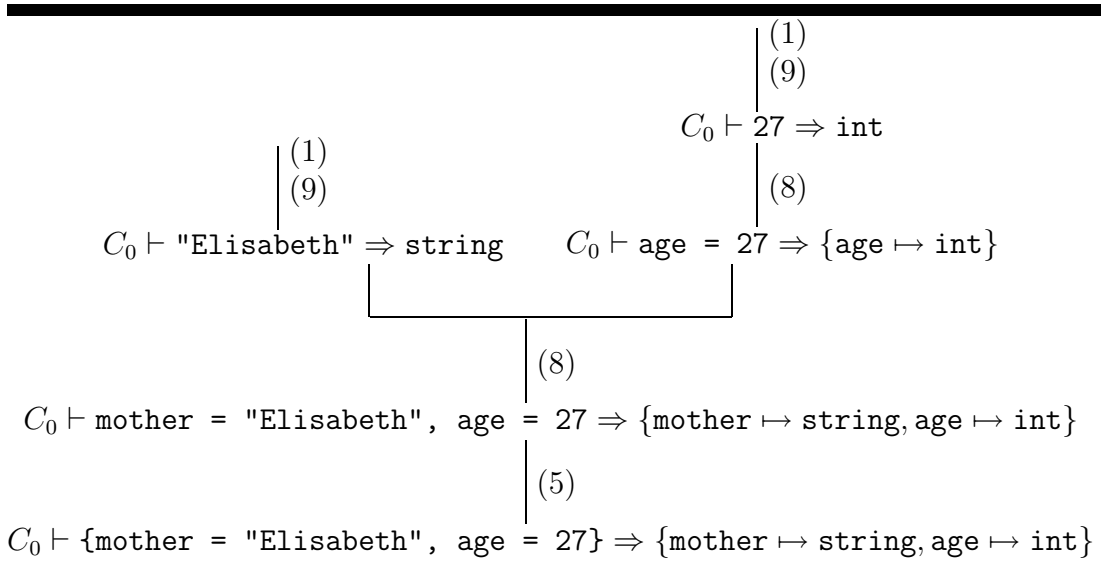

Figure 14: Elaborating a record expression

On the other hand, let *strdesc* = *strid*$_1$: *sigid*$_1$ `and` $\cdots$ `and` *strid*$_n$: *sigid*$_n$. Then by rules 171 and 186

$$IB \vdash strdesc \Rightarrow \{strid_1 \mapsto I_1, \ldots, strid_n \mapsto I_n\}$$

Hence by rules 178–180 we deduce that

$$\texttt{local structure}\ strdesc\ \texttt{in open}\ strid_1 \cdots strid_n\ \texttt{end} \Rightarrow I_1 + \cdots + I_n$$

3.3 The functor closure is $(\texttt{X}\colon I, \texttt{X}, B)$. The structure is $E \downarrow I$ in each case.

3.4 First, all signatures in structure bindings and result signatures in functor bindings could be deleted (ignored). Second, the result interface in a functor closure would be removed, so a functor closure becomes just $(strid : I, strexp, B)$. Third, rules 169 for structure bindings, 187 for functor bindings and 162 for functor application would all be modified by omitting their first option $\langle\,\rangle$.

4.1 See Figure 14

4.2 (1) elaborates to all types of the form $(\tau\ \texttt{list} \to \texttt{int}) * \tau\ \texttt{list} * \texttt{int} \to \texttt{int}$, for all types τ. (2) elaborates, but only to the type $(\texttt{int list} \to \texttt{int}) * \texttt{int list} \to \texttt{int}$. (3) does not elaborate; by the theorem, `length` has a simple type scheme and no simple type scheme generalises both $\texttt{int list} \to \texttt{int}$ and

`bool list` → `int`. (4) does not elaborate; a `case` expression is a derived form, here for

```
(fn x as [] => 0::x::x | other => [0]::other) s
```

so `x` must have a simple type scheme according to the theorem, but no simple type scheme generalises both `int list` and `int list list`.

4.3 (1) elaborates, (2) elaborates to `int list` → `int`, and (3) does not elaborate because `k` must have a simple type scheme.

4.4 The wrong inference is the one marked (**not 17**) in Figure 15. Notice that `'a` occurs free in C_1, namely in the type of `x`. The quantification on `'a` destroys the link between the types of `x` and `y`.

4.5 In both cases the declaration will then elaborate. In the first case because it becomes possible to give `Id` a general type scheme, provided we avoid the explicit type variable when we elaborate `fn z => z`. In the second case, the scoping of the explicit type variable moves inwards to the inner value declaration, so again we can make `Id` polymorphic.

4.6 He realised that his function can elaborate to the type scheme $\forall().\tau$ `list` → τ `list` for all imperative types τ, but not to $\forall$`'_a.'_a list` → `'_a list`, since the outer `let` expression is expansive. Thus Reno cannot use his function first on an `int` list and then on a `bool` list, say.

Reno's function illustrates just how subtle typing side-effects can be. You might expect that Reno's function should get the type $\forall$`'_a.'_a list` → `'_a list`, or even $\forall$`'a.'a list` → `'a list`. But notice that if Reno had forgotten the "inner" initialisation `res:= [ ]` these type schemes would no longer be valid (consider what happens the *second* time this function is called).

5.1 Let C be a context and assume `t` $\notin T$ of C. The resulting type environment consists of the following bindings

$$\begin{array}{lcl}
\texttt{tree} & \mapsto & (\texttt{t}, \{\texttt{LEAF} \mapsto \forall\texttt{'a.'a} \to \texttt{'a t}, \\
 & & \quad \texttt{TIP} \mapsto \forall\texttt{'a.'a*'a t*'a t} \to \texttt{'a t}\}) \\
\texttt{heap} & \mapsto & (\texttt{t}, \{\}) \\
\texttt{intheap} & \mapsto & (\Lambda().\texttt{int t}, \{\})
\end{array}$$

Notice that `tree` and `heap` share with each other but not with `intheap`.

5.2 To the environment (SE, TE, VE, EE), where SE and EE are empty, and

$$\begin{array}{l}
TE = \{\texttt{options} \mapsto (\texttt{t3}, \{\texttt{MAYBE} \mapsto \texttt{t3}\}), \\
\qquad \texttt{positive} \mapsto (\texttt{t2}, \{\texttt{YES} \mapsto \texttt{t2}\})\}
\end{array}$$

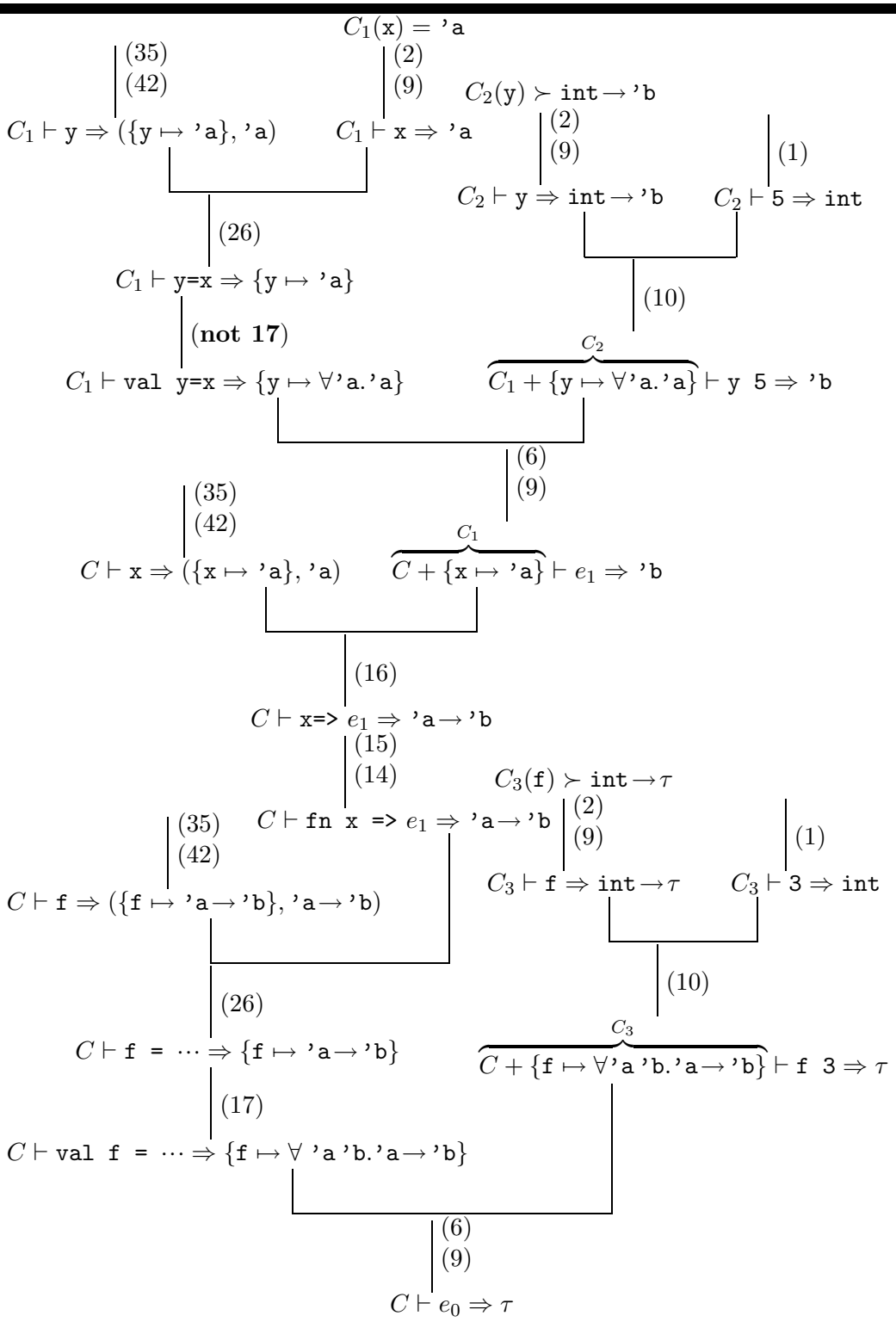

Figure 15: Unsound quantification

and $VE = \{\texttt{YES} \mapsto \texttt{t2}, \texttt{NO} \mapsto \texttt{t1}, \texttt{MAYBE} \mapsto \texttt{t3}\}$. Since $VE(\texttt{YES}) \neq VE(\texttt{NO})$, (a) elaborates to `t2`; (b) elaborates to `t1`, even though the first binding of `options` is overwritten by the later declaration; (c) does not elaborate.

5.3 No, see [App C, p 74].

5.4 (1) and (3) are legal; they both evaluate to `false`.

5.5 The type functions bound to `pair` and `intmap_store` admit equality, although the type function bound to `intmap` does not.

5.6 None whatsoever. The equality and imperative attributes of the bound type variables of a type function have no significance [Sec 4.4, p 19].

5.7 No; `T` will possess the equality attribute after the whole `local` declaration, but not after the whole `abstype` declaration. Otherwise the effect is identical.

6.1 Analogous to the solution of Exercise 3.2. In this case, show that each phrase elaborates to $E_1 + \cdots + E_n$ if and only if there exist structure names $m_1, \ldots, m_n$ such that the condition $B(sigid_i) \geq (m_i, E_i)$ is satisfied for all i.

6.2

$$\Big\{\texttt{A} \mapsto \Big(\texttt{m}, \{\texttt{t} \mapsto (\texttt{t}, \{\texttt{BLUE} \mapsto \texttt{t}, \texttt{RED} \mapsto \texttt{t}\})\}, \{\texttt{BLUE} \mapsto \texttt{t}, \texttt{RED} \mapsto \texttt{t}, \texttt{x} \mapsto \texttt{t}, \texttt{y} \mapsto \texttt{t}, \texttt{p} \mapsto \texttt{t} * \texttt{t}\}\Big),$$
$$\texttt{A1} \mapsto \Big(\texttt{m}, \{\texttt{t} \mapsto (\texttt{t}, \{\})\}, \{\texttt{x} \mapsto \texttt{t}, \texttt{y} \mapsto \texttt{t}\}\Big),$$
$$\texttt{A2} \mapsto \Big(\texttt{m}, \{\texttt{t} \mapsto (\texttt{t}, \{\})\} \{\texttt{p} \mapsto \texttt{t} * \texttt{t}\}\Big)\Big\}$$

6.3

(b) `struct end`
(c) `struct type t = bool end`
(d) `struct type t = bool val x = false end`
(e) `struct type t = bool val x = 3 end`

7.1 `VIEW1` elaborates to

$$(N_1)S_1 = \{\texttt{m1}, \texttt{t1}\}(\texttt{m1}, \{\texttt{t} \mapsto (\texttt{t1}, \{\})\}, \{\texttt{x} \mapsto \texttt{t1}, \texttt{y} \mapsto \texttt{t1}\})$$

and `A` to $S =$

$$(\texttt{m2}, \{\texttt{t} \mapsto (\texttt{t2}, \{\texttt{BLUE} \mapsto, \texttt{t2}, \texttt{RED} \mapsto \texttt{t2}\}\}), \{\texttt{BLUE} \mapsto \texttt{t2}, \texttt{RED} \mapsto \texttt{t2}, \texttt{x} \mapsto \texttt{t2}, \texttt{y} \mapsto \texttt{t2}, \texttt{p} \mapsto \texttt{t2} * \texttt{t2}\})$$

Let S^- be $(\mathtt{m2}, \{\mathtt{t} \mapsto (\mathtt{t2}, \{\})\}, \{\mathtt{x} \mapsto \mathtt{t2}, \mathtt{y} \mapsto \mathtt{t2}\})$. Then S^- is an instance of $(N_1)S_1$ via the realisation which maps `m1` to `m2` and `t1` to `t2`. Incidently, S^- is precisely the structure to which `A1` is bound; it shares with S although it has fewer components.

7.2 We have $\mu(\tau_1 \to (\mathtt{'a}, \mathtt{'b})t_1) = \tau_2 \to (\mathtt{'a}, \mathtt{'b})t_2$ for some substitution μ, so first we have $t_1 = t_2$, $\mu(\mathtt{'a}) = \mathtt{'a}$ and $\mu(\mathtt{'b}) = \mathtt{'b}$. Hence, because τ_1 contains no other type variables, $\tau_2 = \mu(\tau_1) = \tau_1$.

7.3 `Pair` matches `PAIR` via the realisation $(\{\mathtt{m} \mapsto \mathtt{m1}\}, \{\mathtt{t} \mapsto \Lambda\mathtt{'a.'a} * \mathtt{'a}\})$. `Complex` matches `COMPLEX` via the realisation $(\{\mathtt{m} \mapsto \mathtt{m1}, \mathtt{m'} \mapsto \mathtt{m2}\}, \{\mathtt{t} \mapsto \Lambda\mathtt{'a.'a} * \mathtt{'a}, \mathtt{t'} \mapsto \Lambda().\mathtt{real} * \mathtt{real}\})$. The type schemes for `fst`, `snd`, `mk_pair` and `mk_complex` are less polymorphic in the signature instance than in the actual structure. The realisations are uniquely determined. For types, the point is that for every flexible type name t in the signature there is a type constructor tycon bound to (t, CE) for some CE. Thus t must be instantiated to the corresponding type function in the actual structure. Once the realisation has been done, it simply remains to check that the type schemes of the values of the instance are generalised by the type schemes in the actual structure.

7.4 `A` does not match `SIG`, for a type name of arity 1 cannot be realised by a nullary type function. `B` does not match `SIG` either, for after instantiation of the type name in `SIG`, one must get precisely the type structure of `A`, cf. the definition of the enrichment relation $(\theta_1, CE_1) \succ (\theta_2, CE_2)$ [Sec 5.11, p 34]. In general, a datatype specification can only be matched by a datatype declaration if the latter has the same constructors as the former and every constructors is specified and declared with the same type.

8.1 The functor signature is

$$\{\mathtt{m1}, \mathtt{t1}\}\Big(\begin{array}{l} (\mathtt{m1}, \{\mathtt{t} \mapsto \mathtt{t1}\}), \\ \{\mathtt{m2}, \mathtt{t2}\}(\mathtt{m2}, \{\mathtt{u} \mapsto (\mathtt{t2}, CE)\}, CE)\Big) \end{array}$$

where $CE = \{\mathtt{C} \mapsto \mathtt{t1} \to \mathtt{t2}, \mathtt{D} \mapsto \mathtt{t2}\}$ and neither `t1` nor `t2` admits equality. (If `t` had been specified as an `eqtype` then both `t1` and `t2` would admit equality.)

8.2 An admissible inference tree proves $B \vdash \mathit{fundec}_2 \Rightarrow F$ if and only if it is of the form shown in Figure 16, where $\Sigma' \geq S'' \prec S'$ (by the side condition on rule 62 [p 39]), $B_1 = B \oplus \{\mathit{strid} \mapsto S\}$, $B_2 = B_1 \oplus \{\mathit{strid}' \mapsto S''\}$ and $N' = \operatorname{names} S'' \setminus \operatorname{names}((N \text{ of } B) \cup N)$. The desired result follows directly from a comparison with rule 99.

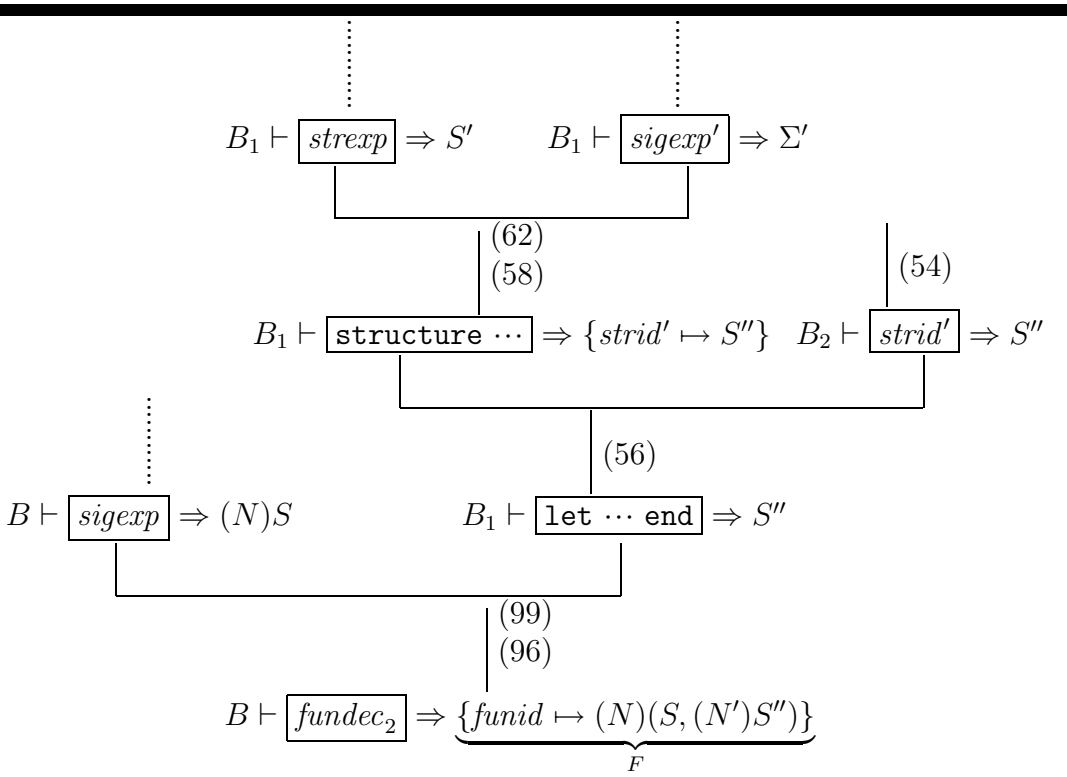

Figure 16: Regarding the result signature of functor as a derived form

8.3 Consider

$strexp_1 \equiv$
```
let structure A: sig type t end =
    struct type t = int end
in struct datatype u = C of A.t | D end
end
```

$strexp_2 \equiv$
```
let functor F(A: sig type t end) =
    struct datatype u = C of A.t | D end
in F(struct type t = int end)
end
```

In the result of elaborating $strexp_1$, `u` admits equality, but in the result of elaborating $strexp_2$, `u` does not admit equality.

8.4 The functor signature for `F` is $(N)(S^*, (\emptyset)S^*)$. In the first declaration, the condition is that $(N)S^* \geq S' \prec S$, and the result is the unique (by type-explication) such structure S'. In the second declaration, one requires a functor-instance $(S'', (N')S') \leq (N)(S^*, (\emptyset)S^*)$ for which $S'' \prec S$, and then the result is S'. But any such functor instance must have $N' = \emptyset$ and $S' = S''$, so the condition and result are the same in the two cases.

8.5 Rule 55 remains unchanged (this was a trap). In rule 62, replace $S\langle'\rangle$ by S. In rule 99, replace $S'\langle'\rangle$ by S' (in two places, after taking account of the correction in Appendix D).

8.6 The following modified version of rule 62 expresses the desired concept:

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow S \qquad B \vdash \mathit{sigexp} \Rightarrow (N')S' \\ S \text{ matches } (N')S' \quad N' \cap N \text{ of } B = \emptyset \end{array}}{B \vdash \mathit{strid}\ \texttt{<:}\ \mathit{sigexp} = \mathit{strexp} \Rightarrow \{\mathit{strid} \mapsto S'\}}$$

Note that the names N' are chosen to be fresh with respect to the basis. Similarly, one could use the following variant of rule 99

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (N)S \quad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow \Sigma' \\ S' \text{ matches } \Sigma' \end{array}}{\begin{array}{r} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \texttt{<:}\ \mathit{sigexp}'\ \texttt{=}\ \mathit{strexp} \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, \Sigma')\} \end{array}}$$

Alternatively, *funid* `(` *strid* `:` *sigexp* `)` `<:` *sigexp*′ `=` *strexp* could be defined as a derived form, namely

funid `(` *strid* `:` *sigexp* `)` `=`
  `let structure` *strid*′ `<:`*sigexp*′ `=` *strexp*
  `in`
    strid′
  `end`

9.1 Yes. The set of all type structures is consistent; note that `A.tree` and `B.B.tree` are consistent and that even `A.tree` and `B.A.tree` are consistent, since `t` $\neq$ `t2`. As for structure name consistency, `A` and `B.B` have the same name and `A.tree` and `B.B.tree` share as required. Were we to replace `m2` by `m`, consistency would be violated, even if we also replaced `t2` by `t`, for $\{\texttt{L}, \texttt{N}\} \neq \{\texttt{N}\}$.

9.2 The declaration elaborates to

$$\Big\{\texttt{SIG} \mapsto \{\texttt{m1}, \texttt{m2}, \texttt{m3}\}\Big(\texttt{m1}, \{\texttt{A} \mapsto (\texttt{m2}, \{\texttt{C} \mapsto (\texttt{m3}, \{\})\}),\ \texttt{B} \mapsto (\texttt{m2}, \{\texttt{C} \mapsto (\texttt{m3}, \{\})\})\}\Big)\Big\}$$

Notice that the sharing of `A` and `B` implies sharing of `A.C` and `B.C` in order to maintain consistency.

9.3 One example is $S_1 = (\texttt{m}, \{\texttt{A} \mapsto (\texttt{m1}, \{\})\})$, $S_2 = (\texttt{m}, \{\})$, and $S_3 = (\texttt{m}, \{\texttt{A} \mapsto (\texttt{m2}, \{\})\})$.

9.4 No. The problem is that in order to satisfy the last sharing constraint, `D` and `E` must have the same name and hence, because of previous sharing constraints, `A` and `A.B.C` must have the same name, and this would give a cycle. We see the "global" nature of admissibility in this example: one might mistakingly think that it is trivial for `D` and `E` to share, since they are both specified using empty specifications.

10.1 The result of the elaboration is

$$\begin{array}{l}\big(\mathtt{m},\{\mathtt{t}\mapsto(\mathtt{t},\{\}),\\ \quad\mathtt{u}\mapsto(\mathtt{t},\{\}),\\ \quad\mathtt{v}\mapsto(\mathtt{t},\{\mathtt{C}\mapsto\mathtt{t},\mathtt{D}\mapsto\mathtt{t}\})\},\\ \{\mathtt{C}\mapsto\mathtt{t},\mathtt{D}\mapsto\mathtt{t}\}\big)\end{array}$$

where the type name `t` must admit equality. In fact this is the only structure possible up to the choice of `m` and `t`. Note that the type function of `t`, `u` and `v` must be a type name, because of the `datatype` specification. Hence `t`, `u` and `v` all admit equality. The point is that equality is an attribute of the type function and so must be identical for all types that share. Constructor environments, on the other hand, can vary within the limits of consistency [Sec 5.2, p 32].

10.2 Yes, for example to $(\mathtt{m1},\{\mathtt{A}\mapsto(\mathtt{m2},\{\}),\mathtt{B}\mapsto(\mathtt{m2},\{\})\})$. Rule 79 [p 41] for `include` gives the necessary freedom to realise the bound names `m2` and `m3` by the same name (compare with the next exercise).

10.3 No; rule 78 [p 41] for `open` forces us to open `Str` precisely as it is (recall that real structures have rigid names), and certainly the names of `A` and `B` are different.

10.4 Interestingly, the construction of $\sim$ does not rely on the existence of an admissifier for A. However, there are the following places where checks are required to ensure that the definition of φ^* makes sense:

1. It is not necessarily the case that there is at most one $m_0 \in [m] \cap N$. An attempt to unify two different rigid structure names must be faulted.

2. It is not necessarily the case that there is at most one $\theta_0 \in [\theta]$ such that θ_0 is not both a type name and flexible. However, if θ_0 and θ_1 are such type functions, we still have $\mathrm{tynames}\,\theta_i \subseteq N$ $(i = 0, 1)$ since A is assumed grounded in N. Hence it is easy to check whether θ_0 and θ_1 are identical. Also, it must be checked that all members of $[\theta]$ have the same arity. Finally, in case θ_0 exists, it must be checked that if θ_0 does not admit equality, then no other member of the equivalence class admits equality.

Subject to the checks in (1) and (2), φ^* is well-defined and it is a realisation fixed on N. However, it may still not be an admissifier for A, for the following reasons:

1. In the absence of φ, we can no longer apply Theorem 9.1 to conclude that $\varphi^* A$ is well-formed and cycle-free. Indeed, it must be checked that φ^* neither destroys type structure well-formedness [Sec 4.9, p 21] nor introduces cycles. In particular, one must fault any attempt to unify two type structures (θ_1, CE_1) and (θ_2, CE_2) if θ_1 is not a type name and $\mathrm{Dom}\, CE_2$ is not empty.

2. The assembly $\varphi^* A$ will satisfy conditions 1 and 2 of consistency [Sec 5.2, p 32], but not necessarily condition 3. Any attempt to unify two type structures with different non-empty constructor environment domains must be faulted.

11.1 Yes. The equality-principal signature is

$$\{\mathtt{m}, \mathtt{t}\}(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{t}, \{\mathtt{C} \mapsto \mathtt{t}\}), \\ \mathtt{U} \mapsto (\mathtt{t}, \{\})\}, \{\mathtt{C} \mapsto \mathtt{t}\})$$

where `t` admits equality. The principal signature is identical to the above, except that `t` does not admit equality.

11.2 Yes, the equality-principal signature is

$$\{\mathtt{m}, \mathtt{t}, \mathtt{t'}\}(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{t'}, \{\}), \\ \mathtt{U} \mapsto (\mathtt{t}, \{\mathtt{C} \mapsto \mathtt{t}\})\}, \{\mathtt{C} \mapsto \mathtt{t}\})$$

where `t` admits equality and `t'` does not admit equality.

11.3 Yes, the equality-principal signature is

$$\{\mathtt{m}, \mathtt{t}, \mathtt{t'}\}(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{t'}, \{\}), \\ \mathtt{U} \mapsto (\mathtt{t}, \{\mathtt{C} \mapsto \mathtt{t'} \to \mathtt{t}\})\}, \{\mathtt{C} \mapsto \mathtt{t'} \to \mathtt{t}\})$$

where neither `t` nor `t'` admits equality. In step 1, `t'` and `t` are taken not to admit equality; the only type name which is candidate for change in step 3 is `t`, but since `t'` does not admit equality, `t` cannot admit equality.

11.4 Yes, the equality-principal signature is

$$\{\mathtt{m}, \mathtt{t}, \mathtt{t'}\}(\mathtt{m}, \{\mathtt{T} \mapsto (\mathtt{t'}, \{\}), \\ \mathtt{U} \mapsto (\mathtt{t}, \{\mathtt{C} \mapsto \mathtt{t'} \to \mathtt{t}\})\}, \{\mathtt{C} \mapsto \mathtt{t'} \to \mathtt{t}\})$$

where both `t` and `t'` admit equality. In step 1, `t'` is found to admit equality; in the principal signature, `t` does not admit equality, but in the equality-principal signature, `t` can be allowed to admit equality.

11.5 Yes. The equality-principal signature is

$\{\mathtt{m},\mathtt{t}\}$
$(\mathtt{m},\{\mathtt{T}\mapsto(\mathtt{t},\{\mathtt{C}\mapsto\mathtt{t}\}),\mathtt{U}\mapsto(\mathtt{t},\{\mathtt{C}\mapsto(\mathtt{int}\to\mathtt{int})\to\mathtt{t}\})\},$
$\{\mathtt{C}\mapsto(\mathtt{int}\to\mathtt{int})\to\mathtt{t}\})$

where `t` does not admit equality. This is one of the few examples of a legal signature which cannot be matched by any real structure.

11.6 No, the principal signature exists, but it does not respect equality.

11.7 Consider the signature expression

```
sig
  type t
  type u = t * t
end
```

This elaborates to all structures of the form
$(\mathtt{m},\{\mathtt{t}\mapsto(\Lambda().\tau,\{\}),$
$\mathtt{u}\mapsto(\Lambda().\tau*\tau,\{\})\})$

for which tynames $\tau\subseteq N$ of B. The only candidate for a principal signature, namely
$\{\mathtt{m},\mathtt{t}\}(\mathtt{m},\{\mathtt{t}\mapsto((\mathtt{t},\{\}),$
$\mathtt{u}\mapsto(\Lambda().\mathtt{t}*\mathtt{t},\{\})\})$

is not reachable by elaboration, since the type structure for `u` is not grounded in N of B.

11.8 In the two possible closed signatures, there will be in one case two *distinct* bound datatype names `t` and `u`, where `t` admits equality but not `u`, and in the other case a *single* bound datatype name which does not admit equality (corresponding to both `T` and `U`). Neither signature is an instance of the other.

D Appendix: Mistakes and Ambiguities

The following mistakes and ambiguities have been found in the Definition. Some are minor errors of wording and presentation. Others are technical points; for the more important of these we refer to the place in this Commentary where they are discussed.

The symbol $\longmapsto$ is used to signify replacement. 'Line -3' means the third line from bottom.

page

4 No space is allowed between the two characters which make up a comment bracket `(*` or `*)`. Even an unmatched `*)` should be detected by the compiler. Thus the expression `(op *)` is illegal. But `(op * )` is legal; so is `op*` .

5 Lines 14–17: The rule given for determining the status of an identifier is not sufficient. The same applies to [Sec 3.2,p 10]. Full rules are given in Appendix B.

12 At end of Sec 3.5 add:

- In the *tyvarseq tycon* in any *typdesc* or *datdesc*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring on the right side of the *datdesc* must occur in *tyvarseq*.

30 Lines 9–10 of [Sec 4.11]: principal type schemes $\longmapsto$ principal environments.
Line -5: $E \succ E' \longmapsto \mathrm{Clos}_C E \succ E'$ (see end of Section 5.3).
Line -3: delete "and imperative type variables" (see previous erratum).

31 Line -12: delete "imperative,".

35 Lines -5,-4: the claim that a principal signature exists must be slightly qualified, since it may be ill-formed in a mild sense. This is discussed at the end of Section 11.3.

42 Rule 86: $\mathrm{tyvars}(\tau) = \emptyset \longmapsto \mathrm{tyvars}(ty) = \emptyset$. This is needed to ensure that rule 86 is a structural contraction; see Section A.1.

44 Rule 99: $\mathrm{names}\, S' \backslash ((N \text{ of } B) \cup N) \longmapsto \mathrm{names}\, S' \langle ' \rangle \backslash ((N \text{ of } B) \cup N)$

46 [Sec 6.1], first bullet: exception bindings $\longmapsto$ constructor and exception bindings.
[Sec 6.1], second bullet: delete "or "`datatype` *datbind*" " (see Section 2.7).
[Sec 6.1], fourth bullet: delete "DatBind, ConBind,".

53 Add a new rule for datatype declarations after rule 129, and change rule 130 for exception declarations, as indicated in Section 2.7.

56 Rule 158 deals incorrectly with the case in which a program redeclares `ref` as a value constructor, since it will always interpret `ref` as a memory reference. Rule 114 [p 51] is similarly at fault in this case. For this reason, compilers may wish to issue a warning if `ref` is redeclared or specified as a value constructor.

67 Some of the derived forms of expressions [Fig 15], such as `()`, must be parsed as atomic expressions; they can be found under *atexp* in the full grammar [App B, Fig 19, p 71]. Similarly, the derived forms of patterns [Fig 16] must be parsed as atomic patterns; they all appear under *atpat* in [Fig 21, p 73].

Note that the meanings of certain derived forms [Fig 15 and 16] change if certain parts of the initial basis are overwritten. For example, the meaning of an `if` $\cdots$ `then` $\cdots$ `else` expression is affected by a rebinding of `true` or `false`; similarly, giving `it` constructor status changes the meaning of the derived form of expressions at top-level [Fig 18]. For this reason, compilers may wish to issue a warning if `true`, `false`, `nil` or `::` is redeclared or specified as a value constructor, exception constructor or variable, or if `it` is declared at top-level as a value constructor or exception constructor.

74 Line -7: (`true`,`false`,`nil`,`::`) $\longmapsto$ (`true`,`false`,`nil`,`::`,`ref`)

77 [App D], fifth bullet; this should read

- $VE'_0 = \{id \mapsto id \ ; \ id \in \text{BasVal}\} \cup \{\texttt{:=} \mapsto \texttt{:=}\} \cup EE'_0$
  $\cup$ {`true` $\mapsto$ `true`, `false` $\mapsto$ `false`, `nil` $\mapsto$ `nil`, `::` $\mapsto$ `::`, `ref` $\mapsto$ `ref`}

80 Line 13: after "initially empty." add

> Any existing contents of the file s are lost. The exception packet
>
> [(`Io`,`"Cannot open` s`"`)]
>
> is returned if write access to the file s is not provided.