Programming in Standard ML

(WORKING DRAFT OF FEBRUARY 13, 2009.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2005

Preface

This book is an introduction to programming with the Standard ML programming language. It began life as a set of lecture notes for Computer Science 15–212: Principles of Programming, the second semester of the introductory sequence in the undergraduate computer science curriculum at Carnegie Mellon University. It has subsequently been used in many other courses at Carnegie Mellon, and at a number of universities around the world. It is intended to supersede my *Introduction to Standard ML*, which has been widely circulated over the last ten years.

Standard ML is a formally defined programming language. *The Definition of Standard ML (Revised)* is the formal definition of the language. It is supplemented by the *Standard ML Basis Library*, which defines a common basis of types that are shared by all implementations of the language. *Commentary on Standard ML* discusses some of the decisions that went into the design of the first version of the language.

There are several implementations of Standard ML available for a wide variety of hardware and software platforms. The best-known compilers are *Standard ML of New Jersey*, *Moscow ML*, *MLKit*, and *PolyML*. These are all freely available on the worldwide web. Please refer to *The Standard ML Home Page* for up-to-date information on Standard ML and its implementations.

Readers at Carnegie Mellon are referred to the *CMU Local Guide* for information about using Standard ML.

Numerous people have contributed directly and indirectly to this text. I am especially grateful to the following people for their helpful comments and suggestions: Nels Beckman, Marc Bezem, James Bostock, Terrence Brannon, Franck van Breugel, Matthew William Cox, Karl Crary, Yaakov Eisenberg, Matt Elder, Mike Erdmann, Matthias Felleisen, Andrei Formiga, Stephen Harris, Joel Jones, David Koppstein, John Lafferty,

Flavio Lerda, Adrian Moos, Bryce Nichols, Michael Norrish, Arthur J. O'Dwyer, Frank Pfenning, Chris Stone, Dave Swasey, Michael Velten, Johan Wallen, Scott Williams, and Jeannette Wing. Richard C. Cobbe helped with font selection. I am also grateful to the many students of 15-212 who used these notes and sent in their suggestions over the years.

These notes are a work in progress. Corrections, comments and suggestions are most welcome.

Contents

Part I

Overview

Standard ML is a type-safe programming language that embodies many innovative ideas in programming language design. It is a statically typed language, with an extensible type system. It supports polymorphic type inference, which all but eliminates the burden of specifying types of variables and greatly facilitates code re-use. It provides efficient automatic storage management for data structures and functions. It encourages functional (effect-free) programming where appropriate, but allows imperative (effect-ful) programming where necessary. It facilitates programming with recursive and symbolic data structures by supporting the definition of functions by pattern matching. It features an extensible exception mechanism for handling error conditions and effecting non-local transfers of control. It provides a richly expressive and flexible module system for structuring large programs, including mechanisms for enforcing abstraction, imposing hierarchical structure, and building generic modules. It is portable across platforms and implementations because it has a precise definition. It provides a portable standard basis library that defines a rich collection of commonly-used types and routines.

Many implementations go beyond the standard to provide experimental language features, extensive libraries of commonly-used routines, and useful program development tools. Details can be found with the documentation for your compiler, but here's some of what you may expect. Most implementations provide an interactive system supporting on-line program development, including tools for compiling, linking, and analyzing the behavior of programs. A few implementations are "batch compilers" that rely on the ambient operating system to manage the construction of large programs from compiled parts. Nearly every compiler generates native machine code, even when used interactively, but some also generate code for a portable abstract machine. Most implementations support separate compilation and provide tools for managing large systems and shared libraries. Some implementations provide tools for tracing and stepping programs; many provide tools for time and space profiling. Most implementations supplement the standard basis library with a rich collection of handy components such as dictionaries, hash tables, or interfaces to the ambient operating system. Some implementations support language extensions such as support for concurrent programming (using message-passing or locking), richer forms of modularity constructs, and support for "lazy" data structures.

Chapter 1

Programming in Standard ML

1.1 A Regular Expression Package

To develop a feel for the language and how it is used, let us consider the implementation of a package for matching strings against regular expressions. We'll structure the implementation into two modules, an implementation of regular expressions themselves and an implementation of a matching algorithm for them.

These two modules are concisely described by the following *signatures*.

```
signature REGEXP = sig
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp |
    Times of regexp * regexp |
    Star of regexp
  exception SyntaxError of string
  val parse : string -> regexp
  val format : regexp -> string
end
signature MATCHER = sig
  structure RegExp : REGEXP
  val match : RegExp.regexp -> string -> bool
end
```

The signature `REGEXP` describes a module that implements regular expressions. It consists of a description of the abstract syntax of regular expres-

sions, together with operations for parsing and unparsing them. The signature `MATCHER` describes a module that implements a matcher for a given notion of regular expression. It contains a function `match` that, when given a regular expression, returns a function that determines whether or not a given string matches that expression. Obviously the matcher is dependent on the implementation of regular expressions. This is expressed by a *structure specification* that specifies a hierarchical dependence of an implementation of a matcher on an implementation of regular expressions — any implementation of the `MATCHER` signature must include an implementation of regular expressions as a constituent module. This ensures that the matcher is self-contained, and does not rely on implicit conventions for determining *which* implementation of regular expressions it employs.

The definition of the abstract syntax of regular expressions in the signature `REGEXP` takes the form of a *datatype declaration* that is reminiscent of a context-free grammar, but which abstracts from matters of lexical presentation (such as precedences of operators, parenthesization, conventions for naming the operators, *etc.*.) The abstract syntax consists of six clauses, corresponding to the regular expressions **0**, **1**, **a**, $r_1 + r_2$, $r_1\, r_2$, and r^*.[1] The functions `parse` and `format` specify the parser and unparser for regular expressions. The parser takes a string as argument and yields a regular expression; if the string is ill-formed, the parser raises the exception SyntaxError with an associated string describing the source of the error. The unparser takes a regular expression and yields a string that parses to that regular expression. In general there are many strings that parse to the same regular expressions; the unparser generally tries to choose one that is easiest to read.

The implementation of the matcher consists of two modules: an implementation of regular expressions and an implementation of the matcher itself. An implementation of a signature is called a *structure*. The implementation of the matching package consists of two structures, one implementing the signature `REGEXP`, the other implementing `MATCHER`. Thus the overall package is implemented by the following two *structure declarations*:

```
structure RegExp :> REGEXP = ...
structure Matcher :> MATCHER = ...
```

The structure identifier `RegExp` is bound to an implementation of the `REGEXP`

[1] Some authors use ∅ for **0** and **″** for **1**.

signature. Conformance with the signature is ensured by the *ascription* of the signature `REGEXP` to the binding of `RegExp` using the "`:>`" notation. Not only does this check that the implementation (which has been elided here) conforms with the requirements of the signature `REGEXP`, but it also ensures that subsequent code cannot rely on any properties of the implementation other than those explicitly specified in the signature. This helps to ensure that modules are kept separate, facilitating subsequent changes to the code.

Similarly, the structure identifier `Matcher` is bound to a structure that implements the matching algorithm in terms of the preceding implementation `RegExp` of `REGEXP`. The ascribed signature specifies that the structure `Matcher` must conform to the requirements of the signature `MATCHER`. Notice that the structure `Matcher` refers to the structure `RegExp` in its implementation.

Once these structure declarations have been processed, we may use the package by referring to its components using *paths*, or *long identifiers*. The function `Matcher.match` has type

```
Matcher.RegExp.regexp -> string -> bool,
```

reflecting the fact that it takes a regular expression *as implemented within the package itself* and yields a matching function on strings. We may build a regular expression by applying the parser, `Matcher.RegExp.parse`, to a string representing a regular expression, then passing the resulting value of type `Matcher.RegExp.regexp` to `Matcher.match`.[2]

Here's an example of the matcher in action:

```
val regexp =
    Matcher.RegExp.parse "(a+b)*"
val matches =
    Matcher.match regexp
val ex1 = matches "aabba"     (* yields true *)
val ex2 = matches "abac"      (* yields false *)
```

[2]It might seem that one can apply `Matcher.match` to the output of `RegExp.parse`, since `Matcher.RegExp.parse` is just `RegExp.parse`. However, this relationship is not stated in the interface, so there is a *pro forma* distinction between the two. See Chapter 22 for more information on the subtle issue of *sharing*.

The use of long identifiers can get tedious at times. There are two typical methods for alleviating the burden. One is to introduce a synonym for a long package name. Here's an example:

```
structure M = Matcher
structure R = M.RegExp
val regexp = R.parse "((a + %).(b + %))*"
val matches = M.match regexp
val ex1 = matches "aabba"
val ex2 = matches "abac"
```

Another is to "open" the structure, incorporating its bindings into the current environment:

```
open Matcher Matcher.RegExp
val regexp = parse "(a+b)*"
val matches = match regexp
val ex1 = matches "aabba"
val ex2 = matches "abac"
```

It is advisable to be sparing in the use of `open` because it is often hard to anticipate exactly which bindings are incorporated into the environment by its use.

Now let's look at the internals of the structures `RegExp` and `Matcher`. Here's a bird's eye view of `RegExp`:

```
structure RegExp :> REGEXP = struct
  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp |
    Times of regexp * regexp |
    Star of regexp
  ⋮
  fun tokenize s = ...
  ⋮
  fun parse s =
      let
          val (r, s') =
```

```
            parse_rexp (tokenize (String.explode s))
      in
          case s' of
            nil => r
          | _ => raise SyntaxError "Bad input."
      end
      handle LexicalError =>
        raise SyntaxError "Bad input."
  ⋮
  fun format r =
      String.implode (format_exp r)
end
```

The elision indicates that portions of the code have been omitted so that we can get a high-level view of the structure of the implementation.

The structure `RegExp` is bracketed by the keywords `struct` and `end`. The type `regexp` is implemented precisely as specified by the `datatype` declaration in the signature `REGEXP`. The parser is implemented by a function that, when given a string, "explodes" it into a list of characters, transforms the character list into a list of "tokens" (abstract symbols representing lexical atoms), and finally parses the resulting list of tokens to obtain its abstract syntax. If there is remaining input after the parse, or if the tokenizer encountered an illegal token, an appropriate syntax error is signalled. The formatter is implemented by a function that, when given a piece of abstract syntax, formats it into a list of characters that are then "imploded" to form a string. The parser and formatter work with character lists, rather than strings, because it is easier to process lists incrementally than it is to process strings.

It is interesting to consider in more detail the structure of the parser since it exemplifies well the use of pattern matching to define functions. Let's start with the tokenizer, which we present here *in toto*:

```
datatype token =
  AtSign | Percent | Literal of char |
  PlusSign | TimesSign |
  Asterisk | LParen | RParen
exception LexicalError
fun tokenize nil = nil
```

```
  | tokenize (#"+" :: cs) = PlusSign :: tokenize cs
  | tokenize (#"." :: cs) = TimesSign :: tokenize cs
  | tokenize (#"*" :: cs) = Asterisk :: tokenize cs
  | tokenize (#"(" :: cs) = LParen :: tokenize cs
  | tokenize (#")" :: cs) = RParen :: tokenize cs
  | tokenize (#"@" :: cs) = AtSign :: tokenize cs
  | tokenize (#"%" :: cs) = Percent :: tokenize cs
  | tokenize (#"\\" :: c :: cs) =
    Literal c :: tokenize cs
  | tokenize (#"\\" :: nil) = raise LexicalError
  | tokenize (#" " :: cs) = tokenize cs
  | tokenize (c :: cs) = Literal c :: tokenize cs
```

The symbol "`@`" stands for the empty regular expression and the symbol "`%`" stands for the regular expression accepting only the null string. Concatentation is indicated by "`.`", alternation by "`+`", and iteration by "`*`".

We use a datatype declaration to introduce the type of tokens corresponding to the symbols of the input language. The function `tokenize` has type `char list -> token list`; it transforms a list of characters into a list of tokens. It is defined by a series of clauses that dispatch on the first character of the list of characters given as input, yielding a list of tokens. The correspondence between characters and tokens is relatively straightforward, the only non-trivial case being to admit the use of a backslash to "quote" a reserved symbol as a character of input. (More sophisticated languages have more sophisticated token structures; for example, words (consecutive sequences of letters) are often regarded as a single token of input.) Notice that it is quite natural to "look ahead" in the input stream in the case of the backslash character, using a pattern that dispatches on the first two characters (if there are such) of the input, and proceeding accordingly. (It is a lexical error to have a backslash at the end of the input.)

Let's turn to the parser. It is a simple recursive-descent parser implementing the precedence conventions for regular expressions given earlier. These conventions may be formally specified by the following grammar, which not only enforces precedence conventions, but also allows for the

use of parenthesization to override them.

$$\begin{array}{rcl} \mathit{rexp} & ::= & \mathit{rtrm} \mid \mathit{rtrm}\texttt{+}\mathit{rexp} \\ \mathit{rtrm} & ::= & \mathit{rfac} \mid \mathit{rfac}\,.\,\mathit{rtrm} \\ \mathit{rfac} & ::= & \mathit{ratm} \mid \mathit{ratm}\texttt{*} \\ \mathit{ratm} & ::= & \texttt{@} \mid \texttt{\%} \mid a \mid \texttt{(}\mathit{rexp}\texttt{)} \end{array}$$

The implementation of the parser follows this grammar quite closely. It consists of four mutually recursive functions, `parse_rexp`, `parse_rtrm`, `parse_rfac`, and `parse_ratm`. These implement what is known as a *recursive descent* parser that dispatches on the head of the token list to determine how to proceed.

```
fun parse_rexp ts =
    let
        val (r, ts') = parse_rtrm ts
    in
        case ts'
          of (PlusSign :: ts'') =>
             let
                 val (r', ts''') = parse_rexp ts''
             in
                (Plus (r, r'), ts''')
             end
           | _ => (r, ts')
    end
and parse_rtrm ts = ...
and parse_rfac ts =
    let
        val (r, ts') = parse_ratm ts
    in
        case ts'
        of (Asterisk :: ts'') => (Star r, ts'')
         | _ => (r, ts')
    end
and parse_ratm nil =
      raise SyntaxError ("No atom")
  | parse_ratm (AtSign :: ts) = (Zero, ts)
  | parse_ratm (Percent :: ts) = (One, ts)
```

```
  | parse_ratm ((Literal c) :: ts) = (Char c, ts)
  | parse_ratm (LParen :: ts) =
    let
        val (r, ts') = parse_rexp ts
    in
        case ts'
          of (RParen :: ts'') => (r, ts'')
           | _ =>
             raise SyntaxError "No close paren"
    end
```

Notice that it is quite simple to implement "lookahead" using patterns that inspect the token list for specified tokens. This parser makes no attempt to recover from syntax errors, but one could imagine doing so, using standard techniques.

This completes the implementation of regular expressions. Now for the matcher. The matcher proceeds by a recursive analysis of the regular expression. The main difficulty is to account for concatenation — to match a string against the regular expression $r_1\, r_2$ we must match some initial segment against r_1, then match the corresponding final segment against r_2. This suggests that we generalize the matcher to one that checks whether some initial segment of a string matches a given regular expression, then passes the remaining final segment to a *continuation*, a function that determines what to do after the initial segment has been successfully matched. This facilitates implementation of concatentation, but how do we ensure that at the outermost call the entire string has been matched? We achieve this by using an *initial continuation* that checks whether the final segment is empty.

Here's the code, written as a structure implementing the signature `MATCHER`:

```
structure Matcher :> MATCHER =
struct
  structure RegExp = RegExp
  open RegExp
  fun match_is Zero _ k = false
    | match_is One cs k = k cs
    | match_is (Char c) nil _ = false
    | match_is (Char c) (d::cs) k = (c=d) andalso (k cs)
```

```
    | match_is (Plus (r1, r2)) cs k =
      match_is r1 cs k  orelse  match_is r2 cs k
    | match_is (Times (r1, r2)) cs k =
      match_is r1 cs (fn cs' => match_is r2 cs' k)
    | match_is (Star r) cs k =
      k cs  orelse
      match_is r cs (fn cs' => match_is (Star r) cs' k)
  fun match regexp string =
      match_is
       regexp
       (String.explode string)
       (fn nil => true | _ => false)
end
```

Note that we incorporate the structure `RegExp` into the structure `Matcher`, in accordance with the requirements of the signature. The function `match` explodes the string into a list of characters (to facilitiate sequential processing of the input), then calls `match_is` with an initial continuation that ensures that the remaining input is empty to determine the result. The type of `match_is` is

```
RegExp.regexp -> char list ->
  (char list -> bool) -> bool.
```

That is, `match_is` takes in succession a regular expression, a list of characters, and a continuation of type `char list -> bool`; it yields as result a value of type `bool`. This is a fairly complicated type, but notice that nowhere did we have to write this type in the code! The type inference mechanism of ML took care of determining what that type must be based on an analysis of the code itself.

Since `match_is` takes a function as argument, it is said to be a *higher-order function*. The simplicity of the matcher is due in large measure to the ease with which we can manipulate functions in ML. Notice that we create a new, unnamed function to pass as a continuation in the case of concatenation — it is the function that matches the second part of the regular expression to the characters remaining after matching an initial segment against the first part. We use a similar technique to implement matching against an iterated regular expression — we attempt to match the null

string, but if this fails, we match against the regular expression being iterated followed by the iteration once again. This neatly captures the "zero or more times" interpretation of iteration of a regular expression.

> *Important*: the code given above contains a subtle error. Can you find it? If not, see chapter 27 for further discussion!

This completes our brief overview of Standard ML. The remainder of these notes are structured into three parts. The first part is a detailed introduction to the core language, the language in which we write programs in ML. The second part is concerned with the module language, the means by which we structure large programs in ML. The third is about programming techniques, methods for building reliable and robust programs. I hope you enjoy it!

1.2 Sample Code

Here is the complete code for this chapter.

Part II

The Core Language

All Standard ML is divided into two parts. The first part, the *core language*, comprises the fundamental programming constructs of the language — the primitive types and operations, the means of defining and using functions, mechanisms for defininining new types, and so on. The second part, the *module language*, comprises the mechanisms for structuring programs into separate units and is described in Part III. Here we introduce the core language.

Chapter 2

Types, Values, and Effects

2.1 Evaluation and Execution

Most familiar programming languages, such as C or Java, are based on an *imperative* model of computation. Programs are thought of as specifying a sequence of *commands* that modify the memory of the computer. Each step of execution examines the current contents of memory, performs a simple computation, modifies the memory, and continues with the next instruction. The individual commands are executed for their *effect* on the memory (which we may take to include both the internal memory and registers and the external input/output devices). The progress of the computation is controlled by evaluation of expressions, such as boolean tests or arithmetic operations, that are executed for their value. Conditional commands branch according to the value of some expression. Many languages maintain a distinction between expressions and commands, but often (in C, for example) expressions may also modify the memory, so that even expression evaluation has an effect.

Computation in ML is of a somewhat different nature. The emphasis in ML is on computation by *evaluation of expressions*, rather than *execution of commands*. The idea of computation is as a generalization of your experience from high school algebra in which you are given a polynomial in a variable x and are asked to calculate its value at a given value of x. We proceed by "plugging in" the given value for x, and then, using the rules of arithmetic, determine the value of the polynomial. The evaluation model of computation used in ML is based on the same idea, but rather than re-

strict ourselves to arithmetic operations on the reals, we admit a richer variety of values and a richer variety of primitive operations on them.

The evaluation model of computation enjoys several advantages over the more familiar imperative model. Because of its close relationship to mathematics, it is much easier to develop mathematical techniques for reasoning about the behavior of programs. These techniques are important tools for helping us to ensure that programs work properly without having to resort to tedious testing and debugging that can only show the presence of errors, never their absence. Moreover, they provide important tools for documenting the reasoning that went into the formulation of a program, making the code easier to understand and maintain.

What is more, the evaluation model subsumes the imperative model as a special case. Execution of commands for the effect on memory can be seen as a special case of evaluation of expressions by introducing primitive operations for allocating, accessing, and modifying memory. Rather than forcing all aspects of computation into the framework of memory modification, we instead take expression evaluation as the primary notion. Doing so allows us to support imperative programming without destroying the mathematical elegance of the evaluation model for programs that don't use memory. As we will see, it is quite remarkable how seldom memory modification is required. Nevertheless, the language provides for storage-based computation for those few times that it is actually necessary.

2.2 The ML Computation Model

Computation in ML consists of evaluation of expressions. Each expression has three important characteristics:

- It may or may not have a *type*.
- It may or may not have a *value*.
- It may or may not engender an *effect*.

These characteristics are all that you need to know to compute with an expression.

The type of an expression is a description of the value it yields, should it yield a value at all. For example, for an expression to have type `int` is to

say that its value (should it have one) is a number, and for an expression to have type `real` is to say that its value (if any) is a floating point number. In general we can think of the type of an expression as a "prediction" of the form of the value that it has, should it have one. Every expression is required to have at least one type; those that do are said to be *well-typed*. Those without a type are said to be *ill-typed*; they are considered ineligible for evaluation. The *type checker* determines whether or not an expression is well-typed, rejecting with an error those that are not.

A well-typed expression is evaluated to determine its value, if indeed it has one. An expression can fail to have a value because its evaluation never terminates or because it raises an exception, either because of a run-time fault such as division by zero or because some programmer-defined condition is signalled during its evaluation. If an expression has a value, the form of that value is predicted by its type. For example, if an expression evaluates to a value v and its type is `bool`, then v must be either `true` or `false`; it cannot be, say, `17` or `3.14`. The *soundness* of the type system ensures the accuracy of the predictions made by the type checker.

Evaluation of an expression might also engender an *effect*. Effects include such phenomena as raising an exception, modifying memory, performing input or output, or sending a message on the network. It is important to note that the type of an expression says nothing about its possible effects! An expression of type `int` might well display a message on the screen before returning an integer value. This possibility is not accounted for in the type of the expression, which classifies only its value. For this reason effects are sometimes called *side effects*, to stress that they happen "off to the side" during evaluation, and are not part of the value of the expression. We will ignore effects until chapter 13. For the time being we will assume that all expressions are *effect-free*, or *pure*.

2.2.1 Type Checking

What is a type? What types are there? Generally speaking, a type is defined by specifying three things:

- a *name* for the type,
- the *values* of the type, and
- the *operations* that may be performed on values of the type.

Often the division of labor into values and operations is not completely clear-cut, but it nevertheless serves as a very useful guideline for describing types.

Let's consider first the type of integers. Its name is `int`. The values of type `int` are the numerals `0`, `1`, `~1`, `2`, `~2`, and so on. (Note that negative numbers are written with a prefix tilde, rather than a minus sign!) Operations on integers include addition, `+`, subtraction, `-`, multiplication, `*`, quotient, `div`, and remainder, `mod`. Arithmetic expressions are formed in the familiar manner, for example, `3*2+6`, governed by the usual rules of precedence. Parentheses may be used to override the precedence conventions, just as in ordinary mathematical practice. Thus the preceding expression may be equivalently written as `(3*2)+6`, but we may also write `3*(2+6)` to override the default precedences.

The formation of expressions is governed by a set of *typing rules* that define the types of expressions in terms of the types of their constituent expressions (if any). The typing rules are generally quite intuitive since they are consistent with our experience in mathematics and in other programming languages. In their full generality the rules are somewhat involved, but we will sneak up on them by first considering only a small fragment of the language, building up additional machinery as we go along.

Here are some simple arithmetic expressions, written using *infix* notation for the operations (meaning that the operator comes between the arguments, as is customary in mathematics):

```
3
3 + 4
4 div 3
4 mod 3
```

Each of these expressions is well-formed; in fact, they each have type `int`. This is indicated by a *typing assertion* of the form *exp* : *typ*, which states that the expression *exp* has the type *typ*. A typing assertion is said to be *valid* iff the expression *exp* does indeed have the type *typ*. The following are all valid typing assertions:

```
3 : int
3 + 4 : int
4 div 3 : int
4 mod 3 : int
```

Why are these typing assertions valid? In the case of the value `3`, it is an axiom that integer numerals have integer type. What about the expression `3+4`? The addition operation takes two arguments, each of which must have type `int`. Since both arguments in fact have type `int`, it follows that the entire expression is of type `int`. For more complex cases we reason analogously, for example, deducing that `(3+4) div (2+3): int` by observing that `(3+4): int` and `(2+3): int`.

The reasoning involved in demonstrating the validity of a typing assertion may be summarized by a *typing derivation* consisting of a nested sequence of typing assertions, each justified either by an axiom, or a typing rule for an operation. For example, the validity of the typing assertion `(3+7) div 5 : int` is justified by the following derivation:

1. `(3+7): int`, because
   (a) `3 : int` because it is an axiom
   (b) `7 : int` because it is an axiom
   (c) the arguments of `+` must be integers, and the result of `+` is an integer
2. `5 : int` because it is an axiom
3. the arguments of `div` must be integers, and the result is an integer

The outermost steps justify the assertion `(3+4) div 5 : int` by demonstrating that the arguments each have type `int`. Recursively, the inner steps justify that `(3+4): int`.

2.2.2 Evaluation

Evaluation of expressions is defined by a set of *evaluation rules* that determine how the value of a compound expression is determined as a function of the values of its constituent expressions (if any). Since the value of an operator is determined by the values of its arguments, ML is sometimes said to be a *call-by-value* language. While this may seem like the only sensible way to define evaluation, we will see in chapter 15 that this need not be the case — some operations may yield a value *without* evaluating their arguments. Such operations are sometimes said to be *lazy*, to distinguish

them from *eager* operations that require their arguments to be evaluated before the operation is performed.

An *evaluation assertion* has the form *exp*⇓*val*. This assertion states that the expression *exp* has value *val*. It should be intuitively clear that the following evaluation assertions are valid.

```
5 ⇓ 5
2+3 ⇓ 5
(2+3) div (1+4) ⇓ 1
```

An evaluation assertion may be justified by an *evaluation derivation*, which is similar to a typing derivation. For example, we may justify the assertion `(3+7) div 5` ⇓ `2` by the derivation

1. `(3+7)` ⇓ `10` because
   (a) `3` ⇓ `3` because it is an axiom
   (b) `7` ⇓ `7` because it is an axiom
   (c) Adding `3` to `7` yields `10`.
2. `5` ⇓ `5` because it is an axiom
3. Dividing `10` by `5` yields `2`.

Note that is an axiom that a numeral evaluates to itself; numerals are fully-evaluated expressions, or *values*. Second, the rules of arithmetic are used to determine that adding `3` and `7` yields `10`.

Not every expression has a value. A simple example is the expression `5 div 0`, which is undefined. If you attempt to evaluate this expression it will incur a run-time error, reflecting the erroneous attempt to find the number *n* that, when multiplied by 0, yields 5. The error is expressed in ML by raising an *exception*; we will have more to say about exceptions in chapter 12. Another reason that a well-typed expression might not have a value is that the attempt to evaluate it leads to an infinite loop. We don't yet have the machinery in place to define such expressions, but we will soon see that it is possible for an expression to *diverge*, or run forever, when evaluated.

2.3 Types, Types, Types

What types are there besides the integers? Here are a few useful *base* types of ML:

- *Type name*: `real`
  - *Values*: `3.14, 2.17, 0.1E6, ...`
  - *Operations*: `+, -, *, /, =, <, ...`
- *Type name*: `char`
  - *Values*: `#"a", #"b", ...`
  - *Operations*: `ord,chr,=, <, ...`
- *Type name*: `string`
  - *Values*: `"abc", "1234", ...`
  - *Operations*: `^, size, =, <, ...`
- *Type name*: `bool`
  - *Values*: `true, false`
  - *Operations*: `if` *exp* `then` exp_1 `else` exp_2

There are many, many (in fact, infinitely many!) others, but these are enough to get us started. (See V for a complete description of the primitive types of ML, including the ones given above.)

Notice that some of the arithmetic operations for real numbers are written the same way as for the corresponding operation on integers. For example, we may write `3.1+2.7` to perform a floating point addition of two floating point numbers. This is called *overloading*; the addition operation is said to be *overloaded* at the types `int` and `real`. In an expression involving addition the type checker tries to resolve which form of addition (fixed point or floating point) you mean. If the arguments are `int`'s, then fixed point addition is used; if the arguments are `real`'s, then floating addition is used; otherwise an error is reported.[1] Note that ML does *not* perform any implicit conversions between types! For example, the expression

[1] If the type of the arguments cannot be determined, the type defaults to `int`.

`3+3.14` is rejected as ill-formed! If you intend floating point addition, you must write instead `real(3)+3.14`, which converts the integer `3` to its floating point representation before performing the addition. If, on the other hand, you intend integer addition, you must write `3+round(3.14)`, which converts `3.14` to an integer by rounding before performing the addition.

Finally, note that floating point division is a *different* operation from integer quotient! Thus we write `3.1/2.7` for the result of dividing `3.1` by `2.7`, which results in a floating point number. We reserve the operator `div` for integers, and use `/` for floating point division.

The *conditional expression*

`if` *exp* `then` exp_1 `else` exp_2

is used to discriminate on a Boolean value. It has type *typ* if *exp* has type `bool` and both exp_1 and exp_2 have type *typ*. Notice that both "arms" of the conditional must have the same type! It is evaluated by first evaluating *exp*, then proceeding to evaluate either exp_1 or exp_2, according to whether the value of *exp* is `true` or `false`. For example,

```
if 1<2 then "less" else "greater"
```

evaluates to `"less"` since the value of the expression `1<2` is true.

Note that the expression

```
if 1<2 then 0 else (1 div 0)
```

evaluates to `0`, even though `1 div 0` incurs a run-time error. This is because evaluation of the conditional proceeds *either* to the `then` clause *or* to the `else` clause, depending on the outcome of the boolean test. Whichever clause is evaluated, the other is simply discarded without further consideration.

Although we may, in fact, test equality of two boolean expressions, it is rarely useful to do so. Beginners often writen conditionals of the form

`if` *exp* `= true then` exp_1 `else` exp_2`.`

But this is equivalent to the simpler expression

`if` *exp* `then` exp_1 `else` exp_2`.`

Similarly, rather than write

`if` *exp* `= false then` exp_1 `else` exp_2,

it is better to write

`if not` *exp* `then` exp_1 `else` exp_2

or, better yet, just

`if` *exp* `then` exp_2 `else` exp_1.

2.4 Type Errors

Now that we have more than one type, we have enough rope to hang ourselves by forming *ill-typed* expressions. For example, the following expressions are not well-typed:

```
size 45
#"1" + 1
#"2" ^ "1"
3.14 + 2
```

In each case we are "misusing" an operator with arguments of the wrong type.

This raises a natural question: is the following expression well-typed or not?

```
if 1<2 then 0 else ("abc"+4)
```

Since the boolean test will come out `true`, the `else` clause will never be executed, and hence need not be constrained to be well-typed. While this reasoning is sensible for such a simple example, in general it is impossible for the type checker to determine the outcome of the boolean test during type checking. To be safe the type checker "assumes the worst" and insists that both clauses of the conditional be well-typed, and in fact have the *same* type, to ensure that the conditional expression can be given a type, namely that of both of its clauses.

2.5 Sample Code

Here is the complete code for this chapter.

Chapter 3

Declarations

3.1 Variables

Just as in any other programming language, values may be assigned to variables, which may then be used in expressions to stand for that value. However, in sharp contrast to most familiar languages, *variables in ML do not vary*! A value may be *bound* to a variable using a construct called a *value binding*. Once a variable is bound to a value, it is bound to it for life; there is no possibility of changing the binding of a variable once it has been bound. In this respect variables in ML are more akin to variables in mathematics than to variables in languages such as C.

A type may also be bound to a *type constructor* using a *type binding*. A bound type constructor stands for the type bound to it, and can never stand for any other type. For this reason a type binding is sometimes called a *type abbreviation* — the type constructor stands for the type to which it is bound.[1]

A value or type binding introduces a "new" variable or type constructor, distinct from all others of that class, for use within its range of significance, or *scope*. Scoping in ML is *static*, or *lexical*, meaning that the range of significance of a variable or type constructor is determined by the program text, not by the order of evaluation of its constituent expressions. (Languages with *dynamic* scope adopt the opposite convention.) For the time being variables and type constructors have *global scope*, meaning that

[1] By the same token a value binding might also be called a *value abbreviation*, but for some reason it never is.

the range of significance of the variable or type constructor is the "rest" of the program — the part that lexically follows the binding. However, we will soon introduce mechanisms for limiting the scopes of variables or type constructors to a given expression.

3.2 Basic Bindings

3.2.1 Type Bindings

Any type may be given a name using a *type binding*. At this stage we have so few types that it is hard to justify binding type names to identifiers, but we'll do it anyway because we'll need it later. Here are some examples of type bindings:

```
type float = real
type count = int and average = real
```

The first type binding introduces the type constructor `float`, which subsequently is synonymous with `real`. The second introduces *two* type constructors, `count` and `average`, which stand for `int` and `real`, respectively.

In general a type binding introduces one or more new type constructors *simultaneously* in the sense that the definitions of the type constructors may not involve any of the type constructors being defined. Thus a binding such as

```
type float = real and average = float
```

is nonsensical (in isolation) since the type constructors `float` and `average` are introduced simultaneously, and hence cannot refer to one another.

The syntax for type bindings is

`type` $tycon_1$ `=` typ_1
`and` ...
`and` $tycon_n$ `=` typ_n

where each $tycon_i$ is a type constructor and each typ_i is a type expression.

3.2.2 Value Bindings

A value may be given a name using a *value binding*. Here are some examples:

```
val m : int = 3+2
val pi : real = 3.14 and e : real = 2.17
```

The first binding introduces the variable `m`, specifying its type to be `int` and its value to be `5`. The second introduces two variables, `pi` and `e`, simultaneously, both having type `real`, and with `pi` having value `3.14` and `e` having value `2.17`. Notice that a value binding specifies both the type and the value of a variable.

The syntax of value bindings is

`val` var_1 `:` typ_1 `=` exp_1
`and ...`
`and` var_n `:` typ_n `=` exp_n,

where each var_i is a variable, each typ_i is a type expression, and each exp_i is an expression.

A value binding of the form

`val` *var* `:` *typ* `=` *exp*

is type-checked by ensuring that the expression *exp* has type *typ*. If not, the binding is rejected as ill-formed. If so, the binding is evaluated using the *bind-by-value* rule: first *exp* is evaluated to obtain its value *val*, then *val* is bound to *var*. If *exp* does not have a value, then the declaration does not bind anything to the variable *var*.

The purpose of a binding is to make a variable available for use within its scope. In the case of a type binding we may use the type variable introduced by that binding in type expressions occurring within its scope. For example, in the presence of the type bindings above, we may write

```
val pi : float = 3.14
```

since the type constructor `float` is bound to the type `real`, the type of the expression `3.14`. Similarly, we may make use of the variable introduced by a value binding in value expressions occurring within its scope.

Continuing from the preceding binding, we may use the expression

```
sin pi
```

to stand for `0.0` (approximately), and we may bind this value to a variable by writing

```
val x : float = sin pi
```

As these examples illustrate, type checking and evaluation are *context dependent* in the presence of type and value bindings since we must refer to these bindings to determine the types and values of expressions. For example, to determine that the above binding for `x` is well-formed, we must consult the binding for `pi` to determine that it has type `float`, consult the binding for `float` to determine that it is synonymous with `real`, which is necessary for the binding of `x` to have type `float`.

The rough-and-ready rule for both type-checking and evaluation is that a bound variable or type constructor is implicitly *replaced* by its binding prior to type checking and evaluation. This is sometimes called the *substitution principle* for bindings. For example, to evaluate the expression `cos x` in the scope of the above declarations, we first replace the occurrence of `x` by its value (approximately `0.0`), then compute as before, yielding (approximately) `1.0`. Later on we will have to refine this simple principle to take account of more sophisticated language features, but it is useful nonetheless to keep this simple idea in mind.

3.3 Compound Declarations

Bindings may be combined to form *declarations*. A binding is an atomic declaration, even though it may introduce many variables simultaneously. Two declarations may be combined by *sequential composition* by simply writing them one after the other, optionally separated by a semicolon. Thus we may write the declaration

```
val m : int = 3+2
val n : int = m*m
```

which binds `m` to `5` and `n` to `25`. Subsequently, we may evaluate `m+n` to obtain the value `30`. In general a sequential composition of declarations has the form $dec_1 \ldots dec_n$, where n is at least 2. The scopes of these declarations

are *nested* within one another: the scope of dec_1 includes $dec_2, \ldots, dec_n$, the scope of dec_2 includes $dec_3, \ldots, dec_n$, and so on.

One thing to keep in mind is that *binding is not assignment*. The binding of a variable never changes; once bound to a value, it is always bound to that value (within the scope of the binding). However, we may *shadow* a binding by introducing a second binding for a variable within the scope of the first binding. Continuing the above example, we may write

```
val n : real = 2.17
```

to introduce a new variable `n` with both a different type and a different value than the earlier binding. The new binding eclipses the old one, which may then be discarded since it is no longer accessible. (Later on, we will see that in the presence of higher-order functions shadowed bindings are not always discarded, but are preserved as private data in a closure. One might say that old bindings never die, they just fade away.)

3.4 Limiting Scope

The scope of a variable or type constructor may be delimited by using `let` expressions and `local` declarations. A `let` expression has the form

`let` *dec* `in` *exp* `end`

where *dec* is any declaration and *exp* is any expression. The scope of the declaration *dec* is limited to the expression *exp*. The bindings introduced by *dec* are discarded upon completion of evaluation of *exp*.

Similarly, we may limit the scope of one declaration to another declaration by writing

`local` *dec* `in` dec' `end`

The scope of the bindings in *dec* is limited to the declaration dec'. After processing dec', the bindings in *dec* may be discarded.

The value of a `let` expression is determined by evaluating the declaration part, then evaluating the expression relative to the bindings introduced by the declaration, yielding this value as the overall value of the `let` expression. An example will help clarify the idea:

```
let
    val m : int = 3
    val n : int = m*m
in
    m*n
end
```

This expression has type `int` and value `27`, as you can readily verify by first calculating the bindings for `m` and `n`, then computing the value of `m*n` relative to these bindings. The bindings for `m` and `n` are local to the expression `m*n`, and are not accessible from outside the expression.

If the declaration part of a `let` expression eclipses earlier bindings, the ambient bindings are restored upon completion of evaluation of the `let` expression. Thus the following expression evaluates to `54`:

```
val m : int = 2
val r : int =
    let
        val m : int = 3
        val n : int = m*m
    in
        m*n
    end * m
```

The binding of `m` is temporarily overridden during the evaluation of the `let` expression, then restored upon completion of this evaluation.

3.5 Typing and Evaluation

To complete this chapter, let's consider in more detail the context-sensitivity of type checking and evaluation in the presence of bindings. The key ideas are:

- Type checking must take account of the declared type of a variable.
- Evaluation must take account of the declared value of a variable.

This is achieved by maintaining *environments* for type checking and evaluation. The *type environment* records the types of variables; the *value*

environment records their values. For example, after processing the compound declaration

```
val m : int = 0
val x : real = Math.sqrt(2.0)
val c : char = #"a"
```

the type environment contains the information

```
val m : int
val x : real
val c : char
```

and the value environment contains the information

```
val m = 0
val x = 1.414
val c = #"a"
```

In a sense the value declarations have been divided in "half", separating the type from the value information.

Thus we see that value bindings have significance for both type checking and evaluation. In contrast type bindings have significance only for type checking, and hence contribute only to the type environment. A type binding such as

```
type float = real
```

is recorded in its entirety in the type environment, and no change is made to the value environment. Subsequently, whenever we encounter the type constructor `float` in a type expression, it is replaced by `real` in accordance with the type binding above.

In chapter 2 we said that a typing assertion has the form $exp : typ$, and that an evaluation assertion has the form $exp \Downarrow val$. While two-place typing and evaluation assertions are sufficient for *closed* expressions (those without variables), we must extend these relations to account for *open* expressions (those with variables). Each must be equipped with an *environment* recording information about type constructors and variables introduced by declarations.

Typing assertions are generalized to have the form

typenv $\vdash$ *exp* : *typ*

where *typenv* is a *type environment* that records the bindings of type constructors and the types of variables that may occur in *exp*.[2] We may think of *typenv* as a sequence of specifications of one of the following two forms:

1. `type` *typvar* `=` *typ*
2. `val` *var* : *typ*

Note that the second form does *not* include the binding for *var*, only its type!

Evaluation assertions are generalized to have the form

valenv $\vdash$ *exp* $\Downarrow$ *val*

where *valenv* is a *value environment* that records the bindings of the variables that may occur in *exp*. We may think of *valenv* as a sequence of specifications of the form

`val` *var* `=` *val*

that bind the value *val* to the variable *var*.

Finally, we also need a new assertion, called *type equivalence*, that determines when two types are equivalent, relative to a type environment. This is written

typenv $\vdash$ $typ_1 \equiv typ_2$

Two types are equivalent iff they are the same when the type constructors defined in *typenv* are replaced by their bindings.

The primary use of a type environment is to record the types of the value variables that are available for use in a given expression. This is expressed by the following axiom:

$\ldots$`val` *var* : *typ* $\ldots \vdash$ *var* : *typ*

[2] The *turnstile* symbol, "$\vdash$", is simply a punctuation mark separating the type environment from the expression and its type.

In words, if the specification `val` *var* `:` *typ* occurs in the type environment, then we may conclude that the variable *var* has type *typ*. This rule glosses over an important point. In order to account for shadowing we require that the *rightmost* specification govern the type of a variable. That way re-binding of variables with the same name but different types behaves as expected.

Similarly, the evaluation relation must take account of the value environment. Evaluation of variables is governed by the following axiom:

$$\ldots \texttt{val}\ \mathit{var}\ \texttt{=}\ \mathit{val} \ldots \vdash \mathit{var} \Downarrow \mathit{val}$$

Here again we assume that the `val` specification is the rightmost one governing the variable *var* to ensure that the scoping rules are respected.

The role of the type equivalence assertion is to ensure that type constructors always stand for their bindings. This is expressed by the following axiom:

$$\ldots \texttt{type}\ \mathit{typvar}\ \texttt{=}\ \mathit{typ} \ldots \vdash \mathit{typvar} \equiv \mathit{typ}$$

Once again, the rightmost specification for *typvar* governs the assertion.

3.6 Sample Code

Here is the complete code for this chapter.

Chapter 4

Functions

4.1 Functions as Templates

So far we just have the means to calculate the values of expressions, and to bind these values to variables for future reference. In this chapter we will introduce the ability to *abstract* the data from a calculation, leaving behind the bare *pattern* of the calculation. This pattern may then be *instantiated* as often as you like so that the calculation may be repeated with specified data values plugged in.

For example, consider the expression `2*(3+4)`. The data might be taken to be the values `2`, `3`, and `4`, leaving behind the pattern □ `*` (□ `+` □), with "holes" where the data used to be. We might equally well take the data to just be `2` and `3`, and leave behind the pattern □ `*` (□ `+ 4`). Or we might even regard `*` and `+` as the data, leaving `2` □ (`3` □ `4`) as the pattern! What is important is that a complete expression can be recovered by filling in the holes with chosen data.

Since a pattern can contain many different holes that can be independently instantiated, it is necessary to give *names* to the holes so that instantiation consists of plugging in a given value for all occurrences of a name in an expression. These names are, of course, just variables, and instantiation is just the process of substituting a value for all occurrences of a variable in a given expression. A pattern may therefore be viewed as a *function* of the variables that occur within it; the pattern is instantiated by *applying* the function to argument values.

This view of functions is similar to our experience from high school

algebra. In algebra we manipulate polynomials such as $x^2 + 2x + 1$ as a form of expression denoting a real number, with the variable x representing a fixed, but unknown, quantity. (Indeed, variables in algebra are sometimes called *unknowns*, or *indeterminates*.) It is also possible to think of a polynomial as a function on the real line: given a real number x, a polynomial determines a real number y computed by the given combination of arithmetic operations. Indeed, we sometimes write equations such as $f(x) = x^2 + 2x + 1$, to stand for the function f determined by the polynomial. In the univariate case we can get away with just writing the polynomial for the function, but in the multivariate case we must be more careful: we may regard the polynomial $x^2 + 2xy + y^2$ to be a function of x, a function of y, or a function of both x and y. In these cases we write $f(x) = x^2 + 2xy + y^2$ when x varies and y is held fixed, and $g(y) = x^2 + 2xy + y^2$ when y varies for fixed x, and $h(x, y) = x^2 + 2xy + y^2$, when both vary jointly.

In algebra it is usually left implicit that the variables x and y range over the real numbers, and that f, g, and h are functions on the real line. However, to be fully explicit, we sometimes write something like

$$f : \mathbb{R} \to \mathbb{R} \ : \ x \in R \mapsto x^2 + 2x + 1$$

to indicate that f is a function on the reals sending $x \in \mathbb{R}$ to $x^2 + 2x + 1 \in \mathbb{R}$. This notation has the virtue of separating the *name* of the function, f, from the function itself, the mapping that sends $x \in \mathbb{R}$ to $x^2 + 2x + 1$. It also emphasizes that functions are a kind of "value" in mathematics (namely, a certain set of ordered pairs), and that the variable f is bound to that value (*i.e.*, that set) by the declaration. This viewpoint is especially important once we consider operators, such as the differential operator, that map functions to functions. For example, if f is a differentiable function on the real line, the function $D\,f$ is its first derivative, another function on the real line. In the case of the function f defined above the function $D\,f$ sends $x \in \mathbb{R}$ to $2x + 2$.

4.2 Functions and Application

The treatment of functions in ML is very similar, except that we stress the *algorithmic* aspects of functions (*how* they determine values from arguments), as well as the *extensional* aspects (*what* they compute). As in

mathematics, a function in ML is a kind of value, namely a value of *function type* of the form *typ* `->` *typ′*. The type *typ* is the *domain type* (the type of arguments) of the function, and *typ′* is its *range type* (the type of its results). We compute with a function by *applying* it to an *argument* value of its domain type and calculating the *result*, a value of its range type. Function application is indicated by juxtaposition: we simply write the argument next to the function.

The values of function type consist of *primitive functions*, such as addition and square root, and *function expressions*, which are also called *lambda expressions*,[1] of the form

`fn` *var* `:` *typ* `=>` *exp*

The variable *var* is called the *parameter*, and the expression *exp* is called the *body*. It has type *typ*`->`*typ′* provided that *exp* has type *typ′* under the assumption that the parameter *var* has the type *typ*.

To apply such a function expression to an argument value *val*, we add the binding

`val` *var* `=` *val*

to the value environment, and evaluate *exp*, obtaining a value *val′*. Then the value binding for the parameter is removed, and the result value, *val′*, is returned as the value of the application.

For example, `Math.sqrt` is a primitive function of type `real->real` that may be applied to a real number to obtain its square root. For example, the expression `Math.sqrt 2.0` evaluates to `1.414` (approximately). We can, if we wish, parenthesize the argument, writing `Math.sqrt (2.0)` for the sake of clarity; this is especially useful for expressions such as `Math.sqrt (Math.sqrt 2.0)`. The square root function is built in. We may write the fourth root function as the following function expression:

```
fn x : real => Math.sqrt (Math.sqrt x)
```

It may be applied to an argument by writing an expression such as

```
(fn x : real => Math.sqrt (Math.sqrt x)) (16.0),
```

[1] For purely historical reasons.

which calculates the fourth root of `16.0`. The calculation proceeds by binding the variable `x` to the argument `16.0`, then evaluating the expression `Math.sqrt (Math.sqrt x)` in the presence of this binding. When evaluation completes, we drop the binding of `x` from the environment, since it is no longer needed.

Notice that we did not give the fourth root function a name; it is an "anonymous" function. We may give it a name using the declaration forms introduced in chapter 3. For example, we may bind the fourth root function to the variable `fourthroot` using the following declaration:

```
val fourthroot : real -> real =
    fn x : real => Math.sqrt (Math.sqrt x)
```

We may then write `fourthroot 16.0` to compute the fourth root of `16.0`.

This notation for defining functions quickly becomes tiresome, so ML provides a special syntax for function bindings that is more concise and natural. Instead of using the `val` binding above to define `fourthroot`, we may instead write

```
fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x)
```

This declaration has the same meaning as the earlier `val` binding, namely it binds `fn x:real => Math.sqrt(Math.sqrt x)` to the variable `fourthroot`.

It is important to note that function applications in ML are evaluated according to the *call-by-value* rule: the arguments to a function are evaluated before the function is called. Put in other terms, functions are defined to act on values, rather than on unevaluated expressions. Thus, to evaluate an expression such as `fourthroot (2.0+2.0)`, we proceed as follows:

1. Evaluate `fourthroot` to the function value `fn x : real => Math.sqrt (Math.sqrt x)`.
2. Evaluate the argument `2.0+2.0` to its value `4.0`
3. Bind `x` to the value `4.0`.
4. Evaluate `Math.sqrt (Math.sqrt x)` to `1.414` (approximately).
    (a) Evaluate `Math.sqrt` to a function value (the primitive square root function).

(b) Evaluate the argument expression `Math.sqrt x` to its value, approximately `2.0`.

i. Evaluate `Math.sqrt` to a function value (the primitive square root function).

ii. Evaluate `x` to its value, `4.0`.

iii. Compute the square root of `4.0`, yielding `2.0`.

(c) Compute the square root of `2.0`, yielding `1.414`.

5. Drop the binding for the variable `x`.

Notice that we evaluate *both* the function and argument positions of an application expression — both the function and argument are expressions yielding values of the appropriate type. The value of the function position must be a value of function type, either a primitive function or a lambda expression, and the value of the argument position must be a value of the domain type of the function. In this case the result value (if any) will be of the range type of the function. Functions in ML are *first-class*, meaning that they may be computed as the value of an expression. We are not limited to applying only named functions, but rather may compute "new" functions on the fly and apply these to arguments. This is a source of considerable expressive power, as we shall see in the sequel.

Using similar techniques we may define functions with arbitrary domain and range. For example, the following are all valid function declarations:

```
fun srev (s:string):string = implode (rev (explode s))
fun pal (s:string):string = s ^ (srev s)
fun double (n:int):int = n + n
fun square (n:int):int = n * n
fun halve (n:int):int = n div 2
fun is_even (n:int):bool = (n mod 2 = 0)
```

Thus `pal "ot"` evaluates to the string `"otto"`, and `is_even 4` evaluates to `true`.

4.3 Binding and Scope, Revisited

A function expression of the form

`fn` *var*`:`*typ* `=>` *exp*

binds the variable *var* within the body *exp* of the function. Unlike `val` bindings, function expressions bind a variable without giving it a specific value. The value of the parameter is only determined when the function is applied, and then only temporarily, for the duration of the evaluation of its body.

It is worth reviewing the rules for binding and scope of variables that we introduced in chapter 3 in the presence of function expressions. As before we adhere to the principle of *static scope*, according to which variables are taken to refer to the *nearest enclosing binding* of that variable, whether by a `val` binding or by a `fn` expression.

Thus, in the following example, the occurrences of `x` in the body of the function `f` refer to the parameter of `f`, whereas the occurrences of `x` in the body of `g` refer to the preceding `val` binding.

```
val x:real = 2.0
fun f(x:real):real = x+x
fun g(y:real):real = x+y
```

Local `val` bindings may shadow parameters, as well as other `val` bindings. For example, consider the following function declaration:

```
fun h(x:real):real =
    let val x:real = 2.0 in x+x end * x
```

The inner binding of `x` by the `val` declaration shadows the parameter `x` of `h`, but *only* within the body of the `let` expression. Thus the last occurrence of `x` refers to the parameter of `h`, whereas the preceding two occurrences refer to the inner binding of `x` to `2.0`.

The phrases "inner" and "outer" binding refer to the *logical structure*, or *abstract syntax* of an expression. In the preceding example, the body of `h` lies "within" the scope of the parameter `x`, and the expression `x+x` lies within the scope of the `val` binding for `x`. Since the occurrences of `x` within the body of the `let` lie within the scope of the inner `val` binding, they are taken to refer to that binding, rather than to the parameter. On the other hand the last occurrence of `x` does not lie within the scope of the `val` binding, and hence refers to the parameter of `h`.

In general the names of parameters do not matter; we can rename them at will without affecting the meaning of the program, provided that we

simultaneously (and consistently) rename the binding occurrence and all uses of that variable. Thus the functions `f` and `g` below are completely equivalent to each other:

```
fun f(x:int):int = x*x
fun g(y:int):int = y*y
```

A parameter is just a placeholder; its name is not important.

Our ability to rename parameters is constrained by the static scoping rule. We may rename a parameter to whatever we'd like, provided that we don't change the way in which uses of a variable are resolved. For example, consider the following situation:

```
val x:real = 2.0
fun h(y:real):real = x+y
```

The parameter `y` to `h` may be renamed to `z` without affecting its meaning. However, we may *not* rename it to `x`, for doing so changes its meaning! That is, the function

```
fun h'(x:real):real = x+x
```

does *not* have the same meaning as `h`, because now both occurrences of `x` in the body of `h'` refer to the parameter, whereas in `h` the variable `x` refers to the outer `val` binding, whereas the variable `y` refers to the parameter.

While this may seem like a minor technical issue, it is essential that you master these concepts now, for they play a central, and rather subtle, role later on.

4.4 Sample Code

Here is the complete code for this chapter.

Chapter 5

Products and Records

5.1 Product Types

A distinguishing feature of ML is that aggregate data structures, such as tuples, lists, arrays, or trees, may be created and manipulated with ease. In contrast to most familiar languages it is not necessary in ML to be concerned with allocation and deallocation of data structures, nor with any particular representation strategy involving, say, pointers or address arithmetic. Instead we may think of data structures as first-class values, on a par with every other value in the language. Just as it is unnecessary to think about "allocating" integers to evaluate an arithmetic expression, it is unnecessary to think about allocating more complex data structures such as tuples or lists.

5.1.1 Tuples

This chapter is concerned with the simplest form of aggregate data structure, the *n-tuple*. An n-tuple is a finite ordered sequence of values of the form

$(val_1, \ldots, val_n)$,

where each val_i is a value. A 2-tuple is usually called a *pair*, a 3-tuple a *triple*, and so on.

An n-tuple is a value of a *product type* of the form

$typ_1 * \ldots * typ_n$.

Values of this type are n-tuples of the form

$(val_1, \ldots, val_n)$,

where val_i is a value of type typ_i (for each $1 \leq i \leq n$).

Thus the following are well-formed bindings:

```
val pair : int * int = (2, 3)
val triple : int * real * string = (2, 2.0, "2")
val quadruple
    : int * int * real * real
    = (2,3,2.0,3.0)
val pair_of_pairs
    : (int * int) * (real * real)
    = ((2,3),(2.0,3.0))
```

The nesting of parentheses matters! A pair of pairs is not the same as a quadruple, so the last two bindings are of distinct values with distinct types.

There are two limiting cases, $n = 0$ and $n = 1$, that deserve special attention. A 0-tuple, which is also known as a *null tuple*, is the empty sequence of values, `()`. It is a value of type `unit`, which may be thought of as the 0-tuple type.[1] The null tuple type is surprisingly useful, especially when programming with effects. On the other hand there seems to be no particular use for 1-tuples, and so they are absent from the language.

As a convenience, ML also provides a general *tuple expression* of the form

$(exp_1, \ldots, exp_n)$,

where each exp_i is an arbitrary expression, not necessarily a value. Tuple expressions are evaluated from left to right, so that the above tuple expression evaluates to the tuple value

$(val_1, \ldots, val_n)$,

provided that exp_1 evaluates to val_1, exp_2 evaluates to val_2, and so on. For example, the binding

[1] In Java (and other languages) the type `unit` is misleadingly written `void`, which suggests that the type has *no* members, but in fact it has exactly one!

```
val pair : int * int = (1+1, 5-2)
```

binds the value `(2, 3)` to the variable `pair`.

Strictly speaking, it is not essential to have tuple expressions as a primitive notion in the language. Rather than write

$(exp_1, \ldots, exp_n)$,

with the (implicit) understanding that the exp_i's are evaluated from left to right, we may instead write

```
let val x1 = exp1
    val x2 = exp2
    ⋮
    val xn = expn
in  (x1,...,xn) end
```

which makes the evaluation order explicit.

5.1.2 Tuple Patterns

One of the most powerful, and distinctive, features of ML is the use of *pattern matching* to access components of aggregate data structures. For example, suppose that *val* is a value of type

```
(int * string) * (real * char)
```

and we wish to retrieve the first component of the second component of *val*, a value of type `real`. Rather than explicitly "navigate" to this position to retrieve it, we may simply use a generalized form of value binding in which we select that component using a pattern:

```
val ((_, _), (r:real, _)) = val
```

The left-hand side of the `val` binding is a tuple pattern that describes a pair of pairs, binding the first component of the second component to the variable `r`. The underscores indicate "don't care" positions in the pattern — their values are not bound to any variable. If we wish to give names to all of the components, we may use the following value binding:

```
val ((i:int, s:string), (r:real, c:char)) = val
```

If we'd like we can even give names to the first and second components of the pair, without decomposing them into constituent parts:

```
val (is:int*string,rc:real*char) = val
```

The general form of a value binding is

`val` *pat* `=` *exp*,

where *pat* is a *pattern* and *exp* is an expression. A pattern is one of three forms:

1. A *variable pattern* of the form *var*:*typ*.
2. A *tuple pattern* of the form $(pat_1, \ldots, pat_n)$, where each pat_i is a pattern. This includes as a special case the null-tuple pattern, `()`.
3. A *wildcard pattern* of the form `_`.

The type of a pattern is determined by an inductive analysis of the form of the pattern:

1. A variable pattern *var*:*typ* is of type *typ*.
2. A tuple pattern $(pat_1, \ldots, pat_n)$ has type $typ_1 * \cdots * typ_n$, where each pat_i is a pattern of type typ_i. The null-tuple pattern `()` has type `unit`.
3. The wildcard pattern `_` has any type whatsoever.

A value binding of the form

`val` *pat* `=` *exp*

is well-typed iff *pat* and *exp* have the same type; otherwise the binding is ill-typed and is rejected.

For example, the following bindings are well-typed:

```
val (m:int, n:int) = (7+1,4 div 2)
val (m:int, r:real, s:string) = (7, 7.0, "7")
val ((m:int,n:int), (r:real, s:real)) = ((4,5),(3.1,2.7))
val (m:int, n:int, r:real, s:real) = (4,5,3.1,2.7)
```

In contrast, the following are ill-typed:

```
val (m:int,n:int,r:real,s:real) = ((4,5),(3.1,2.7))
val (m:int, r:real) = (7+1,4 div 2)
val (m:int, r:real) = (7, 7.0, "7")
```

Value bindings are evaluated using the *bind-by-value* principle discussed earlier, except that the binding process is now more complex than before. First, we evaluate the right-hand side of the binding to a value (if indeed it has one). This happens regardless of the form of the pattern — the right-hand side is *always* evaluated. Second, we perform *pattern matching* to determine the bindings for the variables in the pattern.

The process of matching a value against a pattern is defined by a set of rules for reducing bindings with complex patterns to a set of bindings with simpler patterns, stopping once we reach a binding with a variable pattern. The rules are as follows:

1. The variable binding `val` *var* `=` *val* is irreducible.
2. The wildcard binding `val _ =` *val* is discarded.
3. The tuple binding

   `val (`pat_1`,...,`pat_n`) =`
   `(`val_1`,...,`val_n`)`

   is reduced to the set of n bindings

   `val` pat_1 `=` val_1

   ⋮

   `val` pat_n `=` val_n

   In the case that $n = 0$ the tuple binding is simply discarded.

These simplifications are repeated until all bindings are irreducible, which leaves us with a set of variable bindings that constitute the result of pattern matching.

For example, evaluation of the binding

```
val ((m:int,n:int), (r:real, s:real)) = ((2,3),(2.0,3.0))
```

proceeds as follows. First, we compose this binding into the following two bindings:

```
val (m:int, n:int) = (2,3)
and (r:real, s:real) = (2.0,3.0).
```

Then we decompose each of these bindings in turn, resulting in the following set of four atomic bindings:

```
val m:int = 2
and n:int = 3
and r:real = 2.0
and s:real = 3.0
```

At this point the pattern-matching process is complete.

5.2 Record Types

Tuples are most useful when the number of positions is small. When the number of components grows beyond a small number, it becomes difficult to remember which position plays which role. In that case it is more natural to attach a *label* to each component of the tuple that mediates access to it. This is the notion of a *record type*.

A record type has the form

$\{lab_1 : typ_1, \ldots, lab_n : typ_n\}$,

where $n \geq 0$, and all of the labels lab_i are distinct. A *record value* has the form

$\{lab_1 = val_1, \ldots, lab_n = val_n\}$,

where val_i has type typ_i. A *record pattern* has the form

$\{lab_1 = pat_1, \ldots, lab_n = pat_n\}$

which has type

$\{lab_1 : typ_1, \ldots, lab_n : typ_n\}$

provided that each pat_i has type typ_i.

A record value binding of the form

`val`
$\{lab_1$`=`$pat_1, \ldots, lab_n$`=`$pat_n\}$ `=`
$\{lab_1$`=`$val_1, \ldots, lab_n$`=`$val_n\}$

is decomposed into the following set of bindings

`val` pat_1 `=` val_1
`and ...`
`and` pat_n `=` val_n.

Since the components of a record are identified by name, not position, the order in which they occur in a record value or record pattern is not important. However, in a record *expression* (in which the components may not be fully evaluated), the fields are evaluated from left to right in the order written, just as for tuple expressions.

Here are some examples to help clarify the use of record types. First, let us define the record type `hyperlink` as follows:

```
type hyperlink =
     { protocol : string,
        address : string,
        display : string }
```

The record binding

```
val mailto_rwh : hyperlink =
    { protocol="mailto",
       address="rwh@cs.cmu.edu",
       display="Robert Harper" }
```

defines a variable of type `hyperlink`. The record binding

```
val { protocol=prot, display=disp, address=addr } = mailto_rwh
```

decomposes into the three variable bindings

```
val prot = "mailto"
val addr = "rwh@cs.cmu.edu"
val disp = "Robert Harper"
```

which extract the values of the fields of `mailto_rwh`.

Using wild cards we can extract selected fields from a record. For example, we may write

```
val {protocol=prot, address=_, display=_} = mailto_rwh
```

to bind the variable `prot` to the `protocol` field of the record value `mailto_rwh`.

It is quite common to encounter record types with tens of fields. In such cases even the wild card notation doesn't help much when it comes to selecting one or two fields from such a record. For this we often use *ellipsis patterns* in records, as illustrated by the following example.

```
val {protocol=prot,...} = intro_home
```

The pattern {`protocol=prot,...`} stands for the expanded pattern

{`protocol=prot, address=_, display=_`}

in which the elided fields are implicitly bound to wildcard patterns.

In general the ellipsis is replaced by as many wildcard bindings as are necessary to fill out the pattern to be consistent with its type. In order for this to occur *the compiler must be able to determine unambiguously the type of the record pattern*. Here the right-hand side of the value binding determines the type of the pattern, which then determines which additional fields to fill in. In some situations the context does not disambiguate, in which case you must supply additional type information, or avoid the use of ellipsis notation.

Finally, ML provides a convenient abbreviated form of record pattern

$\{lab_1, \ldots, lab_n\}$

which stands for the pattern

$\{lab_1{=}var_1, \ldots, lab_n{=}var_n\}$

where the variables var_i are variables with the same name as the corresponding label lab_i. For example, the binding

```
val { protocol, address, display } = mailto_rwh
```

decomposes into the sequence of atomic bindings

```
val protocol = "mailto"
val address = "rwh@cs.cmu.edu"
val display = "Robert Harper"
```

This avoids the need to think up a variable name for each field; we can just make the label do "double duty" as a variable.

5.3 Multiple Arguments and Multiple Results

A function may bind more than one argument by using a pattern, rather than a variable, in the argument position. Function expressions are generalized to have the form

`fn` *pat* `=>` *exp*

where *pat* is a pattern and *exp* is an expression. Application of such a function proceeds much as before, except that the argument value is matched against the parameter pattern to determine the bindings of zero or more variables, which are then used during the evaluation of the body of the function.

For example, we may make the following definition of the Euclidean distance function:

```
val dist
    : real * real -> real
    = fn (x:real, y:real) => sqrt (x*x + y*y)
```

This function may then be applied to a pair (a two-tuple!) of arguments to yield the distance between them. For example, `dist (2.0,3.0)` evaluates to (approximately) `4.0`.

Using `fun` notation, the distance function may be defined more concisely as follows:

```
fun dist (x:real, y:real):real = sqrt (x*x + y*y)
```

The meaning is the same as the more verbose `val` binding given earlier.

Keyword parameter passing is supported through the use of record patterns. For example, we may define the distance function using keyword parameters as follows:

```
fun dist' {x=x:real, y=y:real} = sqrt (x*x + y*y)
```

The expression `dist' {x=2.0,y=3.0}` invokes this function with the indicated x and y values.

Functions with multiple results may be thought of as functions yielding tuples (or records). For example, we might compute two different notions of distance between two points at once as follows:

```
fun dist2 (x:real, y:real):real*real
    = (sqrt (x*x+y*y), abs(x-y))
```

Notice that the result type is a pair, which may be thought of as two results.

These examples illustrate a pleasing regularity in the design of ML. Rather than introduce *ad hoc* notions such as multiple arguments, multiple results, or keyword parameters, we make use of the general mechanisms of tuples, records, and pattern matching.

It is sometimes useful to have a function to select a particular component from a tuple or record (*e.g.*, the third component or the component with a given label). Such functions may be easily defined using pattern matching. But since they arise so frequently, they are pre-defined in ML using *sharp notation*. For any tuple type

$typ_1 * \cdots * typ_n$,

and each $1 \leq i \leq n$, there is a function `#`i of type

$typ_1 * \cdots * typ_n$`->`typ_i

defined as follows:

```
fun #i (_, ..., _, x, _, ..., _) = x
```

where `x` occurs in the ith position of the tuple (and there are underscores in the other $n - 1$ positions).

Thus we may refer to the second field of a three-tuple *val* by writing `#2(`*val*`)`. It is bad style, however, to over-use the sharp notation; code is generally clearer and easier to maintain if you use patterns wherever possible. Compare, for example, the following definition of the Euclidean distance function written using sharp notation with the original.

```
fun dist (p:real*real):real
    = sqrt((#1 p)*(#1 p)+(#2 p)*(#2 p))
```

You can easily see that this gets out of hand very quickly, leading to unreadable code. *Use of the sharp notation is strongly discouraged!*

A similar notation is provided for record field selection. The following function `#lab` selects the component of a record with label lab.

```
fun #lab {lab=x,...} = x
```

Notice the use of ellipsis! Bear in mind the disambiguation requirement: any use of `#lab` must be in a context sufficient to determine the full record type of its argument.

5.4 Sample Code

Here is the complete code for this chapter.

Chapter 6

Case Analysis

6.1 Homogeneous and Heterogeneous Types

Tuple types have the property that all values of that type have the same form (n-tuples, for some n determined by the type); they are said to be *homogeneous*. For example, all values of type `int*real` are pairs whose first component is an integer and whose second component is a real. Any type-correct pattern will match any value of that type; there is no possibility of failure of pattern matching. The pattern `(x:int,y:real)` is of type `int*real` and hence will match any value of that type. On the other hand the pattern `(x:int,y:real,z:string)` is of type `int*real*string` and cannot be used to match against values of type `int*real`; attempting to do so *fails at compile time*.

Other types have values of more than one form; they are said to be *heterogeneous* types. For example, a value of type `int` might be `0`, `1`, `~1`, ... or a value of type `char` might be `#"a"` or `#"z"`. (Other examples of heterogeneous types will arise later on.) Corresponding to each of the values of these types is a pattern that matches only that value. Attempting to match any other value against that pattern *fails at execution time* with an error condition called a *bind failure*.

Here are some examples of pattern-matching against values of a heterogeneous type:

```
val 0 = 1-1
val (0,x) = (1-1, 34)
val (0, #"0") = (2-1, #"0")
```

The first two bindings succeed, the third fails. In the case of the second, the variable `x` is bound to `34` after the match. No variables are bound in the first or third examples.

6.2 Clausal Function Expressions

The importance of constant patterns becomes clearer once we consider how to define functions over heterogeneous types. This is achieved in ML using a *clausal function expression* whose general form is

`fn` pat_1 `=>` exp_1
`|` $\vdots$
`|` pat_n `=>` exp_n

Each pat_i is a pattern and each exp_i is an expression involving the variables of the pattern pat_i. Each component *pat*`=>`*exp* is called a *clause*, or a *rule*. The entire assembly of rules is called a *match*.

The typing rules for matches ensure consistency of the clauses. Specifically, there must exist types typ_1 and typ_2 such that

1. Each pattern pat_i has type typ_1.
2. Each expression exp_i has type typ_2, given the types of the variables in pattern pat_i.

If these requirements are satisfied, the function has the type typ_1`->`typ_2.

Application of a clausal function to a value *val* proceeds by considering the clauses *in the order written*. At stage i, where $1 \leq i \leq n$, the argument value *val* is matched against the pattern pat_i; if the pattern match succeeds, evaluation continues with the evaluation of expression exp_i, with the variables of pat_i replaced by their values as determined by pattern matching. Otherwise we proceed to stage $i + 1$. If no pattern matches (*i.e.*, we reach stage $n + 1$), then the application fails with an execution error called a *match failure*.

Here's an example. Consider the following clausal function:

```
val recip : int -> int =
    fn 0 => 0 | n:int => 1 div n
```

This defines an integer-valued reciprocal function on the integers, where the reciprocal of `0` is arbitrarily defined to be `0`. The function has two clauses, one for the argument `0`, the other for non-zero arguments `n`. (Note that `n` is guaranteed to be non-zero because the patterns are considered in order: we reach the pattern `n:int` only if the argument fails to match the pattern `0`.)

The `fun` notation is also generalized so that we may define `recip` using the following more concise syntax:

```
fun recip 0 = 0
  | recip (n:int) = 1 div n
```

One annoying thing to watch out for is that the `fun` form uses an equal sign to separate the pattern from the expression in a clause, whereas the `fn` form uses a double arrow.

Case analysis on the values of a heterogeneous type is performed by application of a clausally-defined function. The notation

```
case exp
  of pat1 => exp1
   | ...
   | patn => expn
```

is short for the application

```
(fn pat1 => exp1
  | ...
  | patn => expn)
exp.
```

Evaluation proceeds by first evaluating *exp*, then matching its value successively against the patterns in the match until one succeeds, and continuing with evaluation of the corresponding expression. The `case` expression fails if no pattern succeeds to match the value.

6.3 Booleans and Conditionals, Revisited

The type `bool` of booleans is perhaps the most basic example of a heterogeneous type. Its values are `true` and `false`. Functions may be defined

on booleans using clausal definitions that match against the patterns `true` and `false`.

For example, the negation function may be defined clausally as follows:

```
fun not true = false
  | not false = true
```

The conditional expression

`if` *exp* `then` exp_1 `else` exp_2

is short-hand for the case analysis

`case` *exp*
`  of true =>` exp_1
`   | false =>` exp_2

which is itself short-hand for the application

`(fn true =>` exp_1 `| false =>` exp_2`)` *exp*.

The "short-circuit" conjunction and disjunction operations are defined as follows. The expression exp_1 `andalso` exp_2 is short for

`if` exp_1 `then` exp_2 `else false`

and the expression exp_1 `orelse` exp_2 is short for

`if` exp_1 `then true else` exp_2.

You should expand these into case expressions and check that they behave as expected. Pay particular attention to the evaluation order, and observe that the call-by-value principle is not violated by these expressions.

6.4 Exhaustiveness and Redundancy

Matches are subject to two forms of "sanity check" as an aid to the ML programmer. The first, called *exhaustiveness checking*, ensures that a well-formed match *covers* its domain type in the sense that every value of the

domain must match one of its clauses. The second, called *redundancy checking*, ensures that no clause of a match is subsumed by the clauses that precede it. This means that the set of values covered by a clause in a match must not be contained entirely within the set of values covered by the preceding clauses of that match.

Redundant clauses are *always* a mistake — such a clause can never be executed. Redundant rules often arise accidentally. For example, the second rule of the following clausal function definition is redundant:

```
fun not True = false
  | not False = true
```

By capitalizing `True` we have turned it into a variable, rather than a constant pattern. Consequently, *every* value matches the first clause, rendering the second redundant.

Since the clauses of a match are considered in the order they are written, redundancy checking is correspondingly order-sensitive. In particular, changing the order of clauses in a well-formed, irredundant match can make it redundant, as in the following example:

```
fun recip (n:int) = 1 div n
  | recip 0 = 0
```

The second clause is redundant because the first matches *any* integer value, including `0`.

Inexhaustive matches may or may not be in error, depending on whether the match might ever be applied to a value that is not covered by any clause. Here is an example of a function with an inexhaustive match that is plausibly in error:

```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
```

When applied to, say, `#"a"`, this function fails. Indeed, the function never returns `false` for any argument!

Perhaps what was intended here is to include a *catch-all* clause at the end:

```
fun is_numeric #"0" = true
    | is_numeric #"1" = true
    | is_numeric #"2" = true
    | is_numeric #"3" = true
    | is_numeric #"4" = true
    | is_numeric #"5" = true
    | is_numeric #"6" = true
    | is_numeric #"7" = true
    | is_numeric #"8" = true
    | is_numeric #"9" = true
    | is_numeric _ = false
```

The addition of a final catch-all clause renders the match exhaustive, because any value not matched by the first ten clauses will surely be matched by the eleventh.

Having said that, it is a very bad idea to simply add a catch-all clause to the end of every match to suppress inexhaustiveness warnings from the compiler. The exhaustiveness checker is your friend! Each such warning is a suggestion to double-check that match to be sure that you've not made a silly error of omission, but rather have intentionally left out cases that are ruled out by the invariants of the program. In chapter 10 we will see that the exhaustiveness checker is an extremely valuable tool for managing code evolution.

6.5 Sample Code

Here is the complete code for this chapter.

Chapter 7

Recursive Functions

So far we've only considered very simple functions (such as the reciprocal function) whose value is computed by a simple composition of primitive functions. In this chapter we introduce *recursive* functions, the principal means of iterative computation in ML. Informally, a recursive function is one that computes the result of a call by possibly making further calls to itself. Obviously, to avoid infinite regress, some calls must return their results without making any recursive calls. Those that do must ensure that the arguments are, in some sense, "smaller" so that the process will eventually terminate.

This informal description obscures a central point, namely the means by which we may convince ourselves that a function computes the result that we intend. In general we must prove that for all inputs of the domain type, the body of the function computes the "correct" value of result type. Usually the argument imposes some additional assumptions on the inputs, called the *pre-conditions*. The correctness requirement for the result is called a *post-condition*. Our burden is to prove that for every input satisfying the pre-conditions, the body evaluates to a result satisfying the post-condition. In fact we may carry out such an analysis for many different pre- and post-condition pairs, according to our interest. For example, the ML type checker proves that the body of a function yields a value of the range type (if it terminates) whenever it is given an argument of the domain type. Here the domain type is the pre-condition, and the range type is the post-condition. In most cases we are interested in deeper properties, examples of which we shall consider below.

To prove the correctness of a recursive function (with respect to given

pre- and post-conditions) it is typically necessary to use some form of inductive reasoning. The base cases of the induction correspond to those cases that make no recursive calls; the inductive step corresponds to those that do. The beauty of inductive reasoning is that we may *assume* that the recursive calls work correctly when showing that a case involving recursive calls is correct. We must separately show that the base cases satisfy the given pre- and post-conditions. Taken together, these two steps are sufficient to establish the correctness of the function itself, by appeal to an induction principle that justifies the particular pattern of recursion.

No doubt this all sounds fairly theoretical. The point of this chapter is to show that it is also profoundly practical.

7.1 Self-Reference and Recursion

In order for a function to "call itself", it must have a name by which it can refer to itself. This is achieved by using a *recursive value binding*, which are ordinary value bindings qualified by the keyword `rec`. The simplest form of a recursive value binding is as follows:

`val rec` *var*`:`*typ* `=` *val*`.`

As in the non-recursive case, the left-hand is a pattern, but here the right-hand side must be a value. In fact the right-hand side must be a function expression, since only functions may be defined recursively in ML. The function may refer to itself by using the variable *var*.

Here's an example of a recursive value binding:

```
val rec factorial : int->int =
    fn 0 => 1 | n:int => n * factorial (n-1)
```

Using `fun` notation we may write the definition of factorial much more clearly and concisely as follows:

```
fun factorial 0 = 1
  | factorial (n:int) = n * factorial (n-1)
```

There is obviously a close correspondence between this formulation of `factorial` and the usual textbook definition of the factorial function in

terms of recursion equations:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad (n > 0) \end{aligned}$$

Recursive value bindings are type-checked in a manner that may, at first glance, seem paradoxical. To check that the binding

`val rec` *var* `:` *typ* `=` *val*

is well-formed, we ensure that the value *val* has type *typ*, *assuming* that *var* has type *typ*. Since *var* refers to the value *val* itself, we are in effect assuming what we intend to prove while proving it!

(Incidentally, since *val* is required to be a function expression, the type *typ* will always be a function type.)

Let's look at an example. To check that the binding for `factorial` given above is well-formed, we *assume* that the variable `factorial` has type `int->int`, then *check* that its definition, the function

```
fn 0 => 1 | n:int => n * factorial (n-1),
```

has type `int->int`. To do so we must check that each clause has type `int->int` by checking for each clause that its pattern has type `int` and that its expression has type `int`. This is clearly true for the first clause of the definition. For the second, we assume that `n` has type `int`, then check that `n * factorial (n-1)` has type `int`. This is so because of the rules for the primitive arithmetic operations and because of our assumption that `factorial` has type `int->int`.

How are applications of recursive functions evaluated? The rules are almost the same as before, with one modification. We must arrange that all occurrences of the variable standing for the function are replaced by the function itself before we evaluate the body. That way all references to the variable standing for the function itself are indeed references to the function itself!

Suppose that we have the following recursive function binding

```
val rec var : typ =
    fn pat1 => exp1
     | ...
     | patn => expn
```

and we wish to apply *var* to the value *val* of type *typ*. As before, we consider each clause in turn, until we find the first pattern pat_i matching *val*. We proceed, as before, by evaluating exp_i, replacing the variables in pat_i by the bindings determined by pattern matching, but, *in addition*, we replace all occurrences of the *var* by its binding in exp_i before continuing evaluation.

For example, to evaluate `factorial 3`, we proceed by retrieving the binding of `factorial` and evaluating

```
(fn 0=>1 | n:int => n*factorial(n-1))(3).
```

Considering each clause in turn, we find that the first doesn't match, but the second does. We therefore continue by evaluating its right-hand side, the expression `n * factorial(n-1)`, after replacing `n` by `3` and `factorial` by its definition. We are left with the sub-problem of evaluating the expression

```
3 * (fn 0 => 1 | n:int => n*factorial(n-1))(2)
```

Proceeding as before, we reduce this to the sub-problem of evaluating

```
3 * (2 * (fn 0=>1 | n:int => n*factorial(n-1))(1)),
```

which reduces to the sub-problem of evaluating

```
3 * (2 * (1 * (fn 0=>1 | n:int => n*factorial(n-1))(0))),
```

which reduces to

```
3 * (2 * (1 * 1)),
```

which then evaluates to `6`, as desired.

Observe that the repeated substitution of `factorial` by its definition ensures that the recursive calls really do refer to the factorial function itself. Also observe that the size of the sub-problems grows until there are no more recursive calls, at which point the computation can complete. In broad outline, the computation proceeds as follows:

1. `factorial 3`
2. `3 * factorial 2`

3. `3 * 2 * factorial 1`
4. `3 * 2 * 1 * factorial 0`
5. `3 * 2 * 1 * 1`
6. `3 * 2 * 1`
7. `3 * 2`
8. `6`

Notice that the size of the expression first grows (in direct proportion to the argument), then shrinks as the pending multiplications are completed. This growth in expression size corresponds directly to a growth in run-time storage required to record the state of the pending computation.

7.2 Iteration

The definition of `factorial` given above should be contrasted with the following two-part definition:

```
fun helper (0,r:int) = r
  | helper (n:int,r:int) = helper (n-1,n*r)
fun factorial (n:int) = helper (n, 1)
```

First we define a "helper" function that takes two parameters, an integer argument and an *accumulator* that records the running partial result of the computation. The idea is that the accumulator re-associates the pending multiplications in the evaluation trace given above so that they can be performed prior to the recursive call, rather than after it completes. This reduces the space required to keep track of those pending steps. Second, we define `factorial` by calling `helper` with argument `n` and initial accumulator value `1`, corresponding to the product of zero terms (empty prefix).

As a matter of programming style, it is usual to conceal the definitions of helper functions using a `local` declaration. In practice we would make the following definition of the iterative version of `factorial`:

```
local
      fun helper (0,r:int) = r
        | helper (n:int,r:int) = helper (n-1,n*r)
in
      fun factorial (n:int) = helper (n,1)
end
```

This way the helper function is not visible, only the function of interest is "exported" by the declaration.

The important thing to observe about `helper` is that it is *iterative*, or *tail recursive*, meaning that the recursive call is the last step of evaluation of an application of it to an argument. This means that the evaluation trace of a call to `helper` with arguments `(3,1)` has the following general form:

1. `helper (3, 1)`
2. `helper (2, 3)`
3. `helper (1, 6)`
4. `helper (0, 6)`
5. `6`

Notice that there is no growth in the size of the expression because there are no pending computations to be resumed upon completion of the recursive call. Consequently, there is no growth in the space required for an application, in contrast to the first definition given above. Tail recursive definitions are analogous to loops in imperative languages: they merely iterate a computation, without requiring auxiliary storage.

7.3 Inductive Reasoning

Time and space usage are important, but what is more important is that the function compute the intended result. The key to the correctness of a recursive function is an inductive argument establishing its correctness. The critical ingredients are these:

1. An *input-output specification* of the intended behavior stating *pre-conditions* on the arguments and a *post-condition* on the result.

2. A proof that the specification holds for each clause of the function, *assuming* that it holds for any recursive calls.

3. An *induction principle* that justifies the correctness of the function as a whole, given the correctness of its clauses.

We'll illustrate the use of inductive reasoning by a graduated series of examples. First consider the simple, non-tail recursive definition of `factorial` given in section 7.1. One reasonable specification for `factorial` is as follows:

1. *Pre-condition*: $n \geq 0$.

2. *Post-condition*: `factorial` n evaluates to $n!$.

We are to establish the following statement of correctness of `factorial`:

> if $n \geq 0$, then `factorial` n evaluates to $n!$.

That is, we show that the pre-conditions imply the post-condition holds of the result of any application. This is called a *total correctness* assertion because it states not only that the post-condition holds of any result of application, but, moreover, that every application in fact yields a result (subject to the pre-condition on the argument).

In contrast, a *partial correctness* assertion does not insist on termination, only that the post-condition holds whenever the application terminates. This may be stated as the assertion

> if $n \geq 0$ and `factorial` n evaluates to p, then $p = n!$.

Notice that this statement is true of a function that diverges whenever it is applied! In this sense a partial correctness assertion is weaker than a total correctness assertion.

Let us establish the total correctness of `factorial` using the pre- and post-conditions stated above. To do so, we apply the principle of *mathematical induction* on the argument n. Recall that this means we are to establish the specification for the case $n = 0$, and, assuming it to hold for $n >= 0$, show that it holds for $n + 1$. The base case, $n = 0$, is trivial: by definition `factorial` n evaluates to 1, which is 0!. Now suppose that $n = m + 1$ for some $m >= 0$. By the inductive hypothesis we have that

`factorial` m evaluates to $m!$ (since $m \geq 0$), and so by definition `factorial` n evaluates to

$$\begin{aligned} n \times m! &= (m+1) \times m! \\ &= (m+1)! \\ &= n!, \end{aligned}$$

as required. This completes the proof.

That was easy. What about the iterative definition of `factorial`? We focus on the behavior of `helper`. A suitable specification is given as follows:

1. *Pre-condition*: $n \geq 0$.
2. *Post-condition*: `helper (`n`,` r`)` evaluates to $n! \times r$.

To show the total correctness of `helper` with respect to this specification, we once again proceed by mathematical induction on n. We leave it as an exercise to give the details of the proof.

With this in hand it is easy to prove the correctness of `factorial` — if $n \geq 0$ then `factorial` n evaluates to the result of `helper (n, 1)`, which evaluates to $n! \times 1 = n!$. This completes the proof.

Helper functions correspond to lemmas, main functions correspond to theorems. Just as we use lemmas to help us prove theorems, we use helper functions to help us define main functions. The foregoing argument shows that this is more than an analogy, but lies at the heart of good programming style.

Here's an example of a function defined by *complete* induction (or *strong* induction), the Fibonacci function, defined on integers $n >= 0$:

```
(* for n>=0, fib n yields the nth Fibonacci number *)
fun fib 0 = 1
  | fib 1 = 1
  | fib (n:int) = fib (n-1) + fib (n-2)
```

The recursive calls are made not only on `n-1`, but also `n-2`, which is why we must appeal to complete induction to justify the definition. This definition of `fib` is very inefficient because it performs many redundant computations: to compute `fib n` requires that we compute `fib (n-1)` and `fib (n-2)`. To compute `fib (n-1)` requires that we compute `fib (n-2)` a second time, and `fib (n-3)`. Computing `fib (n-2)` requires computing `fib`

`(n-3)` again, and `fib (n-4)`. As you can see, there is considerable redundancy here. It can be shown that the running time `fib` of is exponential in its argument, which is quite awful.

Here's a better solution: for each $n >= 0$ compute not only the nth Fibonacci number, but also the $(n-1)$st as well. (For $n = 0$ we define the "-1st" Fibonacci number to be zero). That way we can avoid redundant recomputation, resulting in a linear-time algorithm. Here's the code:

```
(* for n>=0, fib' n evaluates to (a, b), where
   a is the nth Fibonacci number, and
   b is the (n-1)st *)
fun fib' 0 = (1, 0)
  | fib' 1 = (1, 1)
  | fib' (n:int) =
    let
        val (a:int, b:int) = fib' (n-1)
    in
        (a+b, a)
    end
```

You might feel satisfied with this solution since it runs in time linear in n. It turns out (see Graham, Knuth, and Patashnik, *Concrete Mathematics* (Addison-Wesley 1989) for a derivation) that the recurrence

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

has a closed-form solution over the real numbers. This means that the nth Fibonacci number can be calculated directly, without recursion, by using floating point arithmetic. However, this is an unusual case. In most instances recursively-defined functions have no known closed-form solution, so that some form of iteration is inevitable.

7.4 Mutual Recursion

It is often useful to define two functions simultaneously, each of which calls the other (and possibly itself) to compute its result. Such functions

are said to be *mutually recursive*. Here's a simple example to illustrate the point, namely testing whether a natural number is odd or even. The most obvious approach is to test whether the number is congruent to 0 mod 2, and indeed this is what one would do in practice. But to illustrate the idea of mutual recursion we instead use the following inductive characterization: 0 is even, and not odd; $n > 0$ is even iff $n - 1$ is odd; $n > 0$ is odd iff $n - 1$ is even. This may be coded up using two mutually-recursive procedures as follows:

```
fun even 0 = true
  | even n = odd (n-1)
and odd 0 = false
  | odd n = even (n-1)
```

Notice that `even` calls `odd` and `odd` calls `even`, so they are not definable separately from one another. We join their definitions using the keyword `and` to indicate that they are defined simultaneously by mutual recursion.

7.5 Sample Code

Here is the complete code for this chapter.

Chapter 8

Type Inference and Polymorphism

8.1 Type Inference

So far we've mostly written our programs in what is known as the *explicitly typed* style. This means that whenever we've introduced a variable, we've assigned it a type at its point of introduction. In particular every variable in a pattern has a type associated with it. As you may have noticed, this gets a little tedious after a while, especially when you're using clausal function definitions. A particularly pleasant feature of ML is that it allows you to omit this type information whenever it can be determined from context. This process is known as *type inference* since the compiler is inferring the missing type information based on context.

For example, there is no need to give a type to the variable `s` in the function

```
fn s:string => s ^ "\n".
```

The reason is that no other type for `s` makes sense, since `s` is used as an argument to string concatenation. Consequently, you may write simply

```
fn s => s ^ "\n",
```

leaving ML to insert "`:string`" for you.

When is it allowable to omit this information? Almost always, with very few exceptions. It is a deep, and important, result about ML that

missing type information can (almost) always be reconstructed completely and unambiguously where it is omitted. This is called the *principal typing property* of ML: whenever type information is omitted, there is always a *most general* (*i.e.*, *least restrictive*) way to recover the omitted type information. If there is no way to recover the omitted type information, then the expression is ill-typed. Otherwise there is a "best" way to fill in the blanks, which will (almost) always be found by the compiler. This is an amazingly useful, and widely under-appreciated, property of ML. It means, for example, that the programmer can enjoy the full benefits of a static type system without paying any notational penalty whatsoever!

The prototypical example is the identity function, `fn x=>x`. The body of the function places no constraints on the type of `x`, since it merely returns `x` as the result without performing any computation on it. Since the behavior of the identity function is the same for all possible choices of type for its argument, it is said to be *polymorphic*. Therefore the identity function has *infinitely many* types, one for each choice of the type of the parameter `x`. Choosing the type of `x` to be *typ*, the type of the identity function is *typ*`->`*typ*. In other words every type for the identity function has the form *typ*`->`*typ*, where *typ* is the type of the argument.

Clearly there is a pattern here, which is captured by the notion of a *type scheme*. A type scheme is a type expression involving one or more *type variables* standing for an unknown, but arbitrary type expression. Type variables are written `'a` (pronounced "α"), `'b` (pronounced "β"), `'c` (pronounced "γ"), *etc.*. An *instance* of a type scheme is obtained by replacing each of the type variables occurring in it with a type scheme, with the same type scheme replacing each occurrence of a given type variable. For example, the type scheme `'a->'a` has instances `int->int`, `string->string`, `(int*int)->(int*int)`, and `('b->'b)->('b->'b)`, among infinitely many others. However, it does *not* have the type `int->string` as instance, since we are constrained to replace all occurrences of a type variable by the same type scheme. However, the type scheme `'a->'b` has both `int->int` and `int->string` as instances since there are different type variables occurring in the domain and range positions.

Type schemes are used to express the polymorphic behavior of functions. For example, we may write `fn x:'a=>x` for the polymorphic identity function of type `'a->'a`, meaning that the behavior of the identity function is independent of the type of `x`. Similarly, the behavior of the function `fn (x,y)=>x+1` is independent of the type of `y`, but constrains the

type of `x` to be `int`. This may be expressed using type schemes by writing this function in the explicitly-typed form `fn (x:int,y:'a)=>x+1` with type `int*'a->int`.

In these examples we needed only one type variable to express the polymorphic behavior of a function, but usually we need more than one. For example, the function `fn (x,y) = x` constrains neither the type of `x` nor the type of `y`. Consequently we may choose their types freely and independently of one another. This may be expressed by writing this function in the form `fn (x:'a,y:'b)=>x` with type scheme `'a*'b->'a`. Notice that while it is correct to assign the type `'a*'a->'a` to this function, doing so would be overly restrictive since the types of the two parameters need not be the same. However, we could *not* assign the type `'a*'b->'c` to this function because the type of the result must be the same as the type of the first parameter: it returns its first parameter when invoked! The type scheme `'a*'b->'a` precisely captures the constraints that must be satisfied for the function to be type correct. It is said to be the *most general* or *principal type scheme* for the function.

It is a remarkable fact about ML that *every expression* (with the exception of a few pesky examples that we'll discuss below) *has a principal type scheme*. That is, there is (almost) always a *best* or *most general* way to infer types for expressions that *maximizes generality*, and hence *maximizes flexibility* in the use of the expression. Every expression "seeks its own depth" in the sense that an occurrence of that expression is assigned a type that is an instance of its principal type scheme determined by the context of use.

For example, if we write

```
(fn x=>x)(0),
```

the context forces the type of the identity function to be `int->int`, and if we write

```
(fn x=>x)(fn x=>x)(0)
```

the context forces the instance `(int->int)->(int->int)` of the principal type scheme for the identity at the first occurrence, and the instance `int->int` for the second.

How is this achieved? Type inference is a process of *constraint satisfaction*. First, the expression determines a set of equations governing the relationships among the types of its subexpressions. For example, if a function

is applied to an argument, then a constraint equating the domain type of the function with the type of the argument is generated. Second, the constraints are solved using a process similar to Gaussian elimination, called *unification*. The equations can be classified by their solution sets as follows:

1. *Overconstrained*: there is no solution. This corresponds to a type error.
2. *Underconstrained*: there are many solutions. There are two sub-cases: *ambiguous* (due to overloading, which we will discuss further in section 8.3), or *polymorphic* (there is a "best" solution).
3. *Uniquely determined:* there is precisely one solution. This corresponds to a completely unambiguous type inference problem.

The free type variables in the solution to the system of equations may be thought of as determining the "degrees of freedom" or "range of polymorphism" of the type of an expression — the constraints are solvable for any choice of types to substitute for these free type variables.

This description of type inference as a constraint satisfaction procedure accounts for the notorious obscurity of type checking errors in ML. If a program is not type correct, then the system of constraints associated with it will not have a solution. The type inference procedure attempts to find a solution to these constraints, and at some point discovers that it cannot succeed. It is fundamentally impossible to attribute this inconsistency to any particular constraint; all that can be said is that the constraint set as a whole has no solution. The checker usually reports the first unsatisfiable equation it encounters, but this may or may not correspond to the "reason" (in the mind of the programmer) for the type error. The usual method for finding the error is to insert sufficient type information to narrow down the source of the inconsistency until the source of the difficulty is uncovered.

8.2 Polymorphic Definitions

There is an important interaction between polymorphic expressions and value bindings that may be illustrated by the following example. Suppose that we wish to bind the identity function to a variable `I` so that we may refer to it by name. We've previously observed that the identity function

is polymorphic, with principal type scheme `'a->'a`. This may be captured by ascribing this type scheme to the variable `I` at the `val` binding. That is, we may write

```
val I : 'a->'a = fn x=>x
```

to ascribe the type scheme `'a->'a` to the variable `I`. (We may also write

```
fun I(x:'a):'a = x
```

for an equivalent binding of `I`.) Having done this, *each use of `I` determines a distinct instance of the ascribed type scheme* `'a->'a`. That is, both `I 0` and `I I 0` are well-formed expressions, the first assigning the type `int->int` to `I`, the second assigning the types

```
(int->int)->(int->int)
```

and

```
int->int
```

to the two occurrences of `I`. Thus the variable `I` behaves precisely the same as its definition, `fn x=>x`, in any expression where it is used.

As a convenience ML also provides a form of type inference on value bindings that eliminates the need to ascribe a type scheme to the variable when it is bound. If no type is ascribed to a variable introduced by a `val` binding, then it is implicitly ascribed the principal type scheme of the right-hand side. For example, we may write

```
val I = fn x=>x
```

or

```
fun I(x) = x
```

as a binding for the variable . The type checker implicitly assigns the principal type scheme, `'a->'a`, of the binding to the variable `I`. In practice we often allow the type checker to infer the principal type of a variable, but it is often good form to explicitly indicate the intended type as a consistency check and for documentation purposes.

The treatment of `val` bindings during type checking ensures that a bound variable has precisely the same type as its binding. In other words

the type checker behaves as though all uses of the bound variable are implicitly replaced by its binding before type checking. Since this may involve replication of the binding, the meaning of a program is not necessarily preserved by this transformation. (Think, for example, of any expression that opens a window on your screen: if you replicate the expression and evaluate it twice, it will open two windows. This is not the same as evaluating it only once, which results in one window.)

To ensure semantic consistency, variables introduced by a `val` binding are allowed to be polymorphic *only if* the right-hand side is a value. This is called the *value restriction* on polymorphic declarations. For `fun` bindings this restriction is always met since the right-hand side is implicitly a lambda expression, which is a value. However, it might be thought that the following declaration introduces a polymorphic variable of type `'a -> 'a`, but in fact it is rejected by the compiler:

```
val J = I I
```

The reason is that the right-hand side is not a value; it requires computation to determine its value. It is therefore ruled out as inadmissible for polymorphism; the variable `J` may not be used polymorphically in the remainder of the program. In this case the difficulty may be avoided by writing instead

```
fun J x = I I x
```

because now the binding of `J` is a lambda, which is a value.

In some rare circumstances this is not possible, and some polymorphism is lost. For example, the following declaration of a value of list type[1]

```
val l = nil @ nil
```

does not introduce an identifier with a polymorphic type, even though the almost equivalent declaration

```
val l = nil
```

does do so. Since the right-hand side is a list, we cannot apply the "trick" of defining `l` to be a function; we are stuck with a loss of polymorphism in

[1] To be introduced in chapter 9.

this case. This particular example is not very impressive, but occasionally similar examples do arise in practice.

Why is the value restriction necessary? Later on, when we study mutable storage, we'll see that *some* restriction on polymorphism is essential if the language is to be type safe. The value restriction is an easily-remembered sufficient condition for soundness, but as the examples above illustrate, it is by no means necessary. The designers of ML were faced with a choice of simplicity *vs* flexibility; in this case they opted for simplicity at the expense of some expressiveness in the language.

8.3 Overloading

Type information cannot always be omitted. There are a few corner cases that create problems for type inference, most of which arise because of concessions that are motivated by long-standing, if dubious, notational practices.

The main source of difficulty stems from *overloading* of arithmetic operators. As a concession to long-standing practice in informal mathematics and in many programming languages, the same notation is used for both integer and floating point arithmetic operations. As long as we are programming in an explicitly-typed style, this convention creates no particular problems. For example, in the function

```
fn x:int => x+x
```

it is clear that integer addition is called for, whereas in the function

```
fn x:real => x+x
```

it is equally obvious that floating point addition is intended.

However, if we *omit* type information, then a problem arises. What are we to make of the function

```
fn x => x+x ?
```

Does "+" stand for integer or floating point addition? There are two distinct reconstructions of the missing type information in this example, corresponding to the preceding two explictly-typed programs. Which is the compiler to choose?

When presented with such a program, the compiler has two choices:

1. Declare the expression ambiguous, and force the programmer to provide enough explicit type information to resolve the ambiguity.

2. Arbitrarily choose a "default" interpretation, say the integer arithmetic, that forces one interpretation or another.

Each approach has its advantages and disadvantages. Many compilers choose the second approach, but issue a warning indicating that it has done so. To avoid ambiguity, explicit type information is required from the programmer.

The situation is actually a bit more subtle than the preceding discussion implies. The reason is that the type inference process makes use of the surrounding context of an expression to help resolve ambiguities. For example, if the expression `fn x=>x+x` occurs in the following, larger expression, there is in fact no ambiguity:

```
(fn x => x+x)(3).
```

Since the function is applied to an integer argument, there is no question that the only possible resolution of the missing type information is to treat `x` as having type `int`, and hence to treat `+` as integer addition.

The important question is how much context is considered before the situation is considered ambiguous? The rule of thumb is that context is considered up to the nearest enclosing function declaration. For example, consider the following example:

```
let
    val double = fn x => x+x
in
    (double 3, double 4)
end
```

The function expression `fn x=>x+x` will be flagged as ambiguous, even though its only uses are with integer arguments. The reason is that value bindings are considered to be "units" of type inference for which all ambiguity must be resolved before type checking continues. If your compiler adopts the integer interpretation as default, the above program will be accepted (with a warning), but the following one will be rejected:

```
let
    val double = fn x => x+x
in
    (double 3.0, double 4.0)
end
```

Finally, note that the following program *must* be rejected because no resolution of the overloading of addition can render it meaningful:

```
let
    val double = fn x => x+x
in
    (double 3, double 3.0)
end
```

The ambiguity must be resolved at the `val` binding, which means that the compiler must commit at that point to treating the addition operation as either integer or floating point. No single choice can be correct, since we subsequently use `double` at both types.

A closely related source of ambiguity arises from the "record elision" notation described in chapter 5. Consider the function `#name`, defined by

```
fun #name {name=n:string, ...} = n
```

which selects the `name` field of a record. This definition is *ambiguous* because the compiler cannot uniquely determine the domain type of the function! Any of the following types are legitimate domain types for `#name`, none of which is "best":

```
{name:string}
{name:string,salary:real}
{name:string,salary:int}
{name:string,address:string}
```

Of course there are infinitely many such examples, none of which is clearly preferable to the other. This function definition is therefore rejected as ambiguous by the compiler — there is no one interpretation of the function that suffices for all possible uses.

In chapter 5 we mentioned that functions such as `#name` are *pre-defined* by the ML compiler, yet we just now claimed that such a function definition is rejected as ambiguous. Isn't this a contradiction? Not really,

because what happens is that each occurrence of `#name` is replaced by the function

```
fn {name=n,...} = n
```

and then context is used to resolve the "local" ambiguity. This works well, provided that the complete record type of the arguments to `#name` can be determined from context. If not, the uses are rejected as ambiguous. Thus, the following expression is well-typed

```
fn r : {name:string,address:string,salary:int} =>
   (#name r, #address r)
```

because the record type of `r` is explicitly given. If the type of `r` were omitted, the expression would be rejected as ambiguous (unless the context resolves the ambiguity.)

8.4 Sample Code

Here is the code for this chapter.

Chapter 9

Programming with Lists

9.1 List Primitives

In chapter 5 we noted that aggregate data structures are especially easy to handle in ML. In this chapter we consider another important aggregate type, the *list* type. In addition to being an important form of aggregate type it also illustrates two other general features of the ML type system:

1. *Type constructors*, or *parameterized types*. The type of a list reveals the type of its elements.

2. *Recursive types*. The set of values of a list type are given by an inductive definition.

Informally, the values of type *typ* `list` are the finite lists of values of type *typ*. More precisely, the values of type *typ* `list` are given by an *inductive definition*, as follows:

1. `nil` is a value of type *typ* `list`.

2. if h is a value of type *typ*, and t is a value of type *typ* `list`, then $h::t$ is a value of type *typ* `list`.

3. Nothing else is a value of type *typ* `list`.

The type expression *typ* `list` is a postfix notation for the application of the *type constructor* `list` to the type *typ*. Thus `list` is a kind of "function" mapping types to types: given a type *typ*, we may apply `list` to it

to get another type, written *typ* `list`. The forms `nil` and `::` are the *value constructors* of type *typ* `list`. The nullary (no argument) constructor `nil` may be thought of as the empty list. The binary (two argument) constructor `::` constructs a non-empty list from a value h of type *typ* and another value t of type *typ* `list`; the resulting value, h`::`t, of type *typ* `list`, is pronounced "h cons t" (for historical reasons). We say that "h is *cons'd onto* t", that h is the *head* of the list, and that t is its *tail*.

The definition of the values of type *typ* `list` given above is an example of an *inductive definition*. The type is said to be *recursive* because this definition is "self-referential" in the sense that the values of type *typ* `list` are defined in terms of (other) values of the same type. This is especially clear if we examine the types of the value constructors for the type *typ* `list`:

```
val nil : typ list
val (op ::) : typ * typ list -> typ list
```

The notation `op ::` is used to *refer* to the `::` operator as a function, rather than to *use* it to form a list, which requires infix notation.

Two things are notable here:

1. The `::` operation takes as its second argument a value of type *typ* `list`, and yields a result of type *typ* `list`. This self-referential aspect is characteristic of an inductive definition.

2. Both `nil` and `op ::` are *polymorphic* in the type of the underlying elements of the list. Thus `nil` is the empty list of type *typ* `list` for any element type *typ*, and `op ::` constructs a non-empty list independently of the type of the elements of that list.

It is easy to see that a value *val* of type *typ* `list` has the form

val_1`::(`val_2`:: (`$\cdots$ `::(`val_n`::nil)`$\cdots$`))`

for some $n \geq 0$, where val_i is a value of type typ_i for each $1 \leq i \leq n$. For according to the inductive definition of the values of type *typ* `list`, the value *val* must either be `nil`, which is of the above form, or val_1`::`val', where val' is a value of type *typ* `list`. By induction val' has the form

`(`val_2`:: (`$\cdots$ `::(`val_n`::nil)`$\cdots$`))`

and hence *val* again has the specified form.

By convention the operator `::` is *right-associative*, so we may omit the parentheses and just write

val_1`::`val_2`::`$\cdots$`::`val_n`::nil`

as the general form of *val* of type *typ* `list`. This may be further abbreviated using *list notation*, writing

`[` val_1`,` val_2`, ...,` val_n `]`

for the same list. This notation emphasizes the interpretation of lists as finite sequences of values, but it obscures the fundamental inductive character of lists as being built up from `nil` using the `::` operation.

9.2 Computing With Lists

How do we compute with values of list type? Since the values are defined inductively, it is natural that functions on lists be defined recursively, using a clausal definition that analyzes the structure of a list. Here's a definition of the function `length` that computes the number of elements of a list:

```
fun length nil = 0
    | length (_::t) = 1 + length t
```

The definition is given by induction on the structure of the list argument. The base case is the empty list, `nil`. The inductive step is the non-empty list `_::t` (notice that we do not need to give a name to the head). Its definition is given in terms of the tail of the list `t`, which is "smaller" than the list `_::t`. The type of `length` is `'a list -> int`; it is defined for lists of values of any type whatsoever.

We may define other functions following a similar pattern. Here's the function to append two lists:

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append (t, l)
```

This function is built into ML; it is written using infix notation as exp_1 `@` exp_2. The running time of `append` is proportional to the length of the first list, as should be obvious from its definition.

Here's a function to reverse a list.

```
fun rev nil = nil
  | rev (h::t) = rev t @ [h]
```

Its running time is $O(n^2)$, where n is the length of the argument list. This can be demonstrated by writing down a recurrence that defines the running time $T(n)$ on a list of length n.

$$\begin{aligned} T(0) &= O(1) \\ T(n+1) &= T(n) + O(n) \end{aligned}$$

Solving the recurrence we obtain the result $T(n) = O(n^2)$.

Can we do better? Oddly, we can take advantage of the *non-associativity* of `::` to give a tail-recursive definition of `rev`.

```
local
    fun helper (nil, a) = a
      | helper (h::t, a) = helper (t, h::a)
in
    fun rev' l = helper (l, nil)
end
```

The general idea of introducing an accumulator is the same as before, except that by re-ordering the applications of `::` we reverse the list! The helper function reverses its first argument and prepends it to its second argument. That is, `helper (l, a)` evaluates to `(rev l) @ a`, where we assume here an independent definition of `rev` for the sake of the specification. Notice that `helper` runs in time proportional to the length of its first argument, and hence `rev'` runs in time proportional to the length of its argument.

The correctness of functions defined on lists may be established using the principle of *structural induction*. We illustrate this by establishing that the function `helper` satisfies the following specification:

> for every l and a of type *typ* `list`, `helper(`l`, `a`)` evaluates to the result of appending a to the reversal of l.

That is, there are no pre-conditions on l and a, and we establish the post-condition that `helper (`l`, `a`)` yields `(rev `l`) @ `a.

The proof is by structural induction on the list l. If l is `nil`, then `helper (`l`,`a`)` evaluates to a, which fulfills the post-condition. If l is the list h`::`t,

then the application `helper (`l`, `a`)` reduces to the value of `helper (`t`, (`h`::`a`))`. By the inductive hypothesis this is just `(rev `t`) @ (`h` :: `a`)`, which is equivalent to `(rev `t`) @ [`h`] @ `a. But this is just `rev (`h`::`t`) @ `a, which was to be shown.

The principle of structural induction may be summarized as follows. To show that a function works correctly for every list l, it suffices to show

1. The correctness of the function for the empty list, `nil`, and
2. The correctness of the function for $h::t$, assuming its correctness for t.

As with mathematical induction over the natural numbers, structural induction over lists allows us to focus on the *basic* and *incremental* behavior of a function to establish its correctness for all lists.

9.3 Sample Code

Here is the code for this chapter.

Chapter 10

Concrete Data Types

10.1 Datatype Declarations

Lists are one example of the general notion of a *recursive type*. ML provides a general mechanism, the `datatype` declaration, for introducing programmer-defined recursive types. Earlier we introduced `type` declarations as an abbreviation mechanism. Types are given names as documentation and as a convenience to the programmer, but doing so is semantically inconsequential — one could replace all uses of the type name by its definition and not affect the behavior of the program. In contrast the `datatype` declaration provides a means of introducing a *new* type that is distinct from all other types and that does not merely stand for some other type. It is the means by which the ML type system may be extended by the programmer.

The `datatype` declaration in ML has a number of facets. A `datatype` declaration introduces

1. One or more new type constructors. The type constructors introduced may, or may not, be mutually recursive.

2. One or more new value constructors for each of the type constructors introduced by the declaration.

The type constructors may take zero or more arguments; a zero-argument, or *nullary*, type constructor is just a type. Each value constructor may also take zero or more arguments; a nullary value constructor is just a constant. The type and value constructors introduced by the declaration are "new" in the sense that they are distinct from all other type and value

constructors previously introduced; if a datatype re-defines an "old" type or value constructor, then the old definition is shadowed by the new one, rendering the old ones inaccessible in the scope of the new definition.

10.2 Non-Recursive Datatypes

Here's a simple example of a nullary type constructor with four nullary value constructors.

```
datatype suit = Spades | Hearts | Diamonds | Clubs
```

This declaration introduces a new type `suit` with four nullary value constructors, `Spades`, `Hearts`, `Diamonds`, and `Clubs`. This declaration may be read as introducing a type `suit` such that a value of type `suit` is either `Spades`, or `Hearts`, or `Diamonds`, or `Clubs`. There is no significance to the ordering of the constructors in the declaration; we could just as well have written

```
datatype suit = Hearts | Diamonds | Spades | Clubs
```

(or any other ordering, for that matter). It is conventional to capitalize the names of value constructors, but this is not required by the language.

Given the declaration of the type `suit`, we may define functions on it by case analysis on the value constructors using a clausal function definition. For example, we may define the suit ordering in the card game of bridge by the function

```
fun outranks (Spades, Spades) = false
  | outranks (Spades, _) = true
  | outranks (Hearts, Spades) = false
  | outranks (Hearts, Hearts) = false
  | outranks (Hearts, _) = true
  | outranks (Diamonds, Clubs) = true
  | outranks (Diamonds, _) = false
  | outranks (Clubs, _) = false
```

This defines a function of type `suit * suit -> bool` that determines whether or not the first `suit` outranks the second.

Data types may be *parameterized* by a type. For example, the declaration

```
datatype 'a option = NONE | SOME of 'a
```

introduces the unary type constructor `'a option` with two value constructors, `NONE`, with no arguments, and `SOME`, with one. The values of type *typ* `option` are

1. The constant `NONE`, and
2. Values of the form `SOME` *val*, where *val* is a value of type *typ*.

For example, some values of type `string option` are `NONE`, `SOME "abc"`, and `SOME "def"`.

The option type constructor is pre-defined in Standard ML. One common use of option types is to handle functions with an optional argument. For example, here is a function to compute the base-*b* exponential function for natural number exponents that defaults to base 2:

```
fun expt (NONE, n) = expt (SOME 2, n)
  | expt (SOME b, 0) = 1
  | expt (SOME b, n) =
    if n mod 2 = 0 then
       expt (SOME (b*b), n div 2)
    else
       b * expt (SOME b, n-1)
```

The advantage of the option type in this sort of situation is that it avoids the need to make a special case of a particular argument, *e.g.*, using `0` as first argument to mean "use the default exponent".

A related use of option types is in aggregate data structures. For example, an address book entry might have a record type with fields for various bits of data about a person. But not all data is relevant to all people. For example, someone may not have a spouse, but they all have a name. For this we might use a type definition of the form

```
type entry = { name:string, spouse:string option }
```

so that one would create an entry for an unmarried person with a `spouse` field of `NONE`.

Option types may also be used to represent an optional result. For example, we may wish to define a function `reciprocal` that returns the reciprocal of an integer, if it has one, and otherwise indicates that it has

no reciprocal. This is achieve by defining `reciprocal` to have type `int -> int option` as follows:

```
fun reciprocal 0 = NONE
  | reciprocal n = SOME (1 div n)
```

To use the result of a call to *reciprocal* we must perform a case analysis of the form

```
case (reciprocal exp
  of NONE => exp1
   | SOME r => exp2
```

where exp_1 covers the case that *exp* has no reciprocal, and exp_2 covers the case that *exp* has reciprocal `r`.

10.3 Recursive Datatypes

The next level of generality is the recursive type definition. For example, one may define a type *typ* `tree` of binary trees with values of type *typ* at the nodes using the following declaration:

```
datatype 'a tree =
  Empty |
  Node of 'a tree * 'a * 'a tree
```

This declaration corresponds to the informal definition of binary trees with values of type *typ* at the nodes:

1. The empty tree `Empty` is a binary tree.
2. If *tree_1* and *tree_2* are binary trees, and *val* is a value of type *typ*, then `Node (`*tree_1*`,` *val*`,` *tree_2*`)` is a binary tree.
3. Nothing else is a binary tree.

The distinguishing feature of this definition is that it is *recursive* in the sense that binary trees are constructed out of other binary trees, with the empty tree serving as the base case.

(Incidentally, a *leaf* in a binary tree is here represented as a node both of whose children are the empty tree. This definition of binary trees is analogous to starting the natural numbers with zero, rather than one. One can think of the children of a node in a binary tree as the "predecessors" of that node, the only difference compared to the usual definition of predecessor being that a node has two, rather than one, predecessors.)

To compute with a recursive type, use a recursive function. For example, here is the function to compute the *height* of a binary tree:

```
fun height Empty = 0
  | height (Node (lft, _, rht)) =
    1 + max (height lft, height rht)
```

Notice that `height` is called recursively on the children of a node, and is defined outright on the empty tree. This pattern of definition is another instance of structural induction (on the `tree` type). The function `height` is said to be defined by induction on the structure of a tree. The general idea is to define the function directly for the base cases of the recursive type (*i.e.*, value constructors with no arguments or whose arguments do not involve values of the type being defined), and to define it for non-base cases in terms of its definitions for the constituent values of that type. We will see numerous examples of this as we go along.

Here's another example. The *size* of a binary tree is the number of nodes occurring in it. Here's a straightforward definition in ML:

```
fun size Empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The function `size` is defined by structural induction on trees.

A word of warning. One reason to capitalize value constructors is to avoid a pitfall in the ML syntax that we mentioned in chapter 2. Suppose we gave the following definition of `size`:

```
fun size empty = 0
  | size (Node (lft, _, rht)) =
    1 + size lft + size rht
```

The compiler will warn us that the second clause of the definition is *redundant*! Why? Because `empty`, spelled with a lower-case "e", is a *variable*, not

a *constructor*, and hence matches *any* tree whatsoever. Consequently the second clause never applies. By capitalizing constructors we can hope to make mistakes such as these more evident, but in practice you are bound to run into this sort of mistake.

The `tree` data type is appropriate for binary trees: those for which each node has exactly two children. (Of course, either or both children might be the empty tree, so we may consider this to define the type of trees with *at most* two children; it's a matter of terminology which interpretation you prefer.) It should be obvious how to define the type of *ternary* trees, whose nodes have at most three children, and so on for other fixed arities. But what if we wished to define a type of trees with a *variable* number of children? In a so-called *variadic tree* some nodes might have three children, some might have two, and so on. This can be achieved in at least two ways. One way combines lists and trees, as follows:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a tree list
```

Each node has a *list* of children, so that distinct nodes may have different numbers of children. Notice that the empty tree is distinct from the tree with one node and no children because there is no data associated with the empty tree, whereas there is a value of type `'a` at each node.

Another approach is to simultaneously define trees and "forests". A variadic tree is either empty, or a node gathering a "forest" to form a tree; a forest is either empty or a variadic tree together with another forest. This leads to the following definition:

```
datatype 'a tree =
  Empty |
  Node of 'a * 'a forest
and 'a forest =
  None |
  Tree of 'a tree * 'a forest
```

This example illustrates the introduction of two *mutually recursive datatypes*.

Mutually recursive datatypes beget mutually recursive functions. Here's a definition of the size (number of nodes) of a variadic tree:

```
fun size_tree Empty = 0
  | size_tree (Node (_, f)) = 1 + size_forest f
and size_forest None = 0
  | size_forest (Tree (t, f')) = size_tree t + size_forest f'
```

Notice that we define the size of a tree in terms of the size of a forest, and *vice versa*, just as the type of trees is defined in terms of the type of forests.

Many other variations are possible. Suppose we wish to define a notion of binary tree in which data items are associated with branches, rather than nodes. Here's a `datatype` declaration for such trees:

```
datatype 'a tree =
    Empty |
    Node of 'a branch * 'a branch
and 'a branch =
    Branch of 'a * 'a tree
```

In contrast to our first definition of binary trees, in which the branches from a node to its children were *implicit*, we now make the branches themselves *explicit*, since data is attached to them.

For example, we can collect into a list the data items labelling the branches of such a tree using the following code:

```
fun collect Empty = nil
  | collect (Node (Branch (ld, lt), Branch (rd, rt))) =
    ld :: rd :: (collect lt) @ (collect rt)
```

10.4 Heterogeneous Data Structures

Returning to the original definition of binary trees (with data items at the nodes), observe that the *type* of the data items at the nodes must be the same for every node of the tree. For example, a value of type `int tree` has an integer at every node, and a value of type string tree has a string at every node. Therefore an expression such as

```
Node (Empty, 43, Node (Empty, "43", Empty))
```

is ill-typed. The type system insists that trees be *homogeneous* in the sense that the type of the data items is the same at every node.

It is quite rare to encounter heterogeneous data structures in real programs. For example, a dictionary with strings as keys might be represented as a binary search tree with strings at the nodes; there is no need for heterogeneity to represent such a data structure. But occasionally one might wish to work with a *heterogeneous* tree, whose data values at each node are of different types. How would one represent such a thing in ML?

To discover the answer, first think about how one might manipulate such a data structure. When accessing a node, we would need to check at run-time whether the data item is an integer or a string; otherwise we would not know whether to, say, add 1 to it, or concatenate `"1"` to the end of it. This suggests that the data item must be *labelled* with sufficient information so that we may determine the type of the item at run-time. We must also be able to recover the underlying data item itself so that familiar operations (such as addition or string concatenation) may be applied to it.

The required labelling and discrimination is neatly achieved using a datatype declaration. Suppose we wish to represent the type of integer-or-string trees. First, we define the type of values to be integers or strings, marked with a constructor indicating which:

```
datatype int_or_string =
  Int of int |
  String of string
```

Then we define the type of interest as follows:

```
type int_or_string_tree =
  int_or_string tree
```

Voila! Perfectly natural and easy — heterogeneity is really a special case of homogeneity!

10.5 Abstract Syntax

`Datatype` declarations and pattern matching are extremely useful for defining and manipulating the *abstract syntax* of a language. For example, we may define a small language of arithmetic expressions using the following declaration:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr
```

This definition has only three clauses, but one could readily imagine adding others. Here is the definition of a function to evaluate expressions of the language of arithmetic expressions written using pattern matching:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1+n2)
    end
  | eval (Times (e1, e2)) =
    let
        val Numeral n1 = eval e1
        val Numeral n2 = eval e2
    in
        Numeral (n1*n2)
    end
```

The combination of `datatype` declarations and pattern matching contributes enormously to the readability of programs written in ML. A less obvious, but more important, benefit is the error checking that the compiler can perform for you if you use these mechanisms in tandem. As an example, suppose that we extend the type `expr` with a new component for the reciprocal of a number, yielding the following revised definition:

```
datatype expr =
  Numeral of int |
  Plus of expr * expr |
  Times of expr * expr |
  Recip of expr
```

First, observe that the "old" definition of `eval` is no longer applicable to values of type `expr`! For example, the expression

```
eval (Plus (Numeral 1, Numeral 2))
```

is ill-typed, even though it doesn't use the `Recip` constructor. The reason is that the re-declaration of `expr` introduces a "new" type that just happens to have the same name as the "old" type, but is in fact distinct from it. This is a boon because it reminds us to recompile the old code relative to the new definition of the `expr` type.

Second, upon recompiling the definition of `eval` we encounter an *inexhaustive match* warning: the old code no longer applies to every value of type `expr` according to its new definition! We are of course lacking a case for `Recip`, which we may provide as follows:

```
fun eval (Numeral n) = Numeral n
  | eval (Plus (e1, e2)) = ... as before ...
  | eval (Times (e1, e2)) = ... as before ...
  | eval (Recip e) =
    let
        val Numeral n = eval e
    in
        Numeral (1 div n)
    end
```

The value of the checks provided by the compiler in such cases cannot be overestimated. When recompiling a large program after making a change to a `datatype` declaration the compiler will automatically point out *every line of code* that must be changed to conform to the new definition; it is impossible to forget to attend to even a single case. This is a tremendous help to the developer, especially if she is not the original author of the code being modified and is another reason why the static type discipline of ML is a positive benefit, rather than a hindrance, to programmers.

10.6 Sample Code

Here is the code for this chapter.

Chapter 11

Higher-Order Functions

11.1 Functions as Values

Values of function type are *first-class*, which means that they have the same rights and privileges as values of any other type. In particular, functions may be passed as arguments and returned as results of other functions, and functions may be stored in and retrieved from data structures such as lists and trees. We will see that first-class functions are an important source of expressive power in ML.

Functions which take functions as arguments or yield functions as results are known as *higher-order functions* (or, less often, as *functionals* or *operators*). Higher-order functions arise frequently in mathematics. For example, the differential operator is the higher-order function that, when given a (differentiable) function on the real line, yields its first derivative as a function on the real line. We also encounter functionals mapping functions to real numbers, and real numbers to functions. An example of the former is provided by the definite integral viewed as a function of its integrand, and an example of the latter is the definite integral of a given function on the interval $[a, x]$, viewed as a function of a, that yields the area under the curve from a to x as a function of x.

Higher-order functions are less familiar tools for many programmers since the best-known programming languages have only rudimentary mechanisms to support their use. In contrast higher-order functions play a prominent role in ML, with a variety of interesting applications. Their use may be classified into two broad categories:

1. *Abstracting patterns of control.* Higher-order functions are design patterns that "abstract out" the details of a computation to lay bare the skeleton of the solution. The skeleton may be fleshed out to form a solution of a problem by applying the general pattern to arguments that isolate the specific problem instance.

2. *Staging computation.* It arises frequently that computation may be *staged* by expending additional effort "early" to simplify the computation of "later" results. Staging can be used both to improve efficiency and, as we will see later, to control sharing of computational resources.

11.2 Binding and Scope

Before discussing these programming techniques, we will review the critically important concept of *scope* as it applies to function definitions. Recall that Standard ML is a *statically scoped* language, meaning that identifiers are resolved according to the static structure of the program. A *use* of the variable *var* is considered to be a reference to the *nearest lexically enclosing declaration of var*. We say "nearest" because of the possibility of shadowing; if we re-declare a variable *var*, then subsequent uses of *var* refer to the "most recent" (lexically!) declaration of it; any "previous" declarations are temporarily shadowed by the latest one.

This principle is easy to apply when considering sequences of declarations. For example, it should be clear by now that the variable `y` is bound to `32` after processing the following sequence of declarations:

```
val x = 2           (* x=2 *)
val y = x*x         (* y=4 *)
val x = y*x         (* x=8 *)
val y = x*y         (* y=32 *)
```

In the presence of function definitions the situation is the same, but it can be a bit tricky to understand at first.

Here's an example to test your grasp of the lexical scoping principle:

```
val x = 2
fun f y = x+y
val x = 3
val z = f 4
```

After processing these declarations the variable `z` is bound to `6`, not to `7`! The reason is that the occurrence of `x` in the body of `f` refers to the *first* declaration of `x` since it is the nearest lexically enclosing declaration of the occurence, *even though* it has been subsequently re-declared.

This example illustrates three important points:

1. Binding is not assignment! If we were to view the second binding of `x` as an assignment statement, then the value of `z` would be `7`, not `6`.

2. Scope resolution is *lexical*, not *temporal.* We sometimes refer to the "most recent" declaration of a variable, which has a temporal flavor, but we always mean "nearest lexically enclosing at the point of occurrence".

3. "Shadowed" bindings are not lost. The "old" binding for `x` is still available (through calls to `f`), even though a more recent binding has shadowed it.

One way to understand what's going on here is through the concept of a *closure*, a technique for implementing higher-order functions. When a function expression is evaluated, a copy of the environment is attached to the function. Subsequently, all free variables of the function (*i.e.*, those variables not occurring as parameters) are resolved with respect to the environment attached to the function; the function is therefore said to be "closed" with respect to the attached environment. This is achieved at function application time by "swapping" the attached environment of the function for the environment active at the point of the call. The swapped environment is restored after the call is complete. Returning to the example above, the environment associated with the function `f` contains the declaration `val x = 2` to record the fact that at the time the function was evaluated, the variable `x` was bound to the value `2`. The variable `x` is subsequently re-bound to `3`, but when `f` is applied, we temporarily reinstate the binding of `x` to `2`, add a binding of `y` to `4`, then evaluate the body of the function, yielding `6`. We then restore the binding of `x` and drop the binding of `y` before yielding the result.

11.3 Returning Functions

While seemingly very simple, the principle of lexical scope is the source of considerable expressive power. We'll demonstrate this through a series of examples.

To warm up let's consider some simple examples of passing functions as arguments and yielding functions as results. The standard example of passing a function as argument is the `map'` function, which applies a given function to every element of a list. It is defined as follows:

```
fun map' (f, nil) = nil
  | map' (f, h::t) = (f h) :: map' (f, t)
```

For example, the application

```
map' (fn x => x+1, [1,2,3,4])
```

evaluates to the list `[2,3,4,5]`.

Functions may also yield functions as results. What is surprising is that we can create *new* functions during execution, not just return functions that have been previously defined. The most basic (and deceptively simple) example is the function `constantly` that creates constant functions: given a value `k`, the application `constantly k` yields a function that yields `k` whenever it is applied. Here's a definition of `constantly`:

```
val constantly = fn k => (fn a => k)
```

The function constantly has type `'a -> ('b -> 'a)`. We used the `fn` notation for clarity, but the declaration of the function `constantly` may also be written using `fun` notation as follows:

```
fun constantly k a = k
```

Note well that a *white space* separates the two successive arguments to `constantly`! The meaning of this declaration is precisely the same as the earlier definition using `fn` notation.

The value of the application `constantly 3` is the function that is constantly `3`; *i.e.*, it always yields `3` when applied. Yet nowhere have we defined the function that always yields `3`. The resulting function is "created" by the application of `constantly` to the argument `3`, rather than merely

"retrieved" off the shelf of previously-defined functions. In implementation terms the result of the application `constantly 3` is a closure consisting of the function `fn a => k` with the environment `val k = 3` attached to it. The closure is a data structure (a pair) that is created by each application of `constantly` to an argument; the closure is the representation of the "new" function yielded by the application. Notice, however, that the *only* difference between any two results of applying the function `constantly` lies in the attached environment; the underlying function is *always* `fn a => k`. If we think of the lambda as the "executable code" of the function, then this amounts to the observation that no new *code* is created at run-time, just new *instances* of existing code.

This also points out why functions in ML are not the same as code pointers in C. You may be familiar with the idea of passing a pointer to a C function to another C function as a means of passing functions as arguments or yielding functions as results. This may be considered to be a form of "higher-order" function in C, but it must be emphasized that code pointers are significantly less powerful than closures because in C there are only *statically many* possibilities for a code pointer (it must point to one of the functions defined in your code), whereas in ML we may generate *dynamically many* different instances of a function, differing in the bindings of the variables in its environment. The non-varying part of the closure, the code, is directly analogous to a function pointer in C, but there is no counterpart in C of the varying part of the closure, the dynamic environment.

The definition of the function `map'` given above takes a function and list as arguments, yielding a new list as result. Often it occurs that we wish to map the same function across several different lists. It is inconvenient (and a tad inefficient) to keep passing the same function to `map'`, with the list argument varying each time. Instead we would prefer to create a instance of map specialized to the given function that can then be applied to many different lists. This leads to the following definition of the function `map`:

```
fun map f nil = nil
  | map f (h::t) = (f h) :: (map f t)
```

The function `map` so defined has type `('a->'b) -> 'a list -> 'b list`. It takes a function of type `'a -> 'b` as argument, and yields another function of type `'a list -> 'b list` as result.

The passage from `map'` to `map` is called *currying*. We have changed a two-argument function (more properly, a function taking a pair as argument) into a function that takes two arguments in succession, yielding after the first a function that takes the second as its sole argument. This passage can be codified as follows:

```
fun curry f x y = f (x, y)
```

The type of `curry` is

```
('a*'b->'c) -> ('a -> ('b -> 'c)).
```

Given a two-argument function, `curry` returns another function that, when applied to the first argument, yields a function that, when applied to the second, applies the original two-argument function to the first and second arguments, given separately.

Observe that `map` may be alternately defined by the binding

```
fun map f l = curry map' f l
```

Applications are implicitly left-associated, so that this definition is equivalent to the more verbose declaration

```
fun map f l = ((curry map') f) l
```

11.4 Patterns of Control

We turn now to the idea of abstracting patterns of control. There is an obvious similarity between the following two functions, one to add up the numbers in a list, the other to multiply them.

```
fun add_up nil = 0
  | add_up (h::t) = h + add_up t
fun mul_up nil = 1
  | mul_up (h::t) = h * mul_up t
```

What precisely is the similarity? We will look at it from two points of view.

One view is that in each case we have a binary operation and a unit element for it. The result on the empty list is the unit element, and the result on a non-empty list is the operation applied to the head of the list and the result on the tail. This pattern can be abstracted as the function `reduce` defined as follows:

```
fun reduce (unit, opn, nil) =
    unit
  | reduce (unit, opn, h::t) =
    opn (h, reduce (unit, opn, t))
```

Here is the type of `reduce`:

```
val reduce : 'b * ('a*'b->'b) * 'a list -> 'b
```

The first argument is the unit element, the second is the operation, and the third is the list of values. Notice that the type of the operation admits the possibility of the first argument having a different type from the second argument and result.

Using reduce, we may re-define `add_up` and `mul_up` as follows:

```
fun add_up l = reduce (0, op +, l)
fun mul_up l = reduce (1, op *, l)
```

To further check your understanding, consider the following declaration:

```
fun mystery l = reduce (nil, op ::, l)
```

(Recall that "`op ::`" is the function of type `'a * 'a list -> 'a list` that adds a given value to the front of a list.) What function does `mystery` compute?

Another view of the commonality between `add_up` and `mul_up` is that they are both defined by induction on the structure of the list argument, with a base case for `nil`, and an inductive case for `h::t`, defined in terms of its behavior on `t`. But this is really just another way of saying that they are defined in terms of a unit element and a binary operation! The difference is one of perspective: whether we focus on the pattern part of the clauses (the inductive decomposition) or the result part of the clauses (the unit and operation). The recursive structure of `add_up` and `mul_up` is abstracted by the `reduce` functional, which is then specialized to yield `add_up` and `mul_up`. Said another way, *the function reduce abstracts the pattern of defining a function by induction on the structure of a list*.

The definition of `reduce` leaves something to be desired. One thing to notice is that the arguments `unit` and `opn` are carried unchanged through the recursion; only the list parameter changes on recursive calls. While this might seem like a minor overhead, it's important to remember that

multi-argument functions are really single-argument functions that take a tuple as argument. This means that each time around the loop we are constructing a new tuple whose first and second components remain fixed, but whose third component varies. Is there a better way? Here's another definition that isolates the "inner loop" as an auxiliary function:

```
fun better_reduce (unit, opn, l) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red l
    end
```

Notice that each call to `better_reduce` creates a *new* function `red` that uses the parameters `unit` and `opn` of the call to `better_reduce`. This means that `red` is bound to a closure consisting of the code for the function together with the environment active at the point of definition, which will provide bindings for `unit` and `opn` arising from the application of `better_reduce` to its arguments. Furthermore, the recursive calls to `red` no longer carry bindings for `unit` and `opn`, saving the overhead of creating tuples on each iteration of the loop.

11.5 Staging

An interesting variation on `reduce` may be obtained by *staging* the computation. The motivation is that `unit` and `opn` often remain fixed for many different lists (*e.g.*, we may wish to sum the elements of many different lists). In this case `unit` and `opn` are said to be "early" arguments and the list is said to be a "late" argument. The idea of staging is to perform as much computation as possible on the basis of the early arguments, yielding a function of the late arguments alone.

In the case of the function `reduce` this amounts to building `red` on the basis of `unit` and `opn`, yielding it as a function that may be later applied to many different lists. Here's the code:

```
fun staged_reduce (unit, opn) =
    let
        fun red nil = unit
          | red (h::t) = opn (h, red t)
    in
        red
    end
```

The definition of `staged_reduce` bears a close resemblance to the definition of `better_reduce`; the only difference is that the creation of the closure bound to `red` occurs *as soon as* `unit` *and* `opn` *are known*, rather than each time the list argument is supplied. Thus the overhead of closure creation is "factored out" of multiple applications of the resulting function to list arguments.

We could just as well have replaced the body of the let expression with the function

```
fn l => red l
```

but a moment's thought reveals that the meaning is the same.

Note well that we would *not* obtain the effect of staging were we to use the following definition:

```
fun curried_reduce (unit, opn) nil = unit
  | curried_reduce (unit, opn) (h::t) =
    opn (h, curried_reduce (unit, opn) t)
```

If we unravel the `fun` notation, we see that while we are taking two arguments in succession, we are *not* doing any useful work in between the arrival of the first argument (a pair) and the second (a list). A *curried* function does not take significant advantage of staging. Since `staged_reduce` and `curried_reduce` have the same iterated function type, namely

```
('b * ('a * 'b -> 'b)) -> 'a list -> 'b
```

the contrast between these two examples may be summarized by saying *not every function of iterated function type is curried*. Some are, and some aren't. The "interesting" examples (such as `staged_reduce`) are the ones that *aren't* curried. (This directly contradicts established terminology, but it is necessary to deviate from standard practice to avoid a serious misapprehension.)

The time saved by staging the computation in the definition of `staged_reduce` is admittedly minor. But consider the following definition of an append function for lists that takes both arguments at once:

```
fun append (nil, l) = l
  | append (h::t, l) = h :: append(t,l)
```

Suppose that we will have occasion to append many lists to the end of a given list. What we'd like is to build a specialized appender for the first list that, when applied to a second list, appends the second to the end of the first. Here's a naive solution that merely curries append:

```
fun curried_append nil l = l
  | curried_append (h::t) l = h :: curried_append t l
```

Unfortunately this solution doesn't exploit the fact that the first argument is fixed for many second arguments. In particular, each application of the result of applying `curried_append` to a list results in the first list being traversed so that the second can be appended to it.

We can improve on this by staging the computation as follows:

```
fun staged_append nil = fn l => l
  | staged_append (h::t) =
    let
        val tail_appender = staged_append t
    in
        fn l => h :: tail_appender l
    end
```

Notice that the first list is traversed *once* for all applications to a second argument. When applied to a list $[v_1, \ldots, v_n]$, the function `staged_append` yields a function that is equivalent to, but not quite as efficient as, the function

`fn l =>` v_1 `::` v_2 `:: ... ::` v_n `:: l.`

This still takes time proportional to n, but a substantial savings accrues from avoiding the pattern matching required to destructure the original list argument on each call.

11.6 Sample Code

Here is the code for this chapter.

Chapter 12

Exceptions

In the first chapter of these notes we mentioned that expressions in Standard ML always have a type, may have a value, and may have an effect. So far we've concentrated on typing and evaluation. In this chapter we will introduce the concept of an *effect*. While it's hard to give a precise general definition of what we mean by an effect, the idea is that an effect is any action resulting from evaluation of an expression other than returning a value. From this point of view we might consider non-termination to be an effect, but we don't usually think of failure to terminate as a positive "action" in its own right, rather as a failure to take any action.

The main examples of effects in ML are these:

1. *Exceptions*. Evaluation may be aborted by signaling an exceptional condition.
2. *Mutation*. Storage may be allocated and modified during evaluation.
3. *Input/output*. It is possible to read from an input source and write to an output sink during evaluation.
4. *Communication*. Data may be sent to and received from communication channels.

This chapter is concerned with exceptions; the other forms of effects will be considered later.

12.1 Exceptions as Errors

ML is a *safe* language in the sense that its execution behavior may be understood entirely in terms of the constructs of the language itself. Behavior such as "dumping core" or incurring a "bus error" are extra-linguistic notions that may only be explained by appeal to the underlying implementation of the language. These cannot arise in ML. This is ensured by a combination of a static type discipline, which rules out expressions that are manifestly ill-defined (*e.g.*, adding a string to an integer or casting an integer as a function), and by dynamic checks that rule out violations that cannot be detected statically (*e.g.*, division by zero or arithmetic overflow). Static violations are signalled by type checking errors; dynamic violations are signalled by *raising exceptions*.

12.1.1 Primitive Exceptions

The expression `3 + "3"` is ill-typed, and hence cannot be evaluated. In contrast the expression `3 div 0` is well-typed (with type `int`), but incurs a run-time fault that is signalled by raising the exception `Div`. We will indicate this by writing

```
3 div 0 ⇓ raise Div
```

An exception is a form of "answer" to the question "what is the value of this expression?". In most implementations an exception such as this is reported by an error message of the form "`Uncaught exception Div`", together with the line number (or some other indication) of the point in the program where the exception occurred.

Exceptions have names so that we may distinguish different sources of error in a program. For example, evaluation of the expression `maxint * maxint` (where `maxint` is the largest representable integer) causes the exception `Overflow` to be raised, indicating that an arithmetic overflow error arose in the attempt to carry out the multiplication. This is usefully distinguished from the exception `Div`, corresponding to division by zero.

(You may be wondering about the overhead of checking for arithmetic faults. The compiler must generate instructions that ensure that an overflow fault is caught before any subsequent operations are performed. This can be quite expensive on pipelined processors, which sacrifice precise delivery of arithmetic faults in the interest of speeding up execution in the

non-faulting case. Unfortunately it is necessary to incur this overhead if we are to avoid having the behavior of an ML program depend on the underlying processor on which it is implemented.)

Another source of run-time exceptions is an inexhaustive match. Suppose we define the function `hd` as follows

```
fun hd (h::_) = h
```

This definition is inexhaustive since it makes no provision for the possibility of the argument being `nil`. What happens if we apply `hd` to `nil`? The exception `Match` is raised, indicating the failure of the pattern-matching process:

```
hd nil ⇓ raise Match
```

The occurrence of a `Match` exception at run-time is indicative of a violation of a pre-condition to the invocation of a function somewhere in the program. Recall that it is often sensible for a function to be inexhaustive, provided that we take care to ensure that it is never applied to a value outside of its domain. Should this occur (because of a programming mistake, evidently), the result is nevertheless well-defined because ML checks for, and signals, pattern match failure. That is, ML programs are implicitly "bullet-proofed" against failures of pattern matching. The flip side is that if no inexhaustive match warnings arise during type checking, then the exception `Match` can never be raised during evaluation (and hence no run-time checking need be performed).

A related situation is the use of a pattern in a `val` binding to destructure a value. If the pattern can fail to match a value of this type, then a `Bind` exception is raised at run-time. For example, evaluation of the binding

```
val h::_ = nil
```

raises the exception `Bind` since the pattern `h::_` does not match the value `nil`. Here again observe that a `Bind` exception cannot arise unless the compiler has previously warned us of the possibility: no warning, no `Bind` exception.

12.1.2 User-Defined Exceptions

So far we have considered examples of pre-defined exceptions that indicate fatal error conditions. Since the built-in exceptions have a built-

in meaning, it is generally inadvisable to use these to signal program-specific error conditions. Instead we introduce a *new* exception using an `exception` declaration, and signal it using a `raise` expression when a run-time violation occurs. That way we can associate specific exceptions with specific pieces of code, easing the process of tracking down the source of the error.

Suppose that we wish to define a "checked factorial" function that ensures that its argument is non-negative. Here's a first attempt at defining such a function:

```
exception Factorial
fun checked_factorial n =
    if n < 0 then
       raise Factorial
    else if n=0 then
       1
    else n * checked_factorial (n-1)
```

The declaration `exception Factorial` introduces an exception `Factorial`, which we raise in the case that `checked_factorial` is applied to a negative number.

The definition of `checked_factorial` is unsatisfactory in at least two respects. One, relatively minor, issue is that it does not make effective use of pattern matching, but instead relies on explicit comparison operations. To some extent this is unavoidable since we wish to check explicitly for negative arguments, which cannot be done using a pattern. A more significant problem is that `checked_factorial` *repeatedly* checks the validity of its argument on each recursive call, even though we can prove that if the initial argument is non-negative, then so must be the argument on each recursive call. This fact is not reflected in the code. We can improve the definition by introducing an auxiliary function:

```
exception Factorial
local
   fun fact 0 = 1
     | fact n = n * fact (n-1)
in
   fun checked_factorial n =
       if n >= 0 then
```

```
          fact n
       else
          raise Factorial
end
```

Notice that we perform the range check exactly once, and that the auxiliary function makes effective use of pattern-matching.

12.2 Exception Handlers

The use of exceptions to signal error conditions suggests that raising an exception is fatal: execution of the program terminates with the raised exception. But signaling an error is only one use of the exception mechanism. More generally, exceptions can be used to effect *non-local transfers of control*. By using an *exception handler* we may "catch" a raised exception and continue evaluation along some other path. A very simple example is provided by the following driver for the factorial function that accepts numbers from the keyboard, computes their factorial, and prints the result.

```
fun factorial_driver () =
    let
        val input = read_integer ()
        val result =
            toString (checked_factorial input)
    in
        print result
    end
    handle Factorial => print "Out of range."
```

An expression of the form *exp* `handle` *match* is called an *exception handler*. It is evaluated by attempting to evaluate *exp*. If it returns a value, then that is the value of the entire expression; the handler plays no role in this case. If, however, *exp* raises an exception *exc*, then the exception value is matched against the clauses of the match (exactly as in the application of a clausal function to an argument) to determine how to proceed. If the pattern of a clause matches the exception *exc*, then evaluation resumes with the expression part of that clause. If no pattern matches, the exception

exc is *re-raised* so that outer exception handlers may dispatch on it. If no handler handles the exception, then the uncaught exception is signaled as the final result of evaluation. That is, computation is aborted with the uncaught exception *exc*.

In more operational terms, evaluation of *exp* `handle` *match* proceeds by installing an exception handler determined by *match*, then evaluating *exp*. The previous binding of the exception handler is preserved so that it may be restored once the given handler is no longer needed. Raising an exception consists of passing a value of type `exn` to the current exception handler. Passing an exception to a handler de-installs that handler, and re-installs the previously active handler. This ensures that if the handler itself raises an exception, or fails to handle the given exception, then the exception is propagated to the handler active prior to evaluation of the `handle` expression. If the expression does not raise an exception, the previous handler is restored as part of completing the evaluation of the `handle` expression.

Returning to the function `factorial_driver`, we see that evaluation proceeds by attempting to compute the factorial of a given number (read from the keyboard by an unspecified function `read_integer`), printing the result if the given number is in range, and otherwise reporting that the number is out of range. The example is trivialized to focus on the role of exceptions, but one could easily imagine generalizing it in a number of ways that also make use of exceptions. For example, we might repeatedly read integers until the user terminates the input stream (by typing the end of file character). Termination of input might be signaled by an `EndOfFile` exception, which is handled by the driver. Similarly, we might expect that the function `read_integer` raises the exception `SyntaxError` in the case that the input is not properly formatted. Again we would handle this exception, print a suitable message, and resume.

Here's a sketch of a more complicated factorial driver:

```
fun factorial_driver () =
    let
        val input = read_integer ()
        val result =
            toString (checked_factorial input)
        val _ = print result
    in
```

```
        factorial_driver ()
    end
    handle EndOfFile => print "Done."
         | SyntaxError =>
           let
               val _ = print "Syntax error."
           in
               factorial_driver ()
           end
         | Factorial =>
           let
               val _ = print "Out of range."
           in
               factorial_driver ()
           end
```

We will return to a more detailed discussion of input/output later in these notes. The point to notice here is that the code is structured with a completely uncluttered "normal path" that reads an integer, computes its factorial, formats it, prints it, and repeats. The exception handler takes care of the exceptional cases: end of file, syntax error, and domain error. In the latter two cases we report an error, and resume reading. In the former we simply report completion and we are done.

The reader is encouraged to imagine how one might structure this program *without* the use of exceptions. The primary benefits of the exception mechanism are as follows:

1. They *force* you to consider the exceptional case (if you don't, you'll get an uncaught exception at run-time), and
2. They *allow* you to segregate the special case from the normal case in the code (rather than clutter the code with explicit checks).

These aspects work hand-in-hand to facilitate writing robust programs.

A typical use of exceptions is to implement *backtracking*, a programming technique based on exhaustive search of a state space. A very simple, if somewhat artificial, example is provided by the following function to compute change from an arbitrary list of coin values. What is at issue is that the obvious "greedy" algorithm for making change that proceeds

by doling out as many coins as possible in decreasing order of value does not always work. Given only a 5 cent and a 2 cent coin, we cannot make 16 cents in change by first taking three 5's and then proceeding to dole out 2's. In fact we must use two 5's and three 2's to make 16 cents. Here's a method that works for any set of coins:

```
exception Change
fun change _ 0 = nil
  | change nil _ = raise Change
  | change (coin::coins) amt =
    if coin > amt then
       change coins amt
    else
       (coin :: change (coin::coins) (amt-coin))
       handle Change => change coins amt
```

The idea is to proceed greedily, but if we get "stuck", we undo the most recent greedy decision and proceed again from there. Simulate evaluation of the example of `change [5,2] 16` to see how the code works.

12.3 Value-Carrying Exceptions

So far exceptions are just "signals" that indicate that an exceptional condition has arisen. Often it is useful to attach additional information that is passed to the exception handler. This is achieved by attaching values to exceptions.

For example, we might associate with a `SyntaxError` exception a string indicating the precise nature of the error. In a parser for a language we might write something like

```
raise SyntaxError "Integer expected"
```

to indicate a malformed expression in a situation where an integer is expected, and write

```
raise SyntaxError "Identifier expected"
```

to indicate a badly-formed identifier.

To associate a string with the exception `SyntaxError`, we declare it as

```
exception SyntaxError of string.
```

This declaration introduces the exception `SyntaxError` as an exception carrying a string as value. This declaration introduces the *exception constructor* `SyntaxError`.

Exception constructors are in many ways similar to value constructors. In particular they can be used in patterns, as in the following code fragment:

```
... handle SyntaxError msg => (print "Syntax error: " ^ msg)
```

Here we specify a pattern for `SyntaxError` exceptions that also binds the string associated with the exception to the identifier `msg` and prints that string along with an error indication.

Recall that we may use value constructors in two ways:

1. We may use them to create values of a datatype (perhaps by applying them to other values).
2. We may use them to match values of a datatype (perhaps also matching a constituent value).

The situation with exception constructors is symmetric.

1. We may use them to create an exception (perhaps with an associated value).
2. We may use them to match an exception (perhaps also matching the associated value).

Value constructors have types, as we previously mentioned. For example, the list constructors `nil` and `::` have types `'a list` and `'a * 'a list -> 'a list`, respectively. What about exception constructors? A "bare" exception constructor (such as `Factorial` above) has type `exn` and a value-carrying exception constructor (such as `SyntaxError`) has type `string -> exn`. Thus `Factorial` is a value of type `exn`, and

```
SyntaxError "Integer expected"
```

is a value of type `exn`.

The type `exn` is the type of *exception packets*, the data values associated with exceptions. The primitive operation `raise` takes any value of type

`exn` as argument and raises an exception with that value. The clauses of a handler may be applied to any value of type `exn` using the rules of pattern matching described earlier; if an exception constructor is no longer in scope, then the handler cannot catch it (other than via a wild-card pattern).

The type `exn` may be thought of as a kind of built-in datatype, *except that* the constructors of this type are not determined once and for all (as they are with a `datatype` declaration), but rather are *incrementally* introduced as needed in a program. For this reason the type `exn` is sometimes called an *extensible datatype.*

12.4 Sample Code

Here is the code for this chapter.

Chapter 13

Mutable Storage

In this chapter we consider a second form of effect, called a *storage effect*, the allocation or mutation of storage during evaluation. The introduction of storage effects has profound consequences, not all of which are desirable. (Indeed, one connotation of the phrase *side effect* is an unintended consequence of a medication!) While it is excessive to dismiss storage effects as completely undesirable, it is advantageous to minimize the use of storage effects except in situations where the task clearly demands them. We will explore some techniques for programming with storage effects later in this chapter, but first we introduce the primitive mechanisms for programming with mutable storage in ML.

13.1 Reference Cells

To support mutable storage the execution model that we described in chapter 2 is enriched with a *memory* consisting of a finite set of *mutable cells*. A mutable cell may be thought of as a container in which a data value of a specified type is stored. During execution of a program the contents of a cell may be retrieved or replaced by any other value of the appropriate type. Since cells are used by issuing "commands" to modify and retrieve their contents, programming with cells is called *imperative programming*.

Changing the contents of a mutable cell introduces a temporal aspect to evaluation. We speak of the *current* contents of a cell, meaning the value *most recently* assigned to it. We also speak of *previous* and *future* values of a reference cell when discussing the behavior of a program. This is in sharp

contrast to the effect-free fragment of ML, for which no such concepts apply. For example, the binding of a variable does not change while evaluating within the scope of that variable, lending a "permanent" quality to statements about variables — the "current" binding is the only binding that variable will ever have.

The type *typ* `ref` is the type of reference cells containing values of type `typ`. Reference cells are, like all values, first class — they may be bound to variables, passed as arguments to functions, returned as results of functions, appear within data structures, and even be stored within other reference cells.

A reference cell is *created*, or *allocated*, by the function `ref` of type *typ* `->` *typ* `ref`. When applied to a value *val* of type *typ*, `ref` allocates a "new" cell, initializes its content to *val*, and returns a reference to the cell. By "new" we mean that the allocated cell is distinct from all other cells previously allocated, and does not share storage with them.

The *contents* of a cell of type *typ* is retrieved using the function `!` of type *typ* `ref ->` *typ*. Applying `!` to a reference cell yields the current contents of that cell. The contents of a cell is changed by applying the assignment operator `op :=`, which has type *typ* `ref * ` *typ* `-> unit`. Assignment is usually written using infix syntax. When applied to a cell and a value, it replaces the content of that cell with that value, and yields the null-tuple as result.

Here are some examples:

```
val r = ref 0
val s = ref 0
val _ = r := 3
val x = !s + !r
val t = r
val _ = t := 5
val y = !s + !r
val z = !t + !r
```

After execution of these bindings, the variable `x` is bound to `3`, the variable `y` is bound to `5`, and `z` is bound to `10`.

Notice the use of a `val` binding of the form `val _ =` *exp* when *exp* is to be evaluated purely for its effect. The value of *exp* is discarded by the binding, since the left-hand side is a wildcard pattern. In most cases the

expression *exp* has type `unit`, so that its value is guaranteed to be the null-tuple, `()`, if it has a value at all.

A wildcard binding is used to define *sequential composition* of expressions in ML. The expression

exp$_1$; *exp*$_2$

is shorthand for the expression

```
let
    val _ = exp1
in
    exp2
end
```

that first evaluates *exp*$_1$ for its effect, then evaluates *exp*$_2$.

Functions of type *typ*`->unit` are sometimes called *procedures*, because they are executed purely for their effect. This is apparent from the type: it is assured that the value of applying such a function is the null-tuple, `()`, so the only point of applying it is for its effects on memory.

13.2 Reference Patterns

It is a common mistake to omit the exclamation point when referring to the content of a reference, especially when that cell is bound to a variable. In more familiar languages such as C all variables are implicitly bound to reference cells, and they are implicitly *de-referenced* whenever they are used so that a variable always stands for its current contents. This is both a boon and a bane. It is obviously helpful in many common cases since it alleviates the burden of having to explicitly dereference variables whenever their content is required. However, it shifts the burden to the programmer in the case that the address, and not the content, is intended. In C one writes `& x` for the address of (the cell bound to) `x`. Whether explicit or implicit de-referencing is preferable is to a large extent a matter of taste. The burden of explicit de-referencing is not nearly so onerous in ML as it might be in other languages simply because reference cells are used so infrequently in ML programs, whereas they are the sole means of binding variables in more familiar languages.

An alternative to explicitly de-referencing cells is to use *ref patterns*. A pattern of the form `ref` *pat* matches a reference cell whose content matches the pattern *pat*. This means that the cell's contents are implicitly retrieved during pattern matching, and may be subsequently used without explicit de-referencing. In fact, the function `!` may be defined using a `ref` pattern as follows:

```
fun !(ref a) = a
```

When called with a reference cell, it is de-referenced and its contents is bound to `a`, which is returned as result. In practice it is common to use both explicit de-referencing and ref patterns, depending on the situation.

13.3 Identity

Reference cells raise delicate issues of equality that considerably complicate reasoning about programs. In general we say that two expressions (of the same type) are equal iff they cannot be distinguished by any operation in the language. That is, two expressions are distinct iff there is some way within the language to tell them apart. This is called *Leibniz's Principle* of *identity of indiscernables* — we equate everything that we cannot tell apart — and the *indiscernability of identicals* — that which we deem equal cannot be told apart.

What makes Leibniz's Principle tricky to grasp is that it hinges on what we mean by a "way to tell expressions apart". The crucial idea is that we can tell two expressions apart iff there is a *complete program* containing one of the expressions whose *observable behavior* changes when we replace that expression by the other. That is, two expressions are considered equal iff there is no such scenario that distinguishes them. But what do we mean by "complete program"? And what do we mean by "observable behavior"?

For the present purposes we will consider a complete program to be any expression of basic type (say, `int` or `bool` or `string`). The idea is that a complete program is one that computes a concrete result such as a number. The observable behavior of a complete program includes at least these aspects:

1. Its value, or lack thereof, either by non-termination or by raising an uncaught exception.

2. Its visible side effects, include visible modifications to mutable storage or any input/output it may perform.

In contrast here are some behaviors that we will not count as observations:

1. Execution time or space usage.
2. "Private" uses of storage (*e.g.*, internally-allocated reference cells).
3. The name of uncaught exceptions (*i.e.*, we will not distinguish between terminating with the uncaught exception `Bind` and the uncaught exception `Match`.

With these ideas in mind, it should be plausible that if we evaluate these bindings

```
val r = ref 0
val s = ref 0
```

then `r` and `s` are *not* equivalent. Consider the following usage of `r` to compute an integer result:

```
(s := 1 ; !r)
```

Clearly this expression evaluates to `0`, and mutates the binding of `s`. Now replace `r` by `s` to obtain

```
(s := 1 ; !s)
```

This expression evaluates to `1`, and mutates `s` as before. These two complete programs distinguish `r` from `s`, and therefore must be considered distinct.

Had we replaced the binding for `s` by the binding

```
val s = r
```

then the two expressions that formerly distinguished `r` from `s` no long do so — they are, after all, bound to the *same* reference cell! In fact, no program can be concocted that would distinguish them. In this case `r` and `s` are equivalent.

Now consider a third, very similar scenario. Let us declare `r` and `s` as follows:

```
val r = ref ()
val s = ref ()
```

Are `r` and `s` equivalent or not? We might first try to distinguish them by a variant of the experiment considered above. This breaks down because there is only one possible value we can assign to a variable of type `unit ref`! Indeed, one may suspect that `r` and `s` are equivalent in this case, but in fact there is a way to distinguish them! Here's a complete program involving `r` that we will use to distinguish `r` from `s`:

```
if r=r then "it's r" else "it's not"
```

Now replace the first occurrence of `r` by `s` to obtain

```
if s=r then "it's r" else "it's not"
```

and the result is different.

This example hinges on the fact that ML defines equality for values of reference type to be *reference equality* (or, occasionally, *pointer equality*). Two reference cells (of the same type) are equal in this sense iff they both arise from the exact same use of the `ref` operation to allocate that cell; otherwise they are distinct. Thus the two cells bound to `r` and `s` above are observably distinct (by testing reference equality), even though they can only ever hold the value `()`. Had equality not been included as a primitive, any two reference cells of `unit` type would have been equal.

Why does ML provide such a fine-grained notion of equality? "True" equality, as defined by Leibniz's Principle, is, unfortunately, undecidable — there is no computer program that determines whether two expressions are equivalent in this sense. ML provides a useful, conservative approximation to true equality that in some cases is not defined (you cannot test two functions for equality) and in other cases is too picky (it distinguishes reference cells that are otherwise indistinguishable). Such is life.

13.4 Aliasing

To see how reference cells complicate programming, let us consider the problem of *aliasing*. Any two variables of the same reference type might be bound to the *same* reference cell, or to two different reference cells. For example, after the declarations

```
val r = ref 0
val s = ref 0
```

the variables `r` and `s` are not aliases, but after the declaration

```
val r = ref 0
val s = r
```

the variables `r` and `s` are aliases for the same reference cell.

These examples show that we must be careful when programming with variables of reference type. This is particularly problematic in the case of functions, because we cannot assume that two different argument variables are bound to different reference cells. They might, in fact, be bound to the same reference cell, in which case we say that the two variables are *aliases* for one another. For example, in a function of the form

`fn (x:`*typ* `ref, y:`*typ* `ref) =>` *exp*

we may not assume that `x` and `y` are bound to different reference cells. We must always ask ourselves whether we've properly considered aliasing when writing such a function. This is harder to do than it sounds. Aliasing is a huge source of bugs in programs that work with reference cells.

13.5 Programming Well With References

Using references it is possible to mimic the style of programming used in imperative languages such as C. For example, we might define the factorial function in imitation of such languages as follows:

```
fun imperative_fact (n:int) =
    let
        val result = ref 1
        val i = ref 0
        fun loop () =
            if !i = n then
               ()
            else
               (i := !i + 1;
                result := !result * !i;
                loop ())
    in
        loop (); !result
    end
```

Notice that the function `loop` is essentially just a while loop; it repeatedly executes its body until the contents of the cell bound to `i` reaches `n`. The tail call to `loop` is essentially just a `goto` statement to the top of the loop.

It is (appallingly) bad style to program in this fashion. The purpose of the function `imperative_fact` is to compute a simple function on the natural numbers. There is nothing about its definition that suggests that state must be maintained, and so it is senseless to allocate and modify storage to compute it. The definition we gave earlier is shorter, simpler, more efficient, and hence more suitable to the task. This is not to suggest, however, that there are no good uses of references. We will now discuss some important uses of state in ML.

13.5.1 Private Storage

The first example is the use of higher-order functions to manage shared private state. This programming style is closely related to the use of objects to manage state in object-oriented programming languages. Here's an example to frame the discussion:

```
local
   val counter = ref 0
in
   fun tick () = (counter := !counter + 1; !counter)
   fun reset () = (counter := 0)
end
```

This declaration introduces two functions, `tick` of type `unit -> int` and `reset` of type `unit -> unit`. Their definitions share a *private* variable `counter` that is bound to a mutable cell containing the current value of a shared counter. The `tick` operation increments the counter and returns its new value, and the `reset` operation resets its value to zero. The types of the operations suggest that implicit state is involved. In the absence of exceptions and implicit state, there is only one useful function of type `unit->unit`, namely the function that always returns its argument (and it's debatable whether this is really useful!).

The declaration above defines two functions, `tick` and `reset`, that share a single private counter. Suppose now that we wish to have several different *instances* of a counter — different pairs of functions `tick` and `reset` that share different state. We can achieve this by defining a *counter generator* (or *constructor*) as follows:

```
fun new_counter () =
    let
        val counter = ref 0
        fun tick () = (counter := !counter + 1; !counter)
        fun reset () = (counter := 0)
    in
        { tick = tick, reset = reset }
    end
```

The type of `new_counter` is

```
unit -> { tick : unit->int, reset : unit->unit }.
```

We've packaged the two operations into a record containing two functions that share private state. There is an obvious analogy with class-based object-oriented programming. The function `new_counter` may be thought of as a *constructor* for a class of counter *objects*. Each object has a private *instance variable* `counter` that is shared between the *methods* `tick` and `reset` of the object represented as a record with two fields.

Here's how we use counters.

```
val c1 = new_counter ()
val c2 = new_counter ()
#tick c1 ();
(* 1 *)
#tick c1 ();
(* 2 *)
#tick c2 ();
(* 1 *)
#reset c1 ();
#tick c1 ();
(* 1 *)
#tick c2 ();
(* 2 *)
```

Notice that `c1` and `c2` are *distinct* counters that increment and reset independently of one another.

13.5.2 Mutable Data Structures

A second important use of references is to build *mutable data structures*. The data structures (such as lists and trees) we've considered so far are *immutable* in the sense that it is impossible to *change* the structure of the list or tree without building a modified copy of that structure. This is both a benefit and a drawback. The principal benefit is that immutable data structures are *persistent* in that operations performed on them do not destroy the original structure — in ML we can eat our cake and have it too. For example, we can simultaneously maintain a dictionary both before and after insertion of a given word. The principal drawback is that if we aren't really relying on persistence, then it is wasteful to make a copy of a structure if the original is going to be discarded anyway. What we'd like in this case is to have an "update in place" operation to build an *ephemeral* (opposite of persistent) data structure. To do this in ML we make use of references.

A simple example is the type of *possibly circular lists*, or *pcl's*. Informally, a pcl is a finite graph in which every node has at most one neighbor, called its *predecessor*, in the graph. In contrast to ordinary lists the predecessor

relation is not necessarily well-founded: there may be an infinite sequence of nodes arranged in descending order of predecession. Since the graph is finite, this can only happen if there is a cycle in the graph: some node has an ancestor as predecessor. How can such a structure ever come into existence? If the predecessors of a cell are needed to construct a cell, then the ancestor that is to serve as predecessor in the cyclic case can never be created! The "trick" is to employ *backpatching*: the predecessor is initialized to `Nil`, so that the node and its ancestors can be constructed, then it is reset to the appropriate ancestor to create the cycle.

This can be achieved in ML using the following `datatype` declaration:

```
datatype 'a pcl = Pcl of 'a pcell ref
and 'a pcell = Nil | Cons of 'a * 'a pcl;
```

A value of type *typ* `pcl` is essentially a reference to a value of type *typ* `pcell`. A value of type *typ* `pcell` is either `Nil`, the cell at the end of a non-circular possibly-circular list, or `Cons (`*h*`,` *t*`)`, where *h* is a value of type *typ* and `t` is another such possibly-circular list.

Here are some convenient functions for creating and taking apart possibly-circular lists:

```
fun cons (h, t) = Pcl (ref (Cons (h, t)));
fun nill () = Pcl (ref Nil);
fun phd (Pcl (ref (Cons (h, _)))) = h;
fun ptl (Pcl (ref (Cons (_, t)))) = t;
```

To implement backpatching, we need a way to "zap" the tail of a possibly-circular list.

```
fun stl (Pcl (r as ref (Cons (h, _))), u) =
    (r := Cons (h, u));
```

If you'd like, it would make sense to require that the tail of the `Cons` cell be the empty pcl, so that you're only allowed to backpatch at the end of a finite pcl.

Here is a finite and an infinite pcl.

```
val finite = cons (4, cons (3, cons (2, cons (1, nill ()))))
val tail = cons (1, nill());
val infinite = cons (4, cons (3, cons (2, tail)));
val _ = stl (tail, infinite)
```

The last step backpatches the tail of the last cell of `infinite` to be `infinite` itself, creating a circular list.

Now let us define the size of a pcl to be the number of distinct nodes occurring in it. It is an interesting problem is to define a `size` function for pcls that makes *no* use of auxiliary storage (*e.g.*, no set of previously-encountered nodes) and runs in time proportional to the number of cells in the pcl. The idea is to think of running a long race between a tortoise and a hare. If the course is circular, then the hare, which quickly runs out ahead of the tortoise, will eventually come from behind and pass it! Conversely, if this happens, the course must be circular.

```
local
  fun race (Nil, Nil) = 0
    | race (Cons (_, Pcl (ref c)), Nil) =
      1 + race (c, Nil)
    | race (Cons (_, Pcl (ref c)), Cons (_, Pcl (ref Nil))) =
      1 + race (c, Nil)
    | race (Cons (_, l), Cons (_, Pcl (ref (Cons (_, m))))) =
      1 + race' (l, m)
  and race' (Pcl (r as ref c), Pcl (s as ref d)) =
      if r=s then 0 else race (c, d)
in
  fun size (Pcl (ref c)) = race (c, c)
end
```

The hare runs twice as fast as the tortoise. We let the tortoise do the counting; the hare's job is simply to detect cycles. If the hare reaches the finish line, it simply waits for the tortoise to finish counting. This covers the first three clauses of `race`. If the hare has not yet finished, we must continue with the hare running at twice the pace, checking whether the hare catches the tortoise from behind. Notice that it can never arise that the tortoise reaches the end before the hare does! Consequently, the definition of `race` is inexhaustive.

13.6 Mutable Arrays

In addition to reference cells, ML also provides mutable arrays as a primitive data structure. The type *typ* `array` is the type of arrays carrying val-

ues of type *typ*. The basic operations on arrays are these:

```
val array : int * 'a -> 'a array
val length : 'a array -> int
val sub : 'a array * int -> 'a
val update : 'a array * int * 'a -> unit
```

The function `array` creates a new array of a given length, with the given value as the initial value of every element of the array. The function `length` returns the length of an array. The function `sub` performs a subscript operation, returning the ith element of an array `A`, where $0 \leq i < \texttt{length(A)}$. (These are just the basic operations on arrays; please see V for complete information.)

One simple use of arrays is for *memoization*. Here's a function to compute the nth Catalan number, which may be thought of as the number of distinct ways to parenthesize an arithmetic expression consisting of a sequence of n consecutive multiplication's. It makes use of an auxiliary summation function that you can easily define for yourself. (Applying `sum` to f and n computes the sum of $f0 + \cdots + fn$.)

```
fun C 1 = 1
  | C n = sum (fn k => (C k) * (C (n-k))) (n-1)
```

This definition of `C` is hugely inefficient because a given computation may be repeated exponentially many times. For example, to compute `C 10` we must compute `C 1`, `C 2`, ..., `C 9`, and the computation of `C` i engenders the computation of `C 1`, ..., `C` $i-1$ for each $1 \leq i \leq 9$. We can do better by caching previously-computed results in an array, leading to an enormous improvement in execution speed. Here's the code:

```
local
       val limit : int = 100
       val memopad : int option array =
           Array.array (limit, NONE)
in
       fun C' 1 = 1
         | C' n = sum (fn k => (C k)*(C (n-k))) (n-1)
       and C n =
           if n < limit then
              case Array.sub (memopad, n)
```

```
                of SOME r => r
                 | NONE =>
                   let
                       val r = C' n
                   in
                       Array.update (memopad, n, SOME r);
                       r
                   end
           else
              C' n
end
```

Note carefully the structure of the solution. The function `C` is a memoized version of the Catalan number function. When called it consults the memopad to determine whether or not the required result has already been computed. If so, the answer is simply retrieved from the memopad, otherwise the result is computed, stored in the cache, and returned. The function `C'` looks superficially similar to the earlier definition of `C`, with the important difference that the recursive calls are to `C`, rather than `C'` itself. This ensures that sub-computations are properly cached and that the cache is consulted whenever possible.

The main weakness of this solution is that we must fix an upper bound on the size of the cache. This can be alleviated by implementing a more sophisticated cache management scheme that dynamically adjusts the size of the cache based on the calls made to it.

13.7 Sample Code

Here is the code for this chapter.

Chapter 14

Input/Output

The Standard ML Basis Library (described in V) defines a three-layer input and output facility for Standard ML. These modules provide a rudimentary, platform-independent text I/O facility that we summarize briefly here. The reader is referred to V for more details. Unfortunately, there is at present no standard library for graphical user interfaces; each implementation provides its own package. See your compiler's documentation for details.

14.1 Textual Input/Output

The text I/O primitives are based on the notions of an *input stream* and an *output stream*, which are values of type `instream` and `outstream`, respectively. An input stream is an unbounded sequence of characters arising from some source. The source could be a disk file, an interactive user, or another program (to name a few choices). Any source of characters can be attached to an input stream. An input stream may be thought of as a buffer containing zero or more characters that have already been read from the source, together with a means of requesting more input from the source should the program require it. Similarly, an output stream is an unbounded sequence of characters leading to some sink. The sink could be a disk file, an interactive user, or another program (to name a few choices). Any sink for characters can be attached to an output stream. An output stream may be thought of as a buffer containing zero or more characters that have been produced by the program but have yet to be flushed to the

sink.

Each program comes with one input stream and one output stream, called `stdIn` and `stdOut`, respectively. These are ordinarily connected to the user's keyboard and screen, and are used for performing simple text I/O in a program. The output stream `stdErr` is also pre-defined, and is used for error reporting. It is ordinarily connected to the user's screen.

Textual input and output are performed on streams using a variety of primitives. The simplest are `inputLine` and `print`. To read a line of input from a stream, use the function `inputLine` of type `instream -> string`. It reads a line of input from the given stream and yields that line as a string whose last character is the line terminator. If the source is exhausted, return the empty string. To write a line to `stdOut`, use the function `print` of type `string -> unit`. To write to a specific stream, use the function `output` of type `outstream * string -> unit`, which writes the given string to the specified output stream. For interactive applications it is often important to ensure that the output stream is flushed to the sink (*e.g.*, so that it is displayed on the screen). This is achieved by calling `flushOut` of type `outstream -> unit`, which ensures that the output stream is flushed to the sink. The print function is a composition of `output` (to `stdOut`) and `flushOut`.

A new input stream may be created by calling the function `openIn` of type `string -> instream`. When applied to a string, the system attempts to open a file with that name (according to operating system-specific naming conventions) and attaches it as a source to a new input stream. Similarly, a new output stream may be created by calling the function `openOut` of type `string -> outstream`. When applied to a string, the system attempts to create a file with that name (according to operating system-specific naming conventions) and attaches it as a sink for a new output stream. An input stream may be closed using the function `closeIn` of type `instream -> unit`. A closed input stream behaves as if there is no further input available; request for input from a closed input stream yield the empty string. An output stream may be closed using `closeOut` of type `outstream -> unit`. A closed output stream is unavailable for further output; an attempt to write to a closed output stream raises the exception `TextIO.IO`.

The function `input` of type `instream -> string` is a blocking read operation that returns a string consisting of the characters currently available from the source. If none are currently available, but the end of source has

not been reached, then the operation blocks until at least one character is available from the source. If the source is exhausted or the input stream is closed, `input` returns the null string. To test whether an `input` operation would block, use the function `canInput` of type `instream * int -> int option`. Given a stream s and a bound n, the function `canInput` determines whether or not a call to `input` on s would immediately yield up to n characters. If the `input` operation would block, `canInput` yields `NONE`; otherwise it yields `SOME` k, with $0 \leq k \leq n$ being the number of characters immediately available on the input stream. If `canInput` yields `SOME 0`, the stream is either closed or exhausted. The function `endOfStream` of type `instream -> bool` tests whether the input stream is currently at the end (no further input is available from the source). This condition is transitive since, for example, another process might append data to an open file in between calls to `endOfStream`.

The function `output` of type `outstream * string -> unit` writes a string to an output stream. It may block until the sink is able to accept the entire string. The function `flushOut` of type `outstream -> unit` forces any pending output to the sink, blocking until the sink accepts the remaining buffered output.

This collection of primitive I/O operations is sufficient for performing rudimentary textual I/O. For further information on textual I/O, and support for binary I/O and Posix I/O primitives, see the Standard ML Basis Library.

14.2 Sample Code

Here is the code for this chapter.

Chapter 15

Lazy Data Structures

In ML all variables are bound *by value*, which means that the bindings of variables are *fully evaluated* expressions, or *values*. This general principle has several consequences:

1. The right-hand side of a `val` binding is evaluated before the binding is effected. If the right-hand side has no value, the `val` binding does not take effect.

2. In a function application the argument is evaluated before being passed to the function by binding that value to the parameter of the function. If the argument does not have a value, then neither does the application.

3. The arguments to value constructors are evaluated before the constructed value is created.

According to the by-value discipline, the bindings of variables are evaluated, regardless of whether that variable is ever needed to complete execution. For example, to compute the result of applying the function `fn x => 1` to an argument, we never actually need to evaluate the argument, but we do anyway. For this reason ML is sometimes said to be an *eager* language.

An alternative is to bind variables *by name*,[1] which means that the binding of a variable is an *unevaluated* expression, known as a *computation* or a

[1]The terminology is historical, and not well-motivated. It is, however, firmly established.

suspension or a *thunk*.[2] This principle has several consequences:

1. The right-hand side of a `val` binding is not evaluated before the binding is effected. The variable is bound to a computation (unevaluated expression), not a value.

2. In a function application the argument is passed to the function in unevaluated form by binding it directly to the parameter of the function. This holds regardless of whether the argument has a value or not.

3. The arguments to value constructor are left unevaluated when the constructed value is created.

According to the by-name discipline, the bindings of variables are only evaluated (if ever) when their values are required by a primitive operation. For example, to evaluate the expression `x+x`, it is necessary to evaluate the binding of `x` in order to perform the addition. Languages that adopt the by-name discipline are, for this reason, said to be *lazy*.

This discussion glosses over another important aspect of lazy evaluation, called *memoization*. In actual fact laziness is based on a refinement of the *by-name* principle, called the *by-need* principle. According to the by-name principle, variables are bound to unevaluated computations, and are evaluated only as often as the value of that variable's binding is required to complete the computation. In particular, to evaluate the expression `x+x` the value of the binding of `x` is needed *twice*, and hence it is evaluated twice. According to the by-need principle, the binding of a variable is evaluated *at most once* — not at all, if it is never needed, and exactly once if it ever needed at all. Re-evaluation of the same computation is avoided by *memoization*. Once a computation is evaluated, its value is saved for future reference should that computation ever be needed again.

The advantages and disadvantages of lazy *vs.* eager languages have been hotly debated. We will not enter into this debate here, but rather content ourselves with the observation that *laziness is a special case of eagerness*. (Recent versions of) ML have *lazy data types* that allow us to treat unevaluated computations as values of such types, allowing us to incorporate laziness into the language without disrupting its fundamental character

[2] For reasons that are lost in the mists of time.

on which so much else depends. This affords the benefits of laziness, but on a controlled basis — we can use it when it is appropriate, and ignore it when it is not.

The main benefit of laziness is that it supports *demand-driven* computation. This is useful for representing *on-line* data structures that are created only insofar as we examine them. *Infinite* data structures, such as the sequence of *all* prime numbers in order of magnitude, are one example of an on-line data structure. Clearly we cannot ever "finish" creating the sequence of all prime numbers, but we can create as much of this sequence as we need for a given run of a program. *Interactive* data structures, such as the sequence of inputs provided by the user of an interactive system, are another example of on-line data structures. In such a system the user's inputs are not pre-determined at the start of execution, but rather are created "on demand" in response to the progress of computation up to that point. The demand-driven nature of on-line data structures is precisely what is needed to model this behavior.

Note: Lazy evaluation is a non-standard feature of ML that is supported only by the SML/NJ compiler. The lazy evaluation features must be enabled by executing the following at top level:

```
Compiler.Control.lazysml := true;
open Lazy;
```

15.1 Lazy Data Types

SML/NJ provides a general mechanism for introducing lazy data types by simply attaching the keyword `lazy` to an ordinary `datatype` declaration. The ideas are best illustrated by example. We will focus attention on the type of *infinite streams*, which may be declared as follows:

```
datatype lazy 'a stream = Cons of 'a * 'a stream
```

Notice that this type definition has no "base case"! Had we omitted the keyword `lazy`, such a `datatype` would not be very useful, since there would be no way to create a value of that type!

Adding the keyword `lazy` makes all the difference. Doing so specifies that the values of type *typ* `stream` are *computations* of values of the form

`Cons (`*val*`,` *val*′`)`,

where *val* is of type *typ*, and *val′* is another such computation. Notice how this description captures the "incremental" nature of lazy data structures. The computation is not evaluated until we examine it. When we do, its structure is revealed as consisting of an element *val* together with another suspended computation of the same type. Should we inspect that computation, it will again have this form, and so on *ad infinitum*.

Values of type *typ* `stream` are created using a `val rec lazy` declaration that provides a means for building a "circular" data structure. Here is a declaration of the infinite stream of `1`'s as a value of type `int stream`:

```
val rec lazy ones = Cons (1, ones)
```

The keyword `lazy` indicates that we are binding `ones` to a computation, rather than a value. The keyword `rec` indicates that the computation is *recursive* (or *self-referential* or *circular*). It is the computation whose underlying value is constructed using `Cons` (the only possibility) from the integer `1` and *the very same computation itself*.

We can inspect the underlying value of a computation by pattern matching. For example, the binding

```
val Cons (h, t) = ones
```

extracts the "head" and "tail" of the stream `ones`. This is performed by evaluating the computation bound to `ones`, yielding `Cons (1, ones)`, then performing ordinary pattern matching to bind `h` to `1` and `t` to `ones`.

Had the pattern been "deeper", further evaluation would be required, as in the following binding:

```
val Cons (h, (Cons (h', t')) = ones
```

To evaluate this binding, we evaluate `ones` to `Cons (1, ones)`, binding `h` to `1` in the process, then evaluate `ones` again to `Cons (1, ones)`, binding `h'` to `1` and `t'` to `ones`. The general rule is *pattern matching forces evaluation of a computation to the extent required by the pattern*. This is the means by which lazy data structures are evaluated only insofar as required.

15.2 Lazy Function Definitions

The combination of (recursive) `lazy` function definitions and decomposition by pattern matching are the core mechanisms required to support lazy

evaluation. However, there is a subtlety about function definitions that requires careful consideration, and a third new mechanism, the *lazy function* declaration.

Using pattern matching we may easily define functions over lazy data structures in a familiar manner. For example, we may define two functions to extract the head and tail of a stream as follows:

```
fun shd (Cons (h, _)) = h
fun stl (Cons (_, s)) = s
```

These are functions that, when applied to a stream, evaluate it, and match it against the given patterns to extract the head and tail, respectively.

While these functions are surely very natural, there is a subtle issue that deserves careful discussion. The issue is whether these functions are "lazy enough". From one point of view, what we are doing is decomposing a computation by evaluating it and retrieving its components. In the case of the `shd` function there is no other interpretation — we are extracting a value of type *typ* from a value of type *typ* `stream`, which is a computation of a value of the form `Cons (`exp_h`,` exp_t`)`. We can adopt a similar viewpoint about `stl`, namely that it is simply extracting a component value from a computation of a value of the form `Cons (`exp_h`,` exp_t`)`.

However, in the case of `stl`, another point of view is also possible. Rather than think of `stl` as *extracting* a value from a stream, we may instead think of it as *creating* a stream out of another stream. Since streams are computations, the stream created by `stl` (according to this view) should also be suspended until its value is required. Under this interpretation the argument to `stl` should not be evaluated until its result is required, rather than at the time `stl` is applied. This leads to a variant notion of "tail" that may be defined as follows:

```
fun lazy lstl (Cons (_, s)) = s
```

The keyword `lazy` indicates that an application of `lstl` to a stream does *not* immediately perform pattern matching on its argument, but rather *sets up* a stream computation that, when forced, forces the argument and extracts the tail of the stream.

The behavior of the two forms of tail function can be distinguished using `print` statements as follows:

```
val rec lazy s = (print "."; Cons (1, s))
val _ = stl s    (* prints "." *)
val _ = stl s    (* silent *)
val rec lazy s = (print "."; Cons (1, s));
val _ = lstl s   (* silent *)
val _ = stl s    (* prints "." *)
```

Since `stl` evaluates its argument when applied, the "`.`" is printed when it is first called, but not if it is called again. However, since `lstl` only sets up a computation, its argument is not evaluated when it is called, but only when its result is evaluated.

15.3 Programming with Streams

Let's define a function `smap` that applies a function to every element of a stream, yielding another stream. The type of `smap` should be `('a -> 'b) -> 'a stream -> 'b stream`. The thing to keep in mind is that the application of `smap` to a function and a stream should set up (but not compute) another stream that, when forced, forces the argument stream to obtain the head element, applies the given function to it, and yields this as the head of the result.

Here's the code:

```
fun smap f =
    let
        fun lazy loop (Cons (x, s)) =
            Cons (f x, loop s)
    in
        loop
    end
```

We have "staged" the computation so that the partial application of `smap` to a function yields a function that loops over a given stream, applying the given function to each element. This loop is a lazy function to ensure that it merely sets up a stream computation, rather than evaluating its argument when it is called. Had we dropped the keyword `lazy` from the definition of the loop, then an application of `smap` to a function and a stream would immediately force the computation of the head element of

the stream, rather than merely set up a future computation of the same result.

To illustrate the use of smap, here's a definition of the infinite stream of natural numbers:

```
val one_plus = smap (fn n => n+1)
val rec lazy nats = Cons (0, one_plus nats)
```

Now let's define a function `sfilter` of type

```
('a -> bool) -> 'a stream -> 'a stream
```

that filters out all elements of a stream that do not satisfy a given predicate.

```
fun sfilter pred =
    let
        fun lazy loop (Cons (x, s)) =
            if pred x then
               Cons (x, loop s)
            else
               loop s
    in
        loop
    end
```

We can use `sfilter` to define a function `sieve` that, when applied to a stream of numbers, retains only those numbers that are not divisible by a preceding number in the stream:

```
fun m mod n = m - n * (m div n)
fun divides m n = n mod m = 0
fun lazy sieve (Cons (x, s)) =
    Cons (x, sieve (sfilter (not o (divides x)) s))
```

(This example uses `o` for function composition.)

We may now define the infinite stream of primes by applying `sieve` to the natural numbers greater than or equal to 2:

```
val nats2 = stl (stl nats)
val primes = sieve nats2
```

To inspect the values of a stream it is often useful to use the following function that takes $n \geq 0$ elements from a stream and builds a list of those n values:

```
fun take 0 _ = nil
  | take n (Cons (x, s)) = x :: take (n-1) s
```

Here's an example to illustrate the effects of memoization:

```
val rec lazy s = Cons ((print "."; 1), s)
val Cons (h, _) = s;
(* prints ".", binds h to 1 *)
val Cons (h, _) = s;
(* silent, binds h to 1 *)
```

Replace `print ".";1` by a time-consuming operation yielding `1` as result, and you will see that the second time we force `s` the result is returned instantly, taking advantage of the effort expended on the time-consuming operation induced by the first force of `s`.

15.4 Sample Code

Here is the code for this chapter.

Chapter 16

Equality and Equality Types

16.1 Sample Code

Here is the code for this chapter.

Chapter 17

Concurrency

Concurrent ML (CML) is an extension of Standard ML with mechanisms for concurrent programming. It is available as part of the Standard ML of New Jersey compiler. The eXene Library for programming the X windows system is based on CML.

17.1 Sample Code

Here is the code for this chapter.

Part III

The Module Language

The Standard ML *module language* comprises the mechanisms for structuring programs into separate units. Program units are called *structures*. A structure consists of a collection of components, including types and values, that constitute the unit. Composition of units to form a larger unit is mediated by a *signature*, which describes the components of that unit. A signature may be thought of as the type of a unit. Large units may be structured into hierarchies using *substructures*. Generic, or parameterized, units may be defined as *functors*.

Chapter 18

Signatures and Structures

The fundamental constructs of the ML module system are *signatures* and *structures*. A signature may be thought of as an interface or specification of a structure, and a structure may correspondingly be thought of as an implementation of a signature. Many languages (such as Modula, Ada, or Java) have similar constructs: signatures are analogous to interfaces or package specifications or class types, and structures are analogous to implementations or packages or classes. However, these are only rough analogies that should not be taken too seriously.

18.1 Signatures

A *signature* is a *specification*, or a *description*, of a program unit, or *structure*. Structures consist of declarations of type constructors, exception constructors, and value bindings. A signature is an item-by-item specification of these components of a structure. A structure *matches*, or *implements*, a signature iff the requirements of the signature are met by the structure. (This will be made precise below.)

18.1.1 Basic Signatures

A basic signature expression has the form `sig` *specs* `end`, where *specs* is a sequence of specifications. There are four basic forms of specification that

may occur in *specs*:[1]

1. A *type specification* of the form

   `type (`$tyvar_1$`,...,`$tyvar_n$`)` *tycon* `[ =` *typ* `],`

   where the definition *typ* of *tycon* may or may not be present.

2. A *datatype specification*, which has precisely the same form as a datatype declaration.

3. An *exception specification* of the form

   `exception` *excon* `of` *typ*`.`

4. A *value specification* of the form

   `val` *id* `:` *typ*`.`

The sequence of specifications are to be understood in the order given, and no component may be specified more than once. Each specification may refer to the type constructors introduced earlier in the sequence.

Signatures may be given names using a *signature binding*

`signature` *sigid* `=` *sigexp*`,`

where *sigid* is a signature identifier and *sigexp* is a signature expression. Signature identifiers are abbreviations for the signatures to which they are bound. In practice we nearly always bind signature expressions to identifiers and refer to them by name.

Here is an illustrative example of a signature definition. We will refer back to this definition often in the rest of this chapter.

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
```

[1] There are two other forms of specification beyond these four, *substructure specifications* and *sharing specifications*. These will be introduced in chapter 21.

The signature QUEUE specifies a structure that must provide

1. a unary type constructor 'a queue,
2. a nullary exception Empty,
3. a polymorphic value empty of type 'a queue,
4. two polymorphic functions, insert and remove, with the specified type schemes.

Notice that queues are polymorphic in the type of elements of the queue — the same operations are used regardless of the element type.

18.1.2 Signature Inheritance

Signatures may be built up from one another using two principal tools, *signature inclusion* and *signature specialization*. Each is a form of *inheritance* in which a new signature is created by enriching another signature with additional information.

Signature inclusion is used to add more components to an existing signature. For example, if we wish to add an emptiness test to the signature QUEUE we might define the augmented signature, QUEUE_WITH_EMPTY, using the following signature binding:

```
signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
  end
```

As the notation suggests, the signature QUEUE is included into the body of the signature QUEUE_WITH_EMPTY, and an additional component is added.

It is not strictly necessary to use include to define this signature. Indeed, we may define it directly using the following signature binding:

```
signature QUEUE_WITH_EMPTY =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
    val is_empty : 'a queue -> bool
  end
```

There is no semantic difference between the two definitions of `QUEUE_WITH_EMPTY`. Signature inclusion is a convenience that documents the "history" of how the more refined signature was created.

Signature specialization is used to augment an existing signature with additional type definitions. For example, if we wish to refine the signature `QUEUE` to specify that the type constructor `'a queue` must be defined as a pair of lists, we may proceed as follows:

```
signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

The `where type` clause "patches" the signature `QUEUE` by adding a definition for the type constructor `'a queue`.

The signature `QUEUE_AS_LISTS` may also be defined directly as follows:

```
signature QUEUE_AS_LISTS =
  sig
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
```

A `where type` clause may not be used to re-define a type that is already defined in a signature. For example, the following is illegal:

```
signature QUEUE_AS_LISTS_AS_LIST =
  QUEUE_AS_LISTS where type 'a queue = 'a list
```

If you wish to replace the definition of a type constructor in a signature with another definition using `where type`, you must go back to a common ancestor in which that type is not yet defined.

```
signature QUEUE_AS_LIST =
  QUEUE where type 'a queue = 'a list
```

Two signatures are said to be *equivalent* iff they differ only up to the type equivalences induced by type abbreviations.[2] For example, the signature `QUEUE where type 'a queue = 'a list` is equivalent to the signature

```
signature QUEUE_AS_LIST =
  sig
    type 'a queue = 'a list
    exception Empty
    val empty : 'a list
    val insert : 'a * 'a list -> 'a list
    val remove : 'a list -> 'a * 'a list
  end
```

Within the scope of the definition of the type `'a queue` as `'a list`, the two are equivalent, and hence the specifications of the value components are equivalent.

This principle of equivalence is sometimes called *progagation of type sharing*. Within the scope of the type declaration in the signature `QUEUE_AS_LIST`, the type constructors `'a queue` and `'a list` are said to *share*, or are *equivalent*. Therefore they may be used interchangeably within their scope, as we have done above, without affecting the meaning.

18.2 Structures

A *structure* is a unit of program consisting of a sequence of declarations of types, exceptions, and values. Structures are implementations of signatures; signatures are the "types" of structures.

[2] In some languages signatures are compared *by name*, which means that two signatures are equivalent iff they are the same signature identifier. This is not the case in ML.

18.2.1 Basic Structures

The basic form of structure is an encapsulated sequence of declarations of the form `struct` *decs* `end`. The declarations in *decs* are of one of the following four forms:

1. A *type declaration* defining a type constructor.
2. A *datatype declaration* defining a new datatype.
3. An *exception declaration* defining a new exception constructor with a specified argument type.
4. A *value declaration* defining a new value variable with a specified type.

These are precisely the declarations introduced in Part II.

A structure expression is well-formed iff it consists of a well-formed sequence of well-formed declarations (according to the rules given in Part II). A structure expression is evaluated by evaluating each of the declarations within it, in the order given. This amounts to evaluating the right-hand sides of each value declaration in turn to determine its value, which is then bound to the corresponding value identifier. This means, in particular, that any side effects that arise out of evaluating the constituent value bindings occur when the structure expression is evaluated. A structure value is a structure expression in which all bindings are fully evaluated.

A structure may be bound to a structure identifier using a *structure binding* of the form

`structure` *strid* `=` *strexp*

This declaration defines *strid* to stand for the value of *strexp*. Such a declaration is well-formed exactly when *strexp* is well-formed. It is evaluated by evaluating the right-hand side, and binding the resulting structure value to *strid*.

Here is an example of a structure binding:

```
structure Queue =
  struct
    type 'a queue = 'a list * 'a list
    exception Empty
    val empty = (nil, nil)
    fun insert (x, (b,f)) = (x::b, f)
    fun remove (nil, nil) = raise Empty
      | remove (bs, nil) = remove (nil, rev bs)
      | remove (bs, f::fs) = (f, (bs, fs))
  end
```

Recall that a `fun` binding is really an abbreviation for a `val rec` binding! Thus the bindings of the value identifiers `insert` and `remove` are function expressions.

18.2.2 Long and Short Identifiers

Once a structure has been bound to a structure identifier, we may access its components using *paths*, or *long identifiers*, or *qualified names*. A path has the form *strid*`.`*id*.[3] It stands for the `id` component of the structure bound to `strid`. For example, `Queue.empty` refers to the `empty` component of the structure `Queue`. It has type `'a Queue.queue` (note well the syntax!), stating that it is a polymorphic value whose type is built up from the unary type constructor `Queue.queue`. Similarly, the function `Queue.insert` has type `'a * 'a Queue.queue -> 'a Queue.queue` and `Queue.remove` has type `'a Queue.queue -> 'a * 'a Queue.queue`.

Type definitions permeate structure boundaries. For example, the type `'a Queue.queue` is equivalent to the type `'a list` because it is defined to be so in the structure `Q`. We will shortly introduce the means for limiting the visibility of type definitions. Unless special steps are taken, the definitions of types within a structure determine the definitions of the long identifiers that refer to those types within a structure. Consequently, it is correct to write an expression such as

```
val q = Queue.insert (1, ([6,5,4],[1,2,3]))
```

[3] In chapter 21, we will generalize this to admit an arbitrary sequence of *strid*'s separated by a dot.

even though the list `[6,5,4]` was not obtained by using the operations from the structure `Q`. This is because the type `int Queue.queue` is equivalent to the type `int list`, and hence the call to `insert` is well-typed.

The use of long identifiers can get out of hand, cluttering the program, rather than clarifying it. Suppose that we are frequently using the `Queue` operations in a program, so that the code is cluttered with calls to `Queue.empty`, `Queue.insert`, and `Queue.remove`. One way to reduce clutter is to introduce a *structure abbreviation* of the form `structure` *strid* `=` *strid*′ that introduces one structure identifier as an abbreviation for another. For example, after declaring

```
structure Q = Queue
```

we may write `Q.empty`, `Q.insert`, and `Q.remove`, rather than the more verbose forms mentioned above.

Another way to reduce clutter is to *open* the structure `Queue` to incorporate its bindings directly into the current environment. An *open* declaration has the form

`open` $strid_1$ `...` $strid_n$

which incorporates the bindings from the given structures in left-to-right order (later structures override earlier ones when there is overlap). For example, the declaration

```
open Queue
```

incorporates the body of the structure `Queue` into the current environment so that we may write just `empty`, `insert`, and `remove`, without qualification, to refer to the corresponding components of the structure `Queue`.

Although this is surely convenient, using `open` has its disadvantages. One is that we cannot simultaneously open two structures that have a component with the same name. For example, if we write

```
open Queue Stack
```

where the structure `Stack` also has a component empty, then uses of `empty` (without qualification) will stand for `Stack.empty`, not for `Queue.empty`.

Another problem with `open` is that it is hard to control its behavior, since it incorporates the *entire* body of a structure, and hence may inadvertently shadow identifiers that happen to be also used in the structure. For

example, if the structure `Queue` happened to define an auxiliary function `helper`, that function would also be incorporated into the current environment by the declaration `open Queue`, which may not have been intended. This turns out to be a source of many bugs; it is best to use `open` sparingly, and only then in a `let` or `local` declaration (to limit the damage).

18.3 Sample Code

Here is the code for this chapter.

Chapter 19

Signature Matching

When does a structure implement a signature? Roughly speaking, the structure must provide *all of the components* and satisfy *all of the type definitions* required by the signature. Exception components must be provided with types equivalent to those in the signature. Value components must be provided with types at least as general as those in the signature. Type components must be provided with the right arities, and with equivalent definitions (if any).

These are useful rules of thumb; nailing them down is surprisingly tricky. Let us mention a few issues that complicate matters:

- To *minimize bureaucracy*, a structure may provide *more* components than are strictly required by the signature. If a signature requires components `x`, `y`, and `z`, it is sufficient for the structure to provide `x`, `y`, `z`, and `w`.
- To *enhance reuse*, a structure may provide values with *more general* types than are required by the signature. If a signature demands a function of type `int->int`, it is enough to provide a function of type `'a->'a`.
- To *avoid over-specification*, a datatype may be provided where a type is required, and a value constructor may be provided where a value is required.
- To *increase flexibility*, a structure may consist of declarations presented in any sensible order, not just the order specified in the signature, provided that the requirements of the specification are met.

What it means for a structure to implement a signature is very similar to what it means for an expression to have a type. An expression *exp* has type *typ* iff *typ* is an instance of the *principal* type of *exp*. Similarly, we will define a structure to implement a signature iff the principal signature of the structure matches that signature.

19.1 Principal Signatures

The *principal signature* of a structure is, in a precise sense, the *most specific* description of the components of that structure. It captures everything that needs to be known about that structure during type checking. It is an important, and highly non-trivial, property of ML that there is always a principal signature for any well-formed structure. For the purposes of type checking, the principal signature is the official proxy for the structure. We need never examine the code of the structure durng type checking, once its principal signature has been determined.

A structure expression is assigned a principal signature by a component-by-component analysis of its constituent declarations. The principal signature of a structure is obtained as follows:[1]

1. Corresponding to a declaration of the form

   `type (`$tyvar_1$`,...,`$tyvar_n$`)` *tycon* `=` *typ*,

   the principal signature contains the specification

   `type (`$tyvar_1$`,...,`$tyvar_n$`)` *tycon* `=` *typ*

   The principal signature includes the definition of *tycon*.

2. Corresponding to a declaration of the form

   `datatype (`$tyvar_1$`,...,`$tyvar_n$`)` *tycon* `=`
   con_1 `of` typ_1 `|` ... `|` con_k `of` typ_k

   the principal signature contains the specification

[1] These rules gloss over some technical complications that arise only in unusual circumstances. See *The Definition of Standard ML* [3] for complete details.

```
datatype (tyvar1,...,tyvarn) tycon =
  con1 of typ1 | ... | conk of typk
```

The specification is identical to the declaration.

3. Corresponding to a declaration of the form

   ```
   exception id of typ
   ```

   the principal signature contains the specification

   ```
   exception id of typ
   ```

4. Corresponding to a declaration of the form

   ```
   val id = exp
   ```

   the principal signature contains the specification

   ```
   val id : typ
   ```

   where *typ* is the principal type scheme of the expression *exp* (relative to the preceding declarations). Keep in mind that `fun` bindings are really (recursive) `val` bindings of function type.

In brief, the principal signature contains all of the type definitions, datatype definitions, and exception bindings of the structure, plus the principal types of its value bindings.

An important point to keep in mind is that the principal signature of a structure is obtained by inspecting the code of that structure. While this may seem like an obvious point, it has important implications for managing dependencies between modules. We will return to this point in section 20.4 of chapter 20.

19.2 Matching

A candidate signature $sigexp_c$ is said to *match* a target signature $sigexp_t$ iff $sigexp_c$ has all of the components and all of the type equations specified by $sigexp_t$. More precisely,

1. Every type constructor in the target must also be present in the candidate, with the same arity (number of arguments) and an equivalent definition (if any).

2. Every datatype in the target must be present in the candidate, with equivalent types for the value constructors.

3. Every exception in the target must be present in the candidate, with an equivalent argument type.

4. Every value in the target must be present in the candidate, with at least as general a type.

The candidate may have additional components not mentioned in the target, or satisfy additional type equations not required in the target, but it cannot have fewer of either. The target signature may therefore be seen as a *weakening* of the candidate signature, since all of the properties of the latter are true of the former.

It is easy to see that the matching relation is reflexive — every signature matches itself — and transitive — if $sigexp_1$ matches $sigexp_2$ and $sigexp_2$ matches $sigexp_3$, then $sigexp_1$ matches $sigexp_3$. This means that the signature matching relation is a *pre-order*. It is also a partial order, which is to say that if $sigexp_1$ matches $sigexp_2$ and *vice versa*, then $sigexp_1$ and $sigexp_2$ are equivalent signatures (to within propagation of type equations).

It will be helpful to consider some examples. Recall the following signatures from chapter 18.

```
signature QUEUE =
  sig
    type 'a queue
    exception Empty
    val empty : 'a queue
    val insert : 'a * 'a queue -> 'a queue
    val remove : 'a queue -> 'a * 'a queue
  end
signature QUEUE_WITH_EMPTY =
  sig
    include QUEUE
    val is_empty : 'a queue -> bool
```

```
  end
signature QUEUE_AS_LISTS =
  QUEUE where type 'a queue = 'a list * 'a list
```

The signature QUEUE_WITH_EMPTY matches the signature QUEUE, because all of requirements of QUEUE are met by QUEUE_WITH_EMPTY. The converse does not hold, because QUEUE lacks the component is_empty, which is required by QUEUE_WITH_EMPTY.

The signature QUEUE_AS_LISTS matches the signature QUEUE. It is identical to QUEUE, apart from the additional specification of the type 'a queue. The converse fails, because the signature QUEUE does not satisfy the requirement that 'a queue be equivalent to 'a list * 'a list.

Matching does not distinguish between equivalent signatures. For example, consider the following signature:

```
signature QUEUE_AS_LIST = sig
  type 'a queue = 'a list
  exception Empty
  val empty : 'a list
  val insert : 'a * 'a list -> 'a list
  val remove : 'a list -> 'a * 'a list
  val is_empty : 'a list -> bool
end
```

At first glance you might think that this signature does not match the signature QUEUE, since the components of QUEUE_AS_LIST have superficially dissimilar types from those in QUEUE. However, the signature QUEUE_AS_LIST is *equivalent* to the signature QUEUE with type 'a queue = 'a list, which matches QUEUE for reasons noted earlier. Therefore, QUEUE_AS_LIST matches QUEUE as well.

Signature matching may also involve instantiation of polymorphic types. The types of values in the candidate may be more general than required by the target. For example, the signature

```
signature MERGEABLE_QUEUE =
  sig
    include QUEUE
    val merge : 'a queue * 'a queue -> 'a queue
  end
```

matches the signature

```
signature MERGEABLE_INT_QUEUE =
  sig
    include QUEUE
    val merge : int queue * int queue -> int queue
  end
```

because the polymorphic type of `merge` in `MERGEABLE_QUEUE` instantiates to its type in `MERGEABLE_INT_QUEUE`.

Finally, a datatype specification matches a signature that specifies a type with the same name and arity (but no definition), and zero or more value components corresponding to some (or all) of the value constructors of the datatype. The types of the value components must match exactly the types of the corresponding value constructors; no specialization is allowed in this case. For example, the signature

```
signature RBT_DT =
  sig
    datatype 'a rbt =
      Empty |
      Red of 'a rbt * 'a * 'a rbt |
      Black of 'a rbt * 'a * 'a rbt
  end
```

matches the signature

```
signature RBT =
  sig
    type 'a rbt
    val Empty : 'a rbt
    val Red : 'a rbt * 'a * 'a rbt -> 'a rbt
  end
```

The signature `RBT` specifies the type `'a rbt` as abstract, and includes two value specifications that are met by value constructors in the signature `RBT_DT`.

One way to understand this is to mentally rewrite the signature `RBT_DT` in the (fanciful) form[2]

[2]Unfortunately, the "signature" `RBT_DTS` is *not* a legal ML signature!

```
signature RBT_DTS =
  sig
    type 'a rbt
    con Empty : 'a rbt
    con Red : 'a rbt * 'a * 'a rbt -> 'a rbt
    con Black : 'a rbt * 'a * 'a rbt -> 'a rbt
```

The rule is simply that a `val` specification may be matched by a `con` specification.

19.3 Satisfaction

Returning to the motivating question of this chapter, a candidate structure *implements* a target signature iff the principal signature of the candidate structure matches the target signature. By the reflexivity of the matching relation it is immediate that a structure satisfies its principal signature. Therefore any signature implemented by a structure is *weaker* than the principal signature of that structure. That is, the principal signature is the *strongest* signature implemented by a structure.

19.4 Sample Code

Here is the code for this chapter.

Chapter 20

Signature Ascription

Signature ascription imposes the requirement that a structure implement a signature and, in so doing, weakens the signature of that structure for all subsequent uses of it. There are two forms of ascription in ML. Both require that a structure implement a signature; they differ in the extent to which the assigned signature of the structure is weakened by the ascription.

1. *Transparent*, or *descriptive* ascription. The structure is assigned the target signature *augmented* by propagating to the target the definitions of those types in the candidate.

2. *Opaque*, or *restrictive* ascription. The structure is assigned the target signature *as is*, without augmentation.

Both forms of ascription hide components not present in the target signature.

Using modules effectively requires careful control over the propagation of type information. What is held back is as at least as important as what is revealed. Opaque ascription is the means by which type information is curtailed; transparent ascription is the means by which it is propagated.

20.1 Ascribed Structure Bindings

The most common form of signature ascription is in a structure binding. There are two forms, the transparent

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

and the opaque

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

The only difference is that transparent ascription is written using a single colon, ":", whereas opaque ascription is written as ":>".

Ascribed structure bindings are type checked as follows. First, the compiler checks that *strexp* implements *sigexp* according to the rules given in chapter 19. Specifically, the principal signature $sigexp_0$ of *strexp* is determined and matched against *sigexp*. This determines an augmentation $sigexp'$ of *sigexp* by propagating type equations from the principal signature, $sigexp_0$. Second, the structure identifier is assigned a signature based on the form of ascription. For an opaque ascription, it is assigned the signature *sigexp*; for a transparent ascription, it is assigned the signature $sigexp'$.

Incidentally, this makes clear that transparent ascription is really a special case of opaque ascription! Since the augmented signature, $sigexp'$, is itself a signature, we could have written it ourselves and opaquely ascribed it to *strexp*, with the same net effect as a transparent ascription. Thus transparent ascription is really a form of "signature inference" in which we are asking the compiler to fill in details that it is too inconvenient to write ourselves. As we will see below, this is more than a minor convenience!

Ascribed signature bindings are evaluated by first evaluating *strexp*. Then a *view* of the resulting value is formed by dropping all components that are not present in the target signature, *sigexp*.[1] The structure variable *strid* is bound to the view. The formation of the view ensures that the components of a structure may always be accessed in constant time, and that there are no "space leaks" because of components that are present in the structure, but not in the signature.

[1]There are some technical complications to do with dropping type components. For example, if we attempt to include only the value constructors of a datatype, and not the datatype itself, the compiler will implicitly include the datatype to ensure that the types of the constructors are expressible in the signature of the view. Any such type implicitly included in the view is marked as "hidden". See *The Definition of Standard ML* for complete details.

20.2 Opaque Ascription

The primary use of opaque ascription is to enforce data abstraction. A good example is provided by the implementation of queues as, say, pairs of lists.

```
structure Queue :> QUEUE =
  struct
    type 'a queue = 'a list * 'a list
    val empty = (nil, nil)
    fun insert (x, (bs, fs)) = (x::bs, fs)
    exception Empty
    fun remove (nil, nil) = raise Empty
      | remove (bs, f::fs) = (f, (bs, fs))
      | remove (bs, nil) = remove (nil, rev bs)
  end
```

The use of opaque ascription ensures that the type `'a Queue.queue` is abstract. No definition is provided for it in the signature `QUEUE`, and therefore it has no definition in terms of other types of the language; the type `'a Queue.queue` is *abstract*.

The principal effect of rendering `'a Queue.queue` abstract is that *only* the operations `empty`, `insert`, and `remove` may be performed on values of that type. We may not make use of the fact that a queue is "really" a pair of lists simply because it is implemented this way. Instead we have obscured this fact by opaquely ascribing a signature that does not define the type `'a queue`. This ensures that all clients of the structure `Queue` are insulated from the details of how queues are implemented. This means that the implementation can be changed without fear of breaking any client code, a strong tool for enhancing maintainability of large programs. Were abstraction not enforced, the client might well (inadvertently or deliberately) make use of the representation of queues as pairs of lists, forcing all client code to be reconsidered whenever a change of representation occurs.

A closely-related reason for hiding the representation of a type is that it allows us to isolate the enforcement of representation invariants to the implementation of the abstraction. We may think of the type `'a Queue.queue` as the type of states of an abstract machine whose sole instructions are `empty` (the initial state), `insert`, and `remove`. Internally to the structure

`Queue` we may wish to impose invariants on the internal state of the machine. The beauty of data abstraction is that it supports an elegant technique for enforcing such invariants.

The technique, called the *assume-ensure*, or *rely-guarantee*, technique reduces enforcement of representation invariants to these two requirements:

1. All initialization instructions must *ensure* that the invariant holds true of the machine state after execution.

2. All state transition instructions may *assume* that the invariant holds of the inputs states, and must *ensure* that it holds of the output state.

By induction on the number of "instructions" executed, the invariant must hold for all states — *i.e.*, it must really be invariant!

Now suppose that we wish to implement an abstract type of priority queues for an arbitrary element type. The queue operations are no longer polymorphic in the element type because they actually "touch" the elements to determine their relative priorities. Here is a possible signature for priority queues that expresses this dependency:[2]

```
signature PQ =
  sig
    type elt
    val lt : elt * elt -> bool
    type queue
    exception Empty
    val empty : queue
    val insert : elt * queue -> queue
    val remove : queue -> elt * queue
  end
```

Now let us consider an implementation of priority queues in which the elements are taken to be strings. Since priority queues form an abstract type, we would expect to use opaque ascription to ensure that its representation is hidden. This suggests an implementation along these lines:

[2] In chapter 21 we'll introduce better means for structuring this module, but the central points discussed here will not be affected.

```
structure PrioQueue :> PQ =
  struct
    type elt = string
    val lt : string * string -> bool = (op <)
    type queue = ...
    ⋮
  end
```

But not only is the type `PrioQueue.queue` abstract, so is `PrioQueue.elt`! This leaves us no means of creating a value of type `PrioQueue.elt`, and hence we can never call `PrioQueue.insert`. The problem is that the interface is "too abstract" — it should only obscure the identity of the type `queue`, and not that of the type `elt`.

The solution is to augment the signature `PQ` with a definition for the type `elt`, then opaquely ascribe this to `PrioQueue`:

```
signature STRING_PQ = PQ where type elt = string
structure PrioQueue :> STRING_PQ = ...
```

Now the type `PrioQueue.elt` is equivalent to `string`, and we may call `PrioQueue.insert` with a string, as expected.

The moral is that there is always an element of judgement involved in deciding which types to hold abstract, and which to make opaque. In the case of priority queues, the determining factor is that we specified only the operations on `elt` that were required for the implementation of priority queues, and no others. This means that `elt` could not usefully be held abstract, but must instead be specified in the signature. On the other hand the operations on queues are intended to be complete, and so we hold the type abstract.

20.3 Transparent Ascription

Transparent ascription cuts down on the need for explicit specification of type definitions in signatures. As we remarked earlier, we can always replace uses of transparent ascription by a use of opaque ascription with a hand-crafted augmented signature. This can become burdensome. On the other hand, excessive use of transparent ascription impedes modular programming by exposing type information that would better be left abstract.

The prototypical use of transparent ascription is to form a *view* of a structure that eliminates the components that are not necessary in a given context without obscuring the identities of its type components. Consider the signature `ORDERED` defined as follows:

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
  end
```

This signature specifies a type `t` equipped with a comparison operation `lt`.

It should be clear that it would not make sense to opaquely ascribe this signature to a structure. Doing so would preclude ever calling the `lt` operation, for there would be no means of creating values of type `t`. Such a signature is only useful once it has been augmented with a definition for the type `t`. This is precisely what transparent ascription does for you.

For example, consider the following structure binding:

```
structure String : ORDERED =
  struct
    type t = string
    val clt = Char.<
    fun lt (s, t) = ... clt ...
  end
```

This structure implements string comparison in terms of character comparison (say, to implement the lexicographic ordering of strings). Ascription of the signature `ORDERED` ensures two things:

1. The auxiliary function `clt` is pruned out of the structure. It was intended for internal use, and was not meant to be externally visible.

2. The type `String.t` is equivalent to `string`, even though this fact is not present in the signature `ORDERED`. Transparent ascription computes an augmentation of `ORDERED` with this definition exposed. The "true" signature of `String` is the signature

   ```
   ORDERED where type t = string
   ```

which makes clear the underlying definition of `t`.

A related use of transparent ascription is to document an interpretation of a type without rendering it abstract. For example, we may wish to consider the integers ordered in two different ways, one by the standard arithmetic comparison, the other by divisibility. We might make the following declarations to express this:

```
structure IntLt : ORDERED =
  struct
    type t = int
    val lt = (op <)
  end
structure IntDiv : ORDERED =
  struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
  end
```

The ascription specifies the interpretation of `int` as partially ordered, in two senses, but does not hide the type of elements. In particular, `IntLt.t` and `IntDiv.t` are both equivalent to `int`.

20.4 Transparency, Opacity, and Dependency

An important observation is that transparent ascription is, in a sense, a form of opaque ascription. The target signature is augmented with the type definitions of the candidate, and then the augmented signature is opaquely ascribed to the structure. In principle one can always write out the augmented signature by hand (using `where type` clauses), then opaquely ascribe it to the structure. While this is true in principle, in practice it is simply too inconvenient to require the programmer to be fully explicit about the propagation of type information in signatures.[3]

Transparent ascription is a convenience akin to type inference. The compiler automatically fleshes out omitted information in a manner that

[3]Worse, for very technical reasons, it is not always possible to write the augmented signature by hand, because to do so requires access to "hidden" types.

is convenient for the programmer. However, this convenience comes at a price: *whenever you use transparent ascription, the compiler must have access to the source code of the structure to determine the "true" signature of that structure.* This means that any code that makes reference to the ascribed structure depends on the *implementation* of that structure, and not just its (apparent) *interface*. With transparent ascription, what you see (the ascribed signature) is *not* what you get! Rather, what you get is an augmentation of what you see obtained by inspecting the implementation of that structure.

On the other hand, whereas transparent ascription therefore introduces implementation dependencies, opaque ascription eliminates them. Once a signature has been opaquely ascribed to a structure, all future uses of that structure may rely only on the ascribed signature. Since the signature is given independently of the implementation, client code is insulated from changes to the implementation that do not also affect its interface. This greatly facilitates team development and code evolution.

Implementation dependencies are fundamentally *anti-modular*.[4] The goal of modular programming is to isolate parts of programs from one another so that they can be independently developed and modified with minimal interference. The fewer the dependencies between modules, the fewer the problems with integrating modules to form a complete system.

20.5 Sample Code

Here is the code for this chapter.

[4] That's why inheritance in class-based languages is a bad idea. If a class D inherits from a class C, you have an implementation dependency of C on D. This is called the *fragile base class problem*, because changes to C force reconsideration of D.

Chapter 21

Module Hierarchies

So far we have confined ourselves to considering "flat" modules consisting of a linear sequence of declarations of types, exceptions, and values. As programs grow in size and complexity, it becomes important to introduce further structuring mechanisms to support their growth. The ML module language also supports *module hierarchies*, tree-structured configurations of modules that reflect the architecture of a large system.

21.1 Substructures

A *substructure* is a "structure within a structure". Structures bindings (either opaque or transparent) are admitted as components of other structures. Structure specifications of the form

`structure` *strid* `:` *sigexp*

may appear in signatures. There is no distinction between transparent and opaque specifications in a signature, because there is no structure to ascribe!

The type checking and evaluation rules for structures are extended to substructures recursively. The principal signature of a sub-structure binding is determined according to the rules given in chapter 19. A substructure binding in one signature matches the corresponding one in another iff their signatures match according to the rules in chapter 19. Evaluation of a sub-structure binding consists of evaluating the structure expression, then binding the resulting structure value to that identifier.

To see how substructures arise in practice, consider the following programming scenario. The first version of a system makes use of a polymorphic dictionary data structure whose search keys are strings. The signature for such a data structure might be as follows:

```
signature MY_STRING_DICT =
  sig
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * string * 'a -> 'a dict
    val lookup : 'a dict * string -> 'a option
  end
```

The return type of `lookup` is `'a option`, since there may be no entry in the dictionary with the specified key.

The implementation of this abstraction looks approximately like this:

```
structure MyStringDict :> MY_STRING_DICT =
  struct
    datatype 'a dict =
      Empty |
      Node of 'a dict * string * 'a * 'a dict
    val empty = Empty
    fun insert (d, k, v) = ...
    fun lookup (d, k) = ...
  end
```

The omitted implementations of `insert` and `lookup` make use of the built-in lexicographic ordering of strings.

The second version of the system requires another dictionary whose keys are integers, leading to another signature and implementation for dictionaries.

```
signature MY_INT_DICT =
  sig
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * int * 'a -> 'a dict
    val lookup : 'a dict * int -> 'a option
  end
```

```
structure MyIntDict :> MY_INT_DICT =
  sig
    datatype 'a dict =
      Empty |
      Node of 'a dict * int * 'a * 'a dict
    val empty = Empty
    fun insert (d, k, v) = ...
    fun lookup (d, k) = ...
  end
```

The ellided implementations of `insert` and `lookup` make use of the primitive comparison operations on integers.

At this point we may observe an obvious pattern, that of a dictionary with keys of a specific type. To avoid further repetition we decide to abstract out the key type from the signature so that it can be filled in later.

```
signature MY_GEN_DICT =
  sig
    type key
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * key * 'a -> 'a dict
  end
```

Notice that the dictionary abstraction carries with it the type of its keys.

Specific instances of this generic dictionary signature are obtained using `where type`.

```
signature MY_STRING_DICT =
  MY_GEN_DICT where type key = string
signature MY_INT_DICT =
  MY_GEN_DICT where type key = int
```

A string dictionary might then be implemented as follows:

```
structure MyStringDict :> MY_STRING_DICT =
  struct
    type key = string
    datatype 'a dict =
      Empty |
```

```
      Node of 'a dict * key * 'a * 'a dict
    val empty = Empty
    fun insert (Empty, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
        if k < l then           (* string comparison *)
           lookup (dl, k)
        else if k > l then      (* string comparison *)
           lookup (dr, k)
        else
           v
  end
```

By a similar process we may build an implementation `MyIntDict` of the signature `MY_INT_DICT`, with integer keys ordered by the standard integer comparison operations.

Now suppose that we require a third dictionary, with integers as keys, but ordered according to the divisibility ordering.[1] This implementation, say `MyIntDivDict`, makes use of modular arithmetic to compare operations, but has the same signature `MY_INT_DICT` as `MyIntDict`.

```
structure MyIntDivDict :> MY_INT_DICT =
  struct
    type key = int
    datatype 'a dict =
      Empty |
      Node of 'a dict * key * 'a * 'a dict
    fun divides (k, l) = (l mod k = 0)
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
        if divides (k, l) then      (* divisibility test *)
           lookup (dl, k)
        else if divides (l, k) then (* divisibility test *)
           lookup (dr, k)
        else
```

[1] For which, $m < n$ iff m divides n evenly.

```
          v
  end
```

Notice that we required an auxiliary function, `divides`, to implement the comparison in the required sense.

With this in mind, let us re-consider our initial attempt to consolidate the signatures of the various versions of dictionaries in play. In one sense there is nothing to do — the signature `MY_GEN_DICT` suffices. However, as we've just seen, the instances of this signature, which are ascribed to particular implementations, do not determine the interpretation. What we'd like to do is to package the type with its interpretation so that the dictionary module is self-contained. Not only does the dictionary module carry with it the type of its keys, but it also carries the interpretation used on that type.

This is achieved by introducing a substructure binding in the dictionary structure. To begin with we first isolate the notion of an ordered type.

```
signature ORDERED =
  sig
    type t
    val lt : t * t -> bool
    val eq : t * t -> bool
  end
```

This signature describes modules that contain a type `t` equipped with an equality and comparison operation on it.

An implementation of this signature specifies the type and the interpretation, as in the following examples.

```
(* Lexicographically ordered strings. *)
structure LexString : ORDERED =
  struct
    type t = string
    val eq = (op =)
    val lt = (op <)
  end
(* Integers ordered conventionally. *)
structure LessInt : ORDERED =
```

```
  struct
    type t = int
    val eq = (op =)
    val lt = (op <)
  end
(* Integers ordered by divisibility.*)
structure DivInt : ORDERED =
  struct
    type t = int
    fun lt (m, n) = (n mod m = 0)
    fun eq (m, n) = lt (m, n) andalso lt (n, m)
  end
```

Notice that the use of transparent ascription is very natural here, since `ORDERED` is not intended as a self-contained abstraction.

The signature of dictionaries is re-structured as follows:

```
signature DICT =
  sig
    structure Key : ORDERED
    type 'a dict
    val empty : 'a dict
    val insert : 'a dict * Key.t * 'a -> 'a dict
    val lookup : 'a dict * Key.t -> 'a option
  end
```

The signature `DICT` includes as a substructure the key type together with its interpretation as an ordered type.

To enforce abstraction we introduce specialized versions of this signature that specify the key type using a `where type` clause.

```
signature STRING_DICT =
  DICT where type Key.t=string
signature INT_DICT =
  DICT where type Key.t=int
```

These are, respectively, signatures for the abstract type of dictionaries whose keys are strings and integers.

How are these signatures to be implemented? Corresponding to the layering of the signatures, we have a layering of the implementation.

```
structure StringDict :> STRING_DICT =
  struct
    structure Key : ORDERED = LexString
    datatype 'a dict =
      Empty |
      Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
        if Key.lt(k, l) then
           lookup (dl, k)
        else if Key.lt (l, k) then
           lookup (dr, k)
        else
           v
  end
```

Observe that the implementation of `insert` and `lookup` make use of the comparison operations `Key.lt` and `Key.eq`.

Similarly, we may implement `IntDict`, with the standard ordering, as follows:

```
structure LessIntDict :> INT_DICT =
  struct
    structure Key : ORDERED = LessInt
    datatype 'a dict =
      Empty |
      Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
        if Key.lt(k, l) then
           lookup (dl, k)
        else if Key.lt (l, k) then
           lookup (dr, k)
        else
           v
```

```
  end
```

Similarly, dictionaries with integer keys ordered by divisibility may be implemented as follows:

```
structure IntDivDict :> INT_DICT =
  struct
    structure Key : ORDERED = IntDiv
    datatype 'a dict =
      Empty |
      Node of 'a dict * Key.t * 'a * 'a dict
    val empty = Empty
    fun insert (None, k, v) = Node (Empty, k, v, Empty)
    fun lookup (Empty, _) = NONE
      | lookup (Node (dl, l, v, dr), k) =
        if Key.lt(k, l) then
           lookup (dl, k)
        else if Key.lt (l, k) then
           lookup (dr, k)
        else
           v
  end
```

Taking stock of the development, what we have done is to structure the signature of dictionaries to allow the type of keys, together with its interpretation, to vary from one implementation to another. The `Key` substructure may be viewed as a "parameter" of the signature `DICT` that is "instantiated" by specialization to specific types of interest. In this sense substructures subsume the notion of a *parameterized signature* found in some languages. There are several advantages to this:

1. A signature with one or more substructures is still a complete signature. Parameterized signatures, in contrast, are incomplete signatures that must be completed to be used.

2. Any substructure of a signature may play the role of a "parameter". There is no need to designate in advance which are "arguments" and which are "results".

In chapter 23 we will introduce the mechanisms needed to build a generic implementation of dictionaries that may be instantiated by the key type and its ordering.

21.2 Sample Code

Here is the code for this chapter.

Chapter 22

Sharing Specifications

In chapter 21 we illustrated the use of substructures to express the dependence of one abstraction on another. In this chapter we will consider the problem of *symmetric combination* of modules to form larger modules.

22.1 Combining Abstractions

The discussion will be based on a representation of geometry in ML based on the following (drastically simplified) signature.

```
signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
  end
```

For the purposes of this example, we have reduced geometry to two concepts, that of a point in space and that of a sphere.

Points and vectors are fundamental to representing geometry. They are described by the following (abbreviated) signatures:

```
signature VECTOR =
  sig
    type vector
    val zero : vector
    val scale : real * vector -> vector
```

```
    val add : vector * vector -> vector
    val dot : vector * vector -> real
  end
signature POINT =
  sig
    structure Vector : VECTOR
    type point
    (* move a point along a vector *)
    val translate : point * Vector.vector -> point
    (* the vector from a to b *)
    val ray : point * point -> Vector.vector
  end
```

The vector operations support addition, scalar multiplication, and inner product, and include a unit element for addition. The point operations support translation of a point along a vector and the creation of a vector as the "difference" of two points (*i.e.*, the vector from the first to the second).

Spheres are implemented by a module implementing the following (abbreviated) signature:

```
signature SPHERE =
  sig
    structure Vector : VECTOR
    structure Point : POINT
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
  end
```

The operation `sphere` creates a sphere centered at a given point and with the radius vector given.

These signatures are intentionally designed so that the dimension of the space is not part of the specification. This allows us — using the mechanisms to be introduced in chapter 23 — to build packages that work in an arbitrary dimension without requiring run-time conformance checks. It is the structures, and not the signatures, that specify the dimension. Two- and three-dimensional geometry are defined by structure bindings like these:

```
structure Geom2D :> GEOMETRY = ...
structure Geom3D :> GEOMETRY = ...
```

As a consequence of the use of opaque ascription, the types `Geom2D.Point.point` and `Geom3D.Point.point` are distinct. This means that dimensional conformance is enforced by the type checker. For example, we cannot apply `Geom3D.Sphere.sphere` to a point in two-space and a vector in three-space.

This is a good thing: the more static checking we have, the better off we are. Closer inspection reveals that, unfortunately, we have too much of a good thing. Suppose that p and q are two-dimensional points of type `Geom2D.Point.point`. We might expect to be able to form a sphere centered at p with radius determined by the vector from p to q:

```
Geom2D.Sphere.sphere (p, Geom2D.Point.ray (p, q)).
```

But this expression is ill-typed! The reason is that the types `Geom2D.Point.Vector` and `Geom2D.Sphere.Vector.vector` are *also* distinct from one another, which is not at all what we intend.

What has gone wrong? The situation is quite subtle. In keeping with the guidelines discussed in section 21.1, we have incorporated as substructures the structures on which a given structure depends. For example, forming a ray from one point to another yields a vector, so an implementation of `POINT` depends on an implementation of `VECTOR`. Thus, `POINT` has a substructure implementing `VECTOR`, and, similarly, `SPHERE` has substructures implementing `VECTOR` and `POINT`.

This leads to a proliferation of structures. Even in the very simplified geometry signature given above, we have two "copies" of the point abstraction, and three "copies" of the vector abstraction! Since we used opaque ascription to define the two- and three-dimensional implementations of the signature `GEOMETRY`, all of these abstractions are kept distinct from one another, even though they may be implemented identically.

In a sense this is the correct state of affairs. The various "copies" of, say, the vector abstraction might well be distinct from one another. In the elided implementation of two-dimensional geometry, we might have used completely incompatible notions of vector in each of the three places where they are required. Of course, this may not be what is intended, but (so far) there is *nothing in the signature to prevent it*. Hence, we are *compelled* to keep these types distinct.

What is missing is the expression of the intention that the various "copies" of vectors and points within the geometry abstraction be *identical*, so that we can mix-and-match the vectors constructed in various components of

the package. To support this it is necessary to constrain the implementation to use the same notion of vector throughout. This is achieved using a *type sharing constraint*. The revised signatures for the geometry package look like this:

```
signature SPHERE =
  sig
    structure Vector : VECTOR
    structure Point : POINT
    sharing type Point.Vector.vector = Vector.vector
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
  end
signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing type Point.point = Sphere.Point.point
             and  Point.Vector.vector = Sphere.Vector.vector
  end
```

These equations specify that the two "copies" of the point abstraction, and the three "copies" of the vector abstraction must coincide. In the presence of the above sharing specification, the ill-typed expression above becomes well-typed, since now the required type equation holds by explicit specification in the signature.

As a notational convenience we may use a *structure sharing constraint* instead to express the same requirements:

```
signature SPHERE =
  sig
    structure Vector : VECTOR
    structure Point : POINT
    sharing Point.Vector = Vector
    type sphere
    val sphere : Point.point * Vector.vector -> sphere
  end
signature GEOMETRY =
  sig
```

```
    structure Point : POINT
    structure Sphere : SPHERE
    sharing Point = Sphere.Point
        and Point.Vector = Sphere.Vector
  end
```

Rather than specify the required sharing type-by-type, we can instead specify it structure-by-structure, with the meaning that corresponding types of shared structures are required to share. Since each structure in our example contains only one type, the effect of the structure sharing specification above is identical to the preceding type sharing specification.

Not only does the sharing specification ensure that the desired equations hold amongst the various components of an implementation of `GEOMETRY`, but it also constrains the implementation to ensure that these types are the same. It is easy to achieve this requirement by defining a single implementation of points and vectors that is re-used in the higher-level abstractions.

```
structure Vector3D : VECTOR = ...
structure Point3D : POINT =
  struct
    structure Vector : VECTOR = Vector3D
    ⋮
  end
structure Sphere3D : SPHERE =
  struct
    structure Vector : VECTOR = Vector3D
    structure Point : POINT = Point3D
    ⋮
  end
structure Geom3D :> GEOMETRY =
  struct
    structure Point = Point3D
    structure Sphere = Sphere3D
  end
```

The required type sharing constraints are true by construction.

Had we instead replaced the above declaration of `Geom3D` by the following one, the type checker would reject it on the grounds that the re-

quired sharing between `Sphere.Point` and `Point` does not hold, because `Sphere2D.Point` is distinct from `Point3D`.

```
structure Geom3D :> GEOMETRY =
  struct
    structure Point = Point3D
    structure Sphere = Sphere2D
  end
```

It is natural to wonder whether it might be possible to restructure the `GEOMETRY` signature so that the duplication of the point and vector components is avoided, thereby obviating the need for sharing specifications. One *can* re-structure the code in this manner, but doing so would do violence to the overall structure of the program. This is why sharing specifications are so important.

Let's try to re-organize the signature `GEOMETRY` so that duplication of the point and vector structures is avoided. One step is to eliminate the substructure `Vector` from `SPHERE`, replacing uses of `Vector.vector` by `Vector.Point.vector`.

```
signature SPHERE =
  sig
    structure Point : POINT
    type sphere
    val sphere :
      Point.point * Point.Vector.vector -> sphere
  end
```

After all, since the structure `Point` comes equipped with a notion of vector, why not use it?

This cuts down the number of sharing specifications to one:

```
signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing Point = Sphere.Point
  end
```

If we could further eliminate the substructure `Point` from the signature `SPHERE` we would have only one copy of `Point` and no need for a sharing specification.

But what would the signature SPHERE look like in this case?

```
signature SPHERE =
  sig
    type sphere
    val sphere :
      Point.point * Point.Vector.vector -> sphere
  end
```

The problem now is that the signature SPHERE is no longer self-contained. It makes reference to a structure Point, but which Point are we talking about? Any commitment would tie the signature to a specific structure, and hence a specific dimension, contrary to our intentions. Rather, the notion of point must be a generic concept within SPHERE, and hence Point must appear as a substructure. The substructure Point may be thought of as a parameter of the signature SPHERE in the sense discussed earlier.

The only other move available to us is to eliminate the structure Point from the signature GEOMETRY. This is indeed possible, and would eliminate the need for any sharing specifications. But it only defers the problem, rather than solving it. A full-scale geometry package would contain more abstractions that involve points, so that there will still be copies in the other abstractions. Sharing specifications would then be required to ensure that these copies are, in fact, identical.

Here is an example. Let us introduce another geometric abstraction, the semi-space.

```
signature SEMI_SPACE =
  sig
    structure Point : POINT
    type semispace
    val side : Point.point * semispace -> bool option
  end
```

The function side determines (if possible) whether a given point lies in one half of the semi-space or the other.

The expanded GEOMETRY signature would look like this (with the elimination of the Point structure in place).

```
signature EXTD_GEOMETRY =
  sig
```

```
  structure Sphere : SPHERE
  structure SemiSpace : SEMI_SPACE
  sharing Sphere.Point = SemiSpace.Point
end
```

By an argument similar to the one we gave for the signature `SPHERE`, we cannot eliminate the substructure `Point` from the signature `SEMI_SPACE`, and hence we wind up with two copies. Therefore the sharing specification is required.

What is at issue here is a fundamental tension in the very notion of modular programming. On the one hand we wish to *separate* modules from one another so that they may be treated independently. This requires that the signatures of these modules be self-contained. Unbound references to a structure — such as `Point` — ties that signature to a specific implementation, in violation of our desire to treat modules separately from one another. On the other hand we wish to *combine* modules together to form programs. Doing so requires that the composition be coherent, which is achieved by the use of sharing specifications. What sharing specifications do for you is to provide an *after the fact* means of tying together several different abstractions to form a coherent whole. This approach to the problem of coherence is a unique — and uniquely effective — feature of the ML module system.

22.2 Sample Code

Here is the code for this chapter.

Chapter 23

Parameterization

To support code re-use it is useful to define *generic*, or *parameterized*, modules that leave unspecified some aspects of the implementation of a module. The unspecified parts may be *instantiated* to determine specific *instances* of the module. The common part is thereby implemented once and shared among all instances.

In ML such generic modules are called *functors*. A functor is a module-level function that takes a structure as argument and yields a structure as result. Instances are created by *applying* the functor to an argument specifying the interpretation of the parameters.

23.1 Functor Bindings and Applications

Functors are defined using a *functor binding*. There are two forms, the opaque and the transparent. A *transparent* functor has the form

`functor` *funid*`(`*decs*`):`*sigexp* `=` *strexp*

where the result signature, *sigexp*, is transparently ascribed; an *opaque* functor has the form

`functor` *funid*`(`*decs*`):>`*sigexp* `=` *strexp*

where the result signature is opaquely ascribed. A functor is a module-level function whose argument is a sequence of declarations, and whose result is a structure.

Type checking of a functor consists of checking that the functor body matches the ascribed result signature, given that the parameters of the functor have the specified signatures. For a transparent functor the result signature is the augmented signature determined by matching the principal signature of the body (relative to the assumptions governing the parameters) against the given result signature. For opaque functors the ascribed result signature is used as-is, without further augmentation by type equations.

In chapter 21 we developed a modular implementation of dictionaries in which the ordering of keys is made explicit as a substructure. The implementation of dictionaries is the same for each choice of order structure on the keys. This can be expressed by a single functor binding that defines the implementation of dictionaries parametrically in the choice of keys.

```
functor DictFun
  (structure K : ORDERED) :>
    DICT where type Key.t = K.t =
struct
  structure Key : ORDERED = K
  datatype 'a dict =
    Empty |
    Node of 'a dict * Key.t * 'a * 'a dict
  val empty = Empty
  fun insert (None, k, v) =
    Node (Empty, k, v, Empty)
  fun lookup (Empty, _) = NONE
    | lookup (Node (dl, l, v, dr), k) =
      if Key.lt(k, l) then
         lookup (dl, k)
      else if Key.lt (l, k) then
         lookup (dr, k)
      else
         v
end
```

The functor `DictFun` takes as argument a single structure specifiying the ordered key type as a structure implementing signature `ORDERED`. The opaquely ascribed result signature is

```
DICT where type Key.t = K.t.
```

This signature holds the type `'a dict` abstract, but specifies that the key type is the type `K.t` passed as argument to the functor. The body of the functor has been written so that the comparison operations are obtained from the `Key` substructure. This ensures that the dictionary code is independent of the choice of key type and ordering.

Instances of functors are obtained by *application*. A *functor application* has the form

funid(*binds*)

where *binds* is a sequence of bindings of the arguments of the functor.

The signature of a functor application is determined by the following procedure. We assume we are given the signatures of the functor parameters, and also the "true" result signature of the functor (the given signature for opaque functors, the augmented signature for the transparent functors).

1. For each argument, match the argument signature against the corresponding parameter signature of the functor. This determines an augmentation of the parameter signature for each argument (as described in chapter 20).

2. For each reference to a type component of a functor parameter in the result signature, propagate the type definitions of the augmented parameter signature to the result signature.

The signature of the application determined by this procedure is then *opaquely ascribed* to the application. This means that if a type is left abstract in the result signature of a functor, that type is "new" in every instance of that functor. This behavior is called *generativity* of the functor.[1]

Returning to the example of the dictionary functor, the three versions of dictionaries considered in chapter 21 may be obtained by applying `DictFun` to appropriate arguments.

```
structure LtIntDict = DictFun (structure K = LessInt)
structure LexStringDict = DictFun (structure K = LexString)
structure DivIntDict = DictFun (structure K = DivInt)
```

[1]The alternative, called *applicativity*, means that there is *one* abstract type shared by *all* instances of that functor.

In each case the functor DictFun is instantiated by specifying a binding for its argument stucture K. The argument structures are, as described in chapter 21, implementations of the signature ORDERED. They specify the type of keys and the sense in which they are ordered.

The signatures for the structures LtIntDict, LexStringDict, and DivIntDict are determined by instantiating the result signature of the functor DictFun according to the above procedure. Consider the application of DictFun to LtIntDict. The augmented signature resulting from matching the signature of LtIntDict against the parameter signature ORDERED is the signature

```
ORDERED where type t=int
```

Assigning this to the parameter K, we deduce that the type K.t is equivalent to int, and hence the result signature of DictFun is

```
DICT where type Key.t = int
```

so that IntLtDict.Key.t is equivalent to int, as desired. By a similar process we deduce that the signature of LexStringDict is

```
DICT where type Key.t = string
```

and that the signature of DivIntDict is

```
DICT where type Key.t = int.
```

23.2 Functors and Sharing Specifications

In chapter 22 we developed a signature of geometric primitives that contained sharing specifications to ensure that the constituent abstractions may be combined properly. The signature GEOMETRY is defined as follows:

```
signature GEOMETRY =
  sig
    structure Point : POINT
    structure Sphere : SPHERE
    sharing Point = Sphere.Point
        and Point.Vector = Sphere.Vector
        and Sphere.Vector = Sphere.Point.Vector
  end
```

The `sharing` clauses ensure that the `Point` and `Sphere` components are compatible with each other.

Since we expect to define vectors, points, and spheres of various dimensions, it makes sense to implement these as functors, according to the following scheme:

```
functor PointFun
  (structure V : VECTOR) : POINT = ...
functor SphereFun
  (structure V : VECTOR
   structure P : POINT) : SPHERE =
struct
  structure Vector = V
  structure Point = P
  ⋮
end
functor GeomFun
  (structure P : POINT
   structure S : SPHERE) : GEOMETRY =
struct
  structure Point = P
  structure Sphere = S
end
```

A two-dimensional geometry package may then be defined as follows:

```
structure Vector2D : VECTOR = ...
structure Point2D : POINT =
  PointFun (structure V = Vector2D)
structure Sphere2D : SPHERE =
  SphereFun (structure V = Vector2D and P = Point2D)
structure Geom2D : GEOMETRY =
  GeomFun (structure P = Point2D and S = Sphere2D)
```

A three-dimensional version is defined similarly.

There is only one problem: the functors `SphereFun` and `GeomFun` are not well-typed! The reason is that in both cases their result signatures require type equations that are not true of their parameters! For example,

the signature SPHERE requires that Point.Vector be the same as Vector, which is not satisfied by the body of SphereFun. For these to be true, the structures P.Vector and V must be equivalent. This is not true in general, because the functors might be applied to arguments for which this is false. Similar problems plague the functor GeomFun. The solution is to include sharing constraints in the parameter list of the functors, as follows:

```
functor SphereFun
  (structure V : VECTOR
   structure P : POINT
   sharing P.Vector = V) : SPHERE =
struct
  structure Vector = V
  structure Point = P
  ⋮
end
functor GeomFun
  (structure P : POINT
   structure S : SPHERE
   sharing P.Vector = S.Vector and P = S.Point) : GEOMETRY =
struct
  structure Point = P
  structure Sphere = S
end
```

These equations preclude instantiations for which the required equations do not hold, and are sufficient to ensure that the requirements of the result signatures of the functors are met.

23.3 Avoiding Sharing Specifications

As with sharing specifications in signatures, it is natural to wonder whether they can be avoided in functor parameters. Once again, the answer is "yes", but doing so does violence to the structure of your program. The chief virtue of sharing specifications is that they express directly and concisely the required relationships without requiring that these relationships be anticipated when defining the signatures of the parameters. This greatly

facilitates re-use of off-the-shelf code, for which it is impossible to assume any sharing relationships that one may wish to impose in an application.

To see what happens, let's consider the best-case scenario from chapter 22 in which we have minimized sharing specifications to one tying together the sphere and semispace components. That is, we're to implement the following signature:

```
signature EXTD_GEOMETRY =
  sig
    structure Sphere : SPHERE
    structure SemiSpace : SEMI_SPACE
    sharing Sphere.Point = SemiSpace.Point
  end
```

The implementation is a functor of the form

```
functor ExtdGeomFun
  (structure Sp : SPHERE
   structure Ss : SEMI_SPACE
   sharing Sphere.Point = SemiSpace.Point) =
struct
  structure Sphere = Sp
  structure SemiSpace = Ss
end
```

To eliminate the sharing equation in the functor parameter, we must arrange that the sharing equation in the signature `EXTD_GEOMETRY` holds. Simply dropping the sharing specification will not do, because then there is no reason to believe that it will hold as required in the signature. A natural move is to "factor out" the implementation of `POINT`, and use it to ensure that the required equation is true of the functor body. There are two methods for doing this, each with disadvtanges compared to the use of sharing specifications.

One is to make the desired equation true by construction. Rather than take implementations of `SPHERE` and `SEMI_SPACE` as arguments, the functor `ExtdGeomFun` takes only an implementation of `POINT`, then creates in the functor body appropriate implementations of spheres and semi-spaces.

```
functor SphereFun
  (structure P : POINT) : SPHERE =
```

```
struct
  structure Vector = P.Vector
  structure Point = P
  ⋮
end
functor SemiSpaceFun
  (structure P : POINT) : SEMI_SPACE =
struct
  ⋮
end
functor ExtdGeomFun1
  (structure P : POINT) : GEOMETRY =
struct
  structure Sphere =
    SphereFun (structure P = Point)
  structure SemiSpace =
    SemiSpaceFun (structure P = Point)
end
```

The problems with this solution are these:

- The body of `ExtdGeomFun1` makes use of the functors `SphereFun` and `SemiSpaceFun`. In effect we are limiting the geometry functor to arguments that are built from these specific functors, and no other. This is a significant loss of generality that is otherwise present in the functor `ExtdGeomFun`, which may be applied to *any* implementations of `SPHERE` and `SEMI_SPACE`.

- The functor `ExtdGeomFun1` must have as parameter the common element(s) of the components of its body, which is then used to build up the appropriate substructures in a manner consistent with the required sharing. This approach does not scale well when many abstractions are layered atop one another. We must reconstruct the entire hierarchy, starting with the components that are conceptually "furthest away" as arguments.

- There is no inherent reason why `ExtdGeomFun1` must take an implementation of `POINT` as argument. It does so only so that it can recon-

struct the hierarchy so as to satisfy the sharing requirements of the result signature.

Another approach is to factor out the common component, and use this to constrain the arguments to the functor to ensure that the possible arguments are limited to situations for which the required sharing holds.

```
functor ExtdGeomFun2
  (structure P : POINT
   structure Sp : SPHERE where Point = P
   structure Ss : SEMI_SPACE where Point = P) =
struct
  structure Sphere = Sp
  structure SemiSpace = Ss
end
```

Now the required sharing requirements are met, but it is also clear that this approach has no particular advantages over just using a sharing specification. It has the disadvantage of requiring a third argument, whose only role is to make it possible to express the required sharing. An application of this functor must provide not only implementations of `SPHERE` and `SEMI_SPACE`, but also an implementation of `POINT` that is used to build these!

A slightly more sophisticated version of this solution is as follows:

```
functor ExtdGeomFun3
  (structure Sp : SPHERE
   structure Ss : SEMI_SPACE where Point = Sp.Point) =
struct
  structure Sphere = Sp
  structure SemiSpace = Ss
end
```

The "extra" parameter to the functor has been eliminated by choosing one of the components as a "representative" and insisting that the others be compatible with it by using a `where` clause.[2]

[2]Officially, we must write `where type Point.point = Sp.Point.point`, but many compilers accept the syntax above.

This solution has all of the advantages of the direct use of sharing specifications, and no further disadvantages. However, we are forced to violate arbitrarily the inherent symmetry of the situation. We could just as well have written

```
functor ExtdGeomFun4
  (structure Ss : SEMI_SPACE
   structure Sp : SPHERE where Point = Sp.Point) =
struct
  structure Sphere = Sp
  structure SemiSpace = Ss
end
```

without changing the meaning.

Here is the point: *sharing specifications allow a symmetric situation to be treated in a symmetric manner*. The compiler breaks the symmetry by choosing representatives arbitrarily in the manner illustrated above. Sharing specifications off-load the burden of making such tedious (because arbitrary) decisions to the compiler, rather than imposing it on the programmer.

23.4 Sample Code

Here is the code for this chapter.

Part IV

Programming Techniques

In this part of the book we will explore the use of Standard ML to build elegant, reliable, and efficient programs. The discussion takes the form of a series of worked examples illustrating various techniques for building programs.

Chapter 24

Specifications and Correctness

The most important tools for getting programs right are *specification* and *verification*. In this Chapter we review the main ideas in preparation for their subsequent use in the rest of the book.

24.1 Specifications

A *specification* is a description of the behavior of a piece of code. Specifications take many forms:

- *Typing*. A type specification describes the "form" of the value of an expression, without saying anything about the value itself.
- *Effect Behavior*. An effect specification resembles a type specification, but instead of describing the value of an expression, it describes the effects it may engender when evaluated.
- *Input-Output Behavior*. An input-output specification is a mathematical formula, usually an implication, that describes the output of a function for all inputs satisfying some assumptions.
- *Time and Space Complexity*. A complexity specification states the time or space required to evaluate an expression. The specification is most often stated asymptotically in terms of the number of execution steps or the size of a data structure.

- *Equational Specifications*. An equational specification states that one code fragment is equivalent to another. This means that wherever one is used we may replace it with the other without changing the observable behavior of any program in which they occur.

This list by no means exhausts the possibilities. What specifications have in common, however, is a *descriptive*, or *declarative*, flavor, rather than a *prescriptive*, or *operational*, one. A specification states *what* a piece of code does, not *how* it does it.

Good programmers use specifications to state precisely and concisely their intentions when writing a piece of code. The code is written to solve a problem; the specification records for future reference what problem the code was intended to solve. The very act of formulating a precise specification of a piece of code is often the key to finding a good solution to a tricky problem. The guiding principle is this: *if you are unable to write a clear specification, you do not understand the problem well enough to solve it correctly*.

The greatest difficulty in using specifications is in knowing what to say. A common misunderstanding is that a given piece of code has one specification stating all there is to know about it. Rather you should see a specification as describing one of perhaps many properties of interest. In ML every program comes with at least one specification, its type. But we may also consider other specifications, according to interest and need. Sometimes only relatively weak properties are important — the function `square` always yields a non-negative result. Other times stronger properties are needed — the function `fibb` applied to `n` yields the `n`th and `n-1`st Fibonacci number. It's a matter of taste and experience to know what to say and how best to say it.

Recall the following two ML functions from chapter 7:

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib (n-1) + fib (n-2)
fun fib' 0 = (1, 0)
  | fib' 1 = (1, 1)
  | fib' n =
    let
      val (a, b) = fib' (n-1)
```

```
in
  (a+b, a)
end
```

Here are some specifications pertaining to these functions:

- Type specifications:
  - `val fib : int -> int`
  - `val fib' : int -> int * int`
- Effect specifications:
  - The application `fib` n may raise the exception `Overflow`.
  - The application `fib'` n may raise the exception `Overflow`.
- Input-output specifications:
  - If $n \geq 0$, then `fib` n evaluates to the nth Fibonacci number.
  - If $n \geq 0$, then `fib'` n evaluates the nth and $n-1$st Fibonacci number, in that order.
- Time complexity specifications:
  - If $n \geq 0$, then `fib` n terminates in $O(2^n)$ steps.
  - If $n \geq 0$, then `fib'` n terminates in $O(n)$ steps.
- Equivalence specification:

  For all $n \geq 0$, `fib` n is equivalent to `#1(fib'` n`)`

24.2 Correctness Proofs

A program *satisfies*, or *meets*, a specification iff its execution behavior is as described by the specification. *Verification* is the process of checking that a program satisfies a specification. This takes the form of proving a mathematical theorem (by hand or with machine assistance) stating that the program implements a specification.

There are many misunderstandings in the literature about specification and verification. It is worthwhile to take time out to address some of them here.

It is often said that a program is *correct* if it meets a specification. The verification of this fact is then called a *correctness proof*. While there is nothing wrong with this usage, it invites misinterpretation. As we remarked in section 24.1, there is in general no single preferred specification of a piece of code. Verification is always relative to a specification, and hence so is correctness. In particular, a program can be "correct" with respect to one specification, and "incorrect" with respect to another. Consequently, a program is never inherently correct or incorrect; it is only so relative to a particular specification.

A related misunderstanding is the inference that a program "works" from the fact that it is "correct" in this sense. Specifications usually make assumptions about the context in which the program is used. If these assumptions are not satisfied, then all bets are off — nothing can be said about its behavior. For this reason it is entirely possible that a correct program will malfunction when used, not because the specification is not *met*, but rather because the specification is not *relevant* to the context in which the code is used.

Another common misconception about specifications is that they can always be implemented as run-time checks.[1] There are at least two fallacies here:

1. The specification is stated at the level of the source code. It may not even be possible to test it at run-time. See chapter 32 for further discussion of this point.

2. The specification may not be mechanically checkable. For example, we might specify that a function `f` of type `int->int` always yields a non-negative result. But there is *no way* to implement this as a run-time check — this is an undecidable, or uncomputable, problem.

For this reason specifications are strictly more powerful than run-time checks. Correspondingly, it takes more work than a mere conditional

[1] This misconception is encouraged by the C `assert` macro, which introduces an executable test that a certain computable condition holds. This is a fine thing, but from this many people draw the conclusion that assertions (specifications) are simply boolean tests. This is false.

branch to ensure that a program satisfies a specification.

Finally, it is important to note that specification, implementation, and verification go hand-in-hand. It is unrealistic to propose to verify that an arbitrary piece of code satisfies an arbitrary specification. Fundamental computability and complexity results make clear that we can never succeed in such an endeavor. Fortunately, it is also completely artificial. In practice we specify, code, and verify simultaneously, with each activity informing the other. If the verification breaks down, re-consider the code or the specification (or both). If the code is difficult to write, look for insights from the specification and verification. If the specification is complex, rough out the code and proof to see what it should be. There is no "magic bullet", but these are some of the most important, and useful, tools for building elegant, robust programs.

Verifications of specifications take many different forms.

- Type specifications are verified automatically by the ML compiler. If you put type annotations on a function stating the types of the parameters or results of that function, the correctness of these annotations is ensured by the compiler. For example, we may state the intended typing of `fib` as follows:

  ```
  fun fib (n:int):int =
    case n
      of 0 => 1
       | 1 => 1
       | n => fib (n-1) + fib (n-2)
  ```

  This ensures that the type of `fib` is `int->int`.

- Effect specifications must be checked by hand. These generally state that a piece of code "may raise" one or more exceptions. If an exception is not mentioned in such a specification, we cannot conclude that the code does not raise it, only that we have no information. Notice that a handler serves to eliminate a "may raise" specification. For example, the following function may *not* raise the exception `Overflow`, even though `fib` might:

  ```
  fun protected_fib n =
      (fib n) handle Overflow => 0
  ```

A handler makes it possible to specify that an exception *may not* be raised by a given expression (because the handler traps it).

- Input-output specifications require proof, typically using some form of induction. For example, in chapter 7 we proved that `fib` n yields the nth Fibonacci number by complete induction on n.
- Complexity specifications are often verified by solving a recurrence describing the execution time of a program. In the case of `fib` we may read off the following recurrence:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ T(n+2) &= T(n) + T(n+1) + 1 \end{aligned}$$

  Solving this recurrence yields the proof that $T(n) = O(2^n)$.
- Equivalence specifications also require proof. Since equivalence of expressions must account for all possible uses of them, these proofs are, in general, very tricky. One method that often works is "induction plus hand-simulation". For example, it is not hard to prove by induction on n that `fib` n is equivalent to `#1(fib'` n`)`. First, plug in $n = 0$ and $n = 1$, and calculate using the definitions. Then assume the result for n and $n+1$, and consider $n+2$, once again calculating based on the definitions, to obtain the result.

24.3 Enforcement and Compliance

Most specifications have the form of an implication: *if* certain conditions are met, *then* the program behaves a certain way. For example, type specifications are conditional on the types of its free variables. For example, if `x` has type `int`, then `x+1` has type `int`. Input-output specifications are characteristically of this form. For example, if `x` is non-negative, then so is `x+1`.

Just as in ordinary mathematical reasoning, if the premises of such a specification are not true, then all bets are off — nothing can be said about the behavior of the code. The premises of the specification are *pre-conditions* that must be met in order for the program to behave in the manner described by the conclusion, or *post-condition*, of the specification. This

means that the pre-conditions impose obligations on the *caller*, the user of the code, in order for the *callee*, the code itself, to be well-behaved. A conditional specification is a *contract* between the caller and the callee: if the caller meets the pre-conditions, the caller promises to fulfill the post-condition.

In the case of type specifications the compiler enforces this obligation by ruling out as ill-typed any attempt to use a piece of code in a context that does not fulfill its typing assumptions. Returning to the example above, if one attempts to use the expression `x+1` in a context where `x` is not an integer, one can hardly expect that `x+1` will yield an integer. Therefore it is rejected by the type checker as a violation of the stated assumptions governing the types of its free variables.

What about specifications that are not mechanically enforced? For example, if `x` is negative, then we cannot infer anything about `x+1` from the specification given above.[2] To make use of the specification in reasoning about its used in a larger program, it is essential that this pre-condition be met in the context of its use.

Lacking mechanical enforcement of these obligations, it is all too easy to neglect them when writing code. Many programming mistakes can be traced to violation of assumptions made by the callee that are not met by the caller.[3] What can be done about this?

A standard method, called *bullet-proofing*, is to augment the callee with run-time checks that ensure that its pre-conditions are met, raising an exception if they are not. For example, we might write a "bullet-proofed" version of `fib` that ensures that its argument is non-negative as follows:

```
local
  exception PreCond
  fun unchecked_fib 0 = 1
    | unchecked_fib 1 = 1
    | unchecked_fib n =
      unchecked_fib (n-1) + unchecked_fib (n-2)
in
```

[2] There are obviously *other* specifications that carry more information, but we're only concerned here with the one given. Moreover, if f is an unknown function, then we will, in general, only have the specification, and not the code, to reason about.

[3] Sadly, these assumptions are often unstated and can only be culled from the code with great effort, if at all.

```
  fun checked_fib n =
    if n < 0 then
       raise PreCond
    else
       unchecked_fib n
end
```

It is worth noting that we have structured this program to take the pre-condition check out of the loop. It would be poor practice to define `checked_fib` as follows:

```
fun bad_checked_fib n =
  if n < 0 then
     raise PreCond
  else
     case n
       of 0 => 1
        | 1 => 1
        | n => bad_checked_fib (n-1) + bad_checked_fib (n-2)
```

Once we know that the initial argument is non-negative, it is assured that recursive calls also satisfy this requirement, provided that you've done the inductive reasoning to validate the specification of the function.

However, bullet-proofing in this form has several drawbacks. First, it imposes the overhead of checking on *all* callers, even those that have ensured that the desired pre-condition is true. In truth the run-time overhead is minor; the real overhead is requiring that the implementor of the callee take the trouble to impose the checks.

Second, and far more importantly, *bullet-proofing only applies to specifications that can be checked at run-time*. As we remarked earlier, not all specifications are amenable to run-time checks. For these cases there is no question of inserting run-time checks to enforce the pre-condition. For example, we may wish to impose the requirement that a function argument of type `int->int` always yields a non-negative result. There is no run-time check for this condition — we cannot write a function `nonneg` of type `(int->int)->bool` that determines whether or not a function f always yields a non-negative result. In chapter 32 we will consider the use of data abstraction to enforce at compile time specifications that may not be checked at run-time.

Chapter 25

Induction and Recursion

This chapter is concerned with the close relationship between *recursion*and *induction* in programming. If a function is recursively-defined, an inductive proof is required to show that it meets a specification of its behavior. The motto is

> *when programming recursively, think inductively*.

Doing so significantly reduces the time spent debugging, and often leads to more efficient, robust, and elegant programs.

25.1 Exponentiation

Let's start with a very simple series of examples, all involving the computation of the integer exponential function. Our first example is to compute 2^n for integers $n \geq 0$. We seek to define the function `exp` of type `int->int` satisfying the specification

> *if* $n \geq 0$, *then* ***`exp`*** *n evaluates to* 2^n.

The *precondition*, or *assumption*, is that the argument n is non-negative. The *postcondition*, or *guarantee*, is that the result of applying `exp` to n is the number 2^n. The caller is required to establish the precondition before applying `exp`; in exchange, the caller may assume that the result is 2^n.

Here's the code:

```
fun exp 0 = 1
  | exp n = 2 * exp (n-1)
```

Does this function satisfy the specification? It does, and we can prove this by induction on n. If $n = 0$, then `exp` n evaluates to 1 (as you can see from the first line of its definition), which is, of course, 2^0. Otherwise, assume that exp is correct for $n - 1 \geq 0$, and consider the value of `exp` n. From the second line of its definition we can see that this is the value of $2 \times p$, where p is the value of `exp` $(n - 1)$. Inductively, $p \geq 2^{n-1}$, so $2 \times p = 2 \times 2^{n-1} = 2^n$, as desired. Notice that we need not consider arguments $n < 0$ since the precondition of the specification requires that this be so. We must, however, ensure that each recursive call satisfies this requirement in order to apply the inductive hypothesis.

That was pretty simple. Now let us consider the running time of `exp` expressed as a function of n. Assuming that arithmetic operations are executed in constant time, then we can read off a recurrence describing its execution time as follows:

$$\begin{aligned} T(0) &= O(1) \\ T(n+1) &= O(1) + T(n) \end{aligned}$$

We are interested in *solving* a recurrence by finding a closed-form expression for it. In this case the solution is easily obtained:

$$T(n) = O(n)$$

Thus we have a *linear time* algorithm for computing the integer exponential function.

What about space? This is a much more subtle issue than time because it is much more difficult in a high-level language such as ML to see where the space is used. Based on our earlier discussions of recursion and iteration we can argue informally that the definition of `exp` given above requires space given by the following recurrence:

$$\begin{aligned} S(0) &= O(1) \\ S(n+1) &= O(1) + S(n) \end{aligned}$$

The justification is that the implementation requires a constant amount of storage to record the pending multiplication that must be performed upon completion of the recursive call.

Solving this simple recurrence yields the equation

$$S(n) = O(n)$$

expressing that `exp` is also a *linear space* algorithm for the integer exponential function.

Can we do better? Yes, on both counts! Here's how. Rather than count down by one's, multiplying by two at each stage, we use successive squaring to achieve logarithmic time and space requirements. The idea is that if the exponent is even, we square the result of raising 2 to half the given power; otherwise, we reduce the exponent by one and double the result, ensuring that the next exponent will be even. Here's the code:

```
fun square (n:int) = n*n
fun double (n:int) = n+n

fun fast_exp 0 = 1
  | fast_exp n =
    if n mod 2 = 0 then
       square (fast_exp (n div 2))
    else
       double (fast_exp (n-1))
```

Its specification is precisely the same as before. Does this code satisfy the specification? Yes, and we can prove this by using *complete induction*, a form of mathematical induction in which we may prove that $n > 0$ has a desired property by assuming not only that the predecessor has it, but that *all preceding numbers* have it, and arguing that therefore n must have it. Here's how it's done. For $n = 0$ the argument is exactly as before. Suppose, then, that $n > 0$. If $n$is even, the value of `exp` n is the result of squaring the value of `exp` $(n \div 2)$. Inductively this value is $2^{(n \div 2)}$, so squaring it yields $2^{(ndiv2)} \times 2^{(n \div 2)} = 2^{2 \times (n \div 2)} = 2^n$, as required. If, on the other hand, n is odd, the value is the result of doubling `exp` $(n-1)$. Inductively the latter value is $2^{(n-1)}$, so doubling it yields 2^n, as required.

Here's a recurrence governing the running time of `fast_exp` as a function of its argument:

$$\begin{aligned} T(0) &= O(1) \\ T(2n) &= O(1) + T(n) \\ T(2n+1) &= O(1) + T(2n) \\ &= O(1) + T(n) \end{aligned}$$

Solving this recurrence using standard techniques yields the solution

$$T(n) = O(lgn)$$

You should convince yourself that `fast_exp` also requires logarithmic space usage.

Can we do better? Well, it's not possible to improve the time requirement (at least not asymptotically), but we can reduce the space required to $O(1)$ by putting the function into iterative (tail recursive) form. However, this may not be achieved in this case by simply adding an accumulator argument, without also increasing the running time! The obvious approach is to attempt to satisfy the specification

if $n \geq 0$, then `skinny_fast_exp (n, a)` *evaluates to* $2^n \times a$.

Here's some code that achieves this specification:

```
fun skinny_fast_exp (0, a) = a
  | skinny_fast_exp (n, a) =
    if n mod 2 = 0 then
       skinny_fast_exp (n div 2,
         skinny_fast_exp (n div 2, a))
    else
       skinny_fast_exp (n-1, 2*a)
```

It is easy to see that this code works properly for $n = 0$ and for $n > 0$ when n is odd, but what if $n > 0$ is even? Then by induction we compute $2^{(n \div 2)} \times 2^{(n \div 2)} \times a$ by two recursive calls to `skinny_fast_exp`.

This yields the desired result, but what is the running time? Here's a recurrence to describe its running time as a function of n:

$$\begin{aligned} T(0) &= 1 \\ T(2n) &= O(1) + 2T(n) \\ T(2n+1) &= O(1) + T(2n) \\ &= O(1) + 2T(n) \end{aligned}$$

Here again we have a standard recurrence whose solution is

$$T(n) = O(nlgn).$$

Yuck! Can we do better? The key is to recall the following important fact:

$$2^{(2n)} = (2^2)^n = 4^n.$$

We can achieve a logarithmic time and exponential space bound by a *change of base*. Here's the specification:

if $n \geq 0$, then `gen_skinny_fast_exp (b, n, a)` *evaluates to $b^n \times a$.*

Here's the code:

```
fun gen_skinny_fast_exp (b, 0, a) = a
  | gen_skinny_fast_exp (b, n, a) =
    if n mod 2 = 0 then
       gen_skinny_fast_exp (b*b, n div 2, a)
    else
       gen_skinny_fast_exp (b, n - 1, b * a)
```

Let's check its correctness by complete induction on n. The base case is obvious. Assume the specification for arguments smaller than $n > 0$. If n is even, then by induction the result is $(b \times b)^{(n \div 2)} \times a = b^n \times a$, and if n is odd, we obtain inductively the result $b^{(n-1)} \times b \times a = b^n \times a$. This completes the proof.

The trick to achieving an efficient implementation of the exponential function was to compute a more general function that can be implemented using less time and space. Since this is a trick employed by the implementor of the exponential function, it is important to insulate the client from it. This is easily achieved by using a `local` declaration to "hide" the generalized function, making available to the caller a function satisfying the original specification. Here's what the code looks like in this case:

```
local
  fun gen_skinny_fast_exp (b, 0, a) =
    | gen_skinny_fast_exp (b, n, a) = ...
in
  fun exp n = gen_skinny_fast_exp (2, n, 1)
end
```

(The ellided code is the same as above.) The point here is to see how induction and recursion go hand-in-hand, and how we used induction not only to verify programs after-the-fact, but, more importantly, to help discover the program in the first place. If the verification is performed simultaneously with the coding, it is far more likely that the proof will go through and the program will work the first time you run it.

It is important to notice the correspondence between strengthening the specification by adding additional assumptions (and guarantees) and

adding accumulator arguments. What we observe is the apparent paradox that it is often *easier* to do something (superficially) *harder*! In terms of proving, it is often easier to push through an inductive argument for a stronger specification, precisely because we get to assume the result as the inductive hypothesis when arguing the inductive step(s). We are limited only by the requirement that the specification be proved outright at the base case(s); no inductive assumption is available to help us along here. In terms of programming, it is often easier to compute a more complicated function involving accumulator arguments, precisely because we get to exploit the accumulator when making recursive calls. We are limited only by the requirement that the result be defined outright for the base case(s); no recursive calls are available to help us along here.

25.2 The GCD Algorithm

Let's consider a more complicated example, the computation of the greatest common divisor of a pair of non-negative integers. Recall that *m is a divisor of n*, written $m|n$, iff n is a multiple of m, which is to say that there is some $k \geq 0$ such that $n = k \times m$. The *greatest common divisor* of non-negative integers m and n is the largest p such that $p|m$ and $p|n$. (By convention the g.c.d. of 0 and 0 is taken to be 0.) Here's the specification of the *gcd*function:

if $m, n \geq 0$, then `gcd(m,n)` evaluates to the g.c.d. of m and n.

Euclid's algorithm for computing the g.c.d. of m and$n$ is defined by complete induction on the product mn. Here's the algorithm, written in ML:

```
fun gcd (m:int, 0):int = m
  | gcd (0, n:int):int = n
  | gcd (m:int, n:int):int =
    if m>n then
       gcd (m mod n, n)
    else
       gcd (m, n mod m)
```

Why is this algorithm correct? We may prove that `gcd` satisfies the specification by complete induction on the product $m \times n$. If $m \times n$ is zero,

then either $m$or n is zero, in which case the answer is, correctly, the other number. Otherwise the product is positive, and we proceed according to whether $m > n$ or $m \leq n$. Suppose that $m > n$. Observe that $mmodn = m - (m \div n) \times n$, so that $(mmodn) \times n = m \times n - (m \div n)n^2 < m \times n$, so that by induction we return the g.c.d. of $mmodn$ and n. It remains to show that this is the g.c.d. of m and n. If d divides both $mmodn$ and n, then $k \times d = (mmodn) = (m - (m \div n) \times n)$and $l \times d = n$ for some non-negative k and l. Consequently, $k \times d = m - (m \div n) \times l \times d$, so $m = (k + (m \div n) \times l) \times d$, which is to say that d divides m. Now if d' is any other divisor of m and n, then it is also a divisor of $(mmodn)$ and n, so $d > d'$. That is, d is the g.c.d. of m and n. The other case, $m \leq n$, follows similarly. This completes the proof.

At this point you may well be thinking that all this inductive reasoning is surely helpful, but it's no replacement for good old-fashioned "bullet-proofing" — conditional tests inserted at critical junctures to ensure that key invariants do indeed hold at execution time. Sure, you may be thinking, these checks have a run-time cost, but they can be turned off once the code is in production, and anyway the cost is minimal compared to, say, the time required to read and write from disk. It's hard to complain about this attitude, provided that sufficiently cheap checks *can* be put into place and provided that you know *where* to put them to maximize their effectiveness. For example, there's no use checking $i > 0$ at the start of the `then` clause of a test for $i > 0$. Barring compiler bugs, it can't possibly be anything other than the case at that point in the program. Or it may be possible to insert a check whose computation is more expensive (or more complicated) than the one we're trying to perform, in which case we're defeating the purpose by including them!

This raises the question of where should we put such checks, and what checks should be included to help ensure the correct operation (or, at least, graceful malfunction) of our programs? This is an instance of the general problem of writing *self-checking programs*. We'll illustrate the idea by elaborating on the g.c.d. example a bit further. Suppose we wish to write a self-checking g.c.d. algorithm that computes the g.c.d., and then checks the result to ensure that it really is the greatest common divisor of the two given non-negative integers before returning it as result. The code might look something like this:

```
exception GCD_ERROR
```

```
fun checked_gcd (m, n) =
    let
        val d = gcd (m, n)
    in
        if m mod d = 0 andalso
           n mod d = 0 andalso ???
        then
           d
        else
           raise GCD_ERROR
    end
```

It's obviously no problem to check that putative g.c.d., d, is in fact a common divisor of $m$and n, but how do we check that it's the *greatest* common divisor? Well, one choice is to simply try all numbers between d and the smaller of m and n to ensure that no intervening number is also a divisor, refuting the maximality of d. But this is clearly so inefficient as to be impractical. But there's a better way, which, it has to be emphasized, relies on the kind of mathematical reasoning we've been considering right along. Here's an important fact:

> *d is the g.c.d. of m and n iff d divides both m and n and can be written as a linear combination of m and n.*

That is, d is the g.c.d. of $m$and n iff $m = k \times d$ for some $k \geq 0$, $n = l \times d$ for some $l \geq 0$, and $d = a \times m + b \times n$ for some integers (possibly negative!) a and b. We'll prove this constructively by giving a program to compute not only the g.c.d. d of m and n, but also the coefficients a and b such that $d = a \times m + b \times n$. Here's the specification:

> *if $m, n \geq 0$, then `ggcd (m, n)` evaluates to `(d, a, b)` such that d divides m, d divides n, and $d = a \times m + b \times n$; consequently, d is the g.c.d. of m and n.*

And here's the code to compute it:

```
fun ggcd (0, n) = (n, 0, 1)
  | ggcd (m, 0) = (m, 1, 0)
  | ggcd (m, n) =
    if m>n then
```

```
        let
            val (d, a, b) = ggcd (m mod n, n)
        in
            (d, a, b - a * (m div n))
        end
    else
        let
            val (d, a, b) = ggcd (m, n mod m)
        in
            (d, a - b*(n div m), b)
        end
```

We may easily check that this code satisfies the specification by induction on the product $m \times n$. If $m \times n = 0$, then either m or n is 0, in which case the result follows immediately. Otherwise assume the result for smaller products, and show it for $m \times n > 0$. Suppose $m > n$; the other case is handled analogously. Inductively we obtain d, a, and b such that d is the g.c.d. of $mmodn$ and n, and hence is the g.c.d. of m and n, and $d = a \times (mmodn) + b \times n$. Since $mmodn = m - (m \div n) \times n$, it follows that $d = a \times m + (b - a \times (m \div n)) \times n$, from which the result follows.

Now we can write a self-checking g.c.d. as follows:

```
exception GCD_ERROR
fun checked_gcd (m, n) =
    let
        val (d, a, b) = ggcd (m, n)
    in
        if m mod d = 0 andalso
           n mod d = 0 andalso d = a*m+b*n
        then
           d
        else
           raise GCD_ERROR
    end
```

This algorithm takes no more time (asymptotically) than the original, and, moreover, ensures that the result is correct. This illustrates the power of the interplay between mathematical reasoning methods such as induction and number theory and programming methods such as bulletproofing to achieve robust, reliable, and, what is more important, elegant programs.

25.3 Sample Code

Here is the code for this chapter.

Chapter 26

Structural Induction

The importance of induction and recursion are not limited to functions defined over the integers. Rather, the familiar concept of *mathematical induction* over the natural numbers is an instance of the more general notion of *structural induction* over values of an *inductively-defined type*. Rather than develop a general treatment of inductively-defined types, we will rely on a few examples to illustrate the point. Let's begin by considering the natural numbers as an inductively defined type.

26.1 Natural Numbers

The set of natural numbers, N, may be thought of as the smallest set containing 0 and closed under the formation of successors. In other words, n is an element of N iff either $n = 0$ or $n = m + 1$ for some $m$in N. Still another way of saying it is to define N by the following clauses:

1. 0 is an element of N.
2. If m is an element of N, then so is $m + 1$.
3. Nothing else is an element of N.

(The third clause is sometimes called the *extremal clause*; it ensures that we are talking about N and not just some superset of it.) All of these definitions are equivalent ways of saying the same thing.

Since N is inductively defined, we may prove properties of the natural numbers by *structural induction*, which in this case is just ordinary mathematical induction. Specifically, to prove that a property $P(n)$ holds of every n in N, it suffices to demonstrate the following facts:

1. Show that $P(0)$ holds.
2. Assuming that $P(m)$ holds, show that $P(m+1)$ holds.

The pattern of reasoning follows exactly the structure of the inductive definition — the base case is handled outright, and the inductive step is handled by assuming the property for the predecessor and show that it holds for the numbers.

The principal of structural induction also licenses the definition of functions by *structural recursion*. To define a function f with domain N, it suffices to proceed as follows:

1. Give the value of $f(0)$.
2. Give the value of $f(m+1)$ in terms of the value of $f(m)$.

Given this information, there is a unique function f with domain N satisfying these requirements. Specifically, we may show by induction on $n \geq 0$ that the value of f is uniquely determined on all values $m \leq n$. If $n = 0$, this is obvious since $f(0)$ is defined by the first clause. If $n = m+1$, then by induction the value of f is determined for all values $k \leq m$. But the value of f at n is determined as a function of $f(m)$, and hence is uniquely determined. Thus f is uniquely determined for all values of n in N, as was to be shown.

The natural numbers, viewed as an inductively-defined type, may be represented in ML using a `datatype` declaration, as follows:

```
datatype nat = Zero | Succ of nat
```

The constructors correspond one-for-one with the clauses of the inductive definition. The extremal clause is implicit in the `datatype` declaration since the given constructors are assumed to be *all* the ways of building values of type `nat`. This assumption forms the basis for exhaustiveness checking for clausal function definitions.

(You may object that this definition of the type `nat` amounts to a unary (base 1) representation of natural numbers, an unnatural and space-wasting

representation. This is indeed true; in practice the natural numbers are represented as non-negative machine integers to avoid excessive overhead. Note, however, that this representation places a fixed upper bound on the size of numbers, whereas the unary representation does not. Hybrid representations that enjoy the benefits of both are, of course, possible and occasionally used when enormous numbers are required.)

Functions defined by structural recursion are naturally represented by clausal function definitions, as in the following example:

```
fun double Zero = Zero
  | double (Succ n) = Succ (Succ (double n))
fun exp Zero = Succ(Zero)
  | exp (Succ n) = double (exp n)
```

The type checker ensures that we have covered all cases, but it does not ensure that the pattern of structural recursion is strictly followed — we may accidentally define $f(m+1)$ in terms of itself or some $f(k)$ where $k > m$, breaking the pattern. The reason this is admitted is that the ML compiler cannot always follow our reasoning: we may have a clever algorithm in mind that isn't easily expressed by a simple structural induction. To avoid restricting the programmer, the language assumes the best and allows any form of definition.

Using the principle of structure induction for the natural numbers, we may prove properties of functions defined over the naturals. For example, we may easily prove by structural induction over the type `nat` that for every $n \in N$, `exp` n evaluates to a positive number. (In previous chapters we carried out proofs of more interesting program properties.)

26.2 Lists

Generalizing a bit, we may think of the type `'a list` as inductively defined by the following clauses:

1. `nil` is a value of type `'a list`
2. If h is a value of type `'a`, and t is a value of type `'a list`, then $h::t$ is a value of type `'a list`.
3. Nothing else is a value of type `'a list`.

This definition licenses the following principle of structural induction over lists. To prove that a property P holds of all lists l, it suffices to proceed as follows:

1. Show that P holds for `nil`.
2. Show that P holds for $h::t$, assuming that P holds for t.

Similarly, we may define functions by structural recursion over lists as follows:

1. Define the function for `nil`.
2. Define the function for $h::t$ in terms of its value for t.

The clauses of the inductive definition of lists correspond to the following (built-in) datatype declaration in ML:

```
datatype 'a list = nil | :: of 'a * 'a list
```

(We are neglecting the fact that `::` is regarded as an infix operator.)

The principle of structural recursion may be applied to define the reverse function as follows:

```
fun reverse nil = nil
  | reverse (h::t) = reverse t @ [h]
```

There is one clause for each constructor, and the value of reverse for $h::t$ is defined in terms of its value for t. (We have ignored questions of time and space efficiency to avoid obscuring the induction principle underlying the definition of `reverse`.)

Using the principle of structural induction over lists, we may prove that `reverse` l evaluates to the reversal of l. First, we show that `reverse nil` yields `nil`, as indeed it does and ought to. Second, we assume that `reverse` t yields the reversal of t, and argue that `reverse (`$h::t$`)` yields the reversal of $h::t$, as indeed it does since it returns `reverse (`t `@ [`h`])`.

26.3 Trees

Generalizing even further, we can introduce *new* inductively-defined types such as *2-3 trees* in which interior nodes are either binary (have two children) or ternary (have three children). Here's a definition of 2-3 trees in ML:

```
datatype 'a twth_tree =
  Empty |
  Bin of 'a * 'a twth_tree * 'a twth_tree |
  Ter of 'a * 'a twth_tree * 'a twth_tree * 'a twth_tree
```

How might one define the "size" of a value of this type? Your first thought should be to write down a template like the following:

```
fun size Empty = ???
  | size (Bin (_, t1, t2)) = ???
  | size (Ter (_, t1, t2, t3)) = ???
```

We have one clause per constructor, and will fill in the ellided expressions to complete the definition. In many cases (including this one) the function is defined by structural recursion. Here's the complete definition:

```
fun size Empty = 0
  | size (Bin (_, t1, t2)) =
    1 + size t1 + size t2
  | size (Ter (_, t1, t2, t3)) =
    1 + size t1 + size t2 + size t3
```

Obviously this function computes the number of nodes in the tree, as you can readily verify by structural induction over the type `'a twth_tree`.

26.4 Generalizations and Limitations

Does this pattern apply to *every* datatype declaration? Yes and no. No matter what the form of the declaration it always makes sense to define a function over it by a clausal function definition with one clause per constructor. Such a definition is guaranteed to be exhaustive (cover all cases), and serves as a valuable guide to structuring your code. (It is especially valuable if you change the datatype declaration, because then the compiler will inform you of what clauses need to be added or removed from functions defined over that type in order to restore it to a sensible definition.) The slogan is:

> To define functions over a datatype, use a clausal definition with one clause per constructor

The catch is that not every datatype declaration supports a principle of structural induction because it is not always clear what constitutes the predecessor(s) of a constructed value. For example, the declaration

```
datatype D = Int of int | Fun of D->D
```

is problematic because a value of the form `Fun(`f`)` is not constructed directly from another value of type `D`, and hence it is not clear what to regard as its predecessor. In practice this sort of definition comes up only rarely; in most cases datatypes are naturally viewed as inductively defined.

26.5 Abstracting Induction

It is interesting to observe that the pattern of structural recursion may be directly codified in ML as a higher-order function. Specifically, we may associate with each inductively-defined type a higher-order function that takes as arguments values that determine the base case(s) and step case(s) of the definition, and defines a function by structural induction based on these arguments. An example will illustrate the point. The pattern of structural induction over the type `nat` may be codified by the following function:

```
fun nat_rec base step =
    let
        fun loop Zero = base
          | loop (Succ n) = step (loop n)
    in
        loop
    end
```

This function has the type `'a -> ('a -> 'a) -> nat -> 'a`.

Given the first two arguments, `nat_rec` yields a function of type `nat -> 'a` whose behavior is determined at the base case by the first argument and at the inductive step by the second. Here's an example of the use of `nat_rec` to define the exponential function:

```
val double =
    nat_rec Zero (fn result => Succ (Succ result))
val exp =
    nat_rec (Succ Zero) double
```

Note well the pattern! The arguments to `nat_rec` are

1. The value for `Zero`.
2. The value for `Succ` n defined in terms of its value for n.

Similarly, the pattern of list recursion may be captured by the following functional:

```
fun list_recursion base step =
    let
        fun loop nil = base
          | loop (h::t) = step (h, loop t)
    in
        loop
    end
```

The type of the function `list_recursion` is

```
'a -> ('b * 'a -> 'a) -> 'b list -> 'a
```

It may be instantiated to define the `reverse` function as follows:

```
val reverse = list_recursion nil (fn (h, t) => t @ [h])
```

Finally, the principle of structural recursion for values of type `'a twth_tree` is given as follows:

```
fun twth_rec base bin_step ter_step =
    let
        fun loop Empty = base
          | loop (Bin (v, t1, t2)) =
            bin_step (v, loop t1, loop t2)
          | loop (Ter (v, t1, t2, t3)) =
            ter_step (v, loop t1, loop t2, loop t3)
    in
        loop
    end
```

Notice that we have two inductive steps, one for each form of node. The type of `twth_rec` is

```
'a -> ('b * 'a * 'a -> 'a) -> ('b * 'a * 'a * 'a -> 'a) -> 'b twth_tree -> 'a
```

We may instantiate it to define the function size as follows:

```
val size =
  twth_rec 0
           (fn (_, s1, s2)) => 1+s1+s2)
           (fn (_, s1, s2, s3)) => 1+s1+s2+s3)
```

Summarizing, the principle of structural induction over a recursive datatype is naturally codified in ML using pattern matching and higher-order functions. Whenever you're programming with a datatype, you should use the techniques outlined in this chapter to structure your code.

26.6 Sample Code

Here is the code for this chapter.

Chapter 27

Proof-Directed Debugging

In this chapter we'll put specification and verification techniques to work in devising a regular expression matcher. The code is similar to that sketched in chapter 1, but we will use verification techiques to detect and correct a subtle error that may not be immediately apparent from inspecting or even testing the code. We call this process *proof-directed debugging*.

The first task is to devise a precise specification of the regular expression matcher. This is a difficult problem in itself. We then attempt to verify that the matching program developed in chapter 1 satisfies this specification. The proof attempt breaks down. Careful examination of the failure reveals a counterexample to the specification — the program does *not* satisfy it. We then consider how best to resolve the problem, not by *change of implementation*, but instead by *change of specification*.

27.1 Regular Expressions and Languages

Before we begin work on the matcher, let us first define the set of regular expressions and their meaning as a set of strings. The set of *regular expressions* is given by the following grammar:

$$r \quad ::= \quad \mathbf{0} \mid \mathbf{1} \mid \mathtt{a} \mid r_1\, r_2 \mid r_1 + r_2 \mid r^*$$

Here a ranges over a given *alphabet*, a set of primitive "letters" that may be used in a regular expression. A *string* is a finite sequence of letters of the alphabet. We write ε for the null string, the empty sequence of letters. We write $s_1\, s_2$ for the concatenation of the strings s_1 and s_2, the string s

consisting of the letters in s_1 followed by those in s_2. The *length* of a string is the number of letters in it. We do no distinguish between a character and the unit-length string consisting solely of that character. Thus we write $a\,s$ for the extension of s with the letter a at the front.

A *language* is a set of strings. Every regular expression r stands for a particular language $L(r)$, the *language* of r, which is defined by induction on the structure of r as follows:

$$\begin{array}{rcl} L(\mathbf{0}) & = & 0 \\ L(\mathbf{1}) & = & 1 \\ L(\mathtt{a}) & = & \{\,\mathtt{a}\,\} \\ L(r_1\,r_2) & = & L(r_1)\,L(r_2) \\ L(r_1 + r_2) & = & L(r_1) + L(r_2) \\ L(r^*) & = & L(r)^* \end{array}$$

This definition employs the following operations on languages:

$$\begin{array}{rcl} 0 & = & \emptyset \\ 1 & = & \{\,\varepsilon\,\} \\ L_1 + L_2 & = & L_1 \cup L_2 \\ L_1\,L_2 & = & \{\,s_1\,s_2 \mid s_1 \in L_1,\ s_2 \in L_2\,\} \\ L^{(0)} & = & 1 \\ L^{(i+1)} & = & L\,L^{(i)} \\ L^* & = & \bigcup_{i\geq 0} L^{(i)} \end{array}$$

An important fact about L^* is that it is the smallest language L' such that $1 + L\,L' \subseteq L'$. Spelled out, this means two things:

1. $1 + L\,L^* \subseteq L^*$, which is to say that
   (a) $\varepsilon \in L^*$, and
   (b) if $s \in L$ and $s' \in L^*$, then $s\,s' \in L^*$.
2. If $1 + L\,L' \subseteq L'$, then $L^* \subseteq L'$.

This means that L^* is the smallest language (with respect to language containment) that contains the null string and is closed under concatenation on the left by L.

Let's prove that this is the case. First, since $L^{(0)} = 1$, it follows immediately that $\varepsilon \in L^*$. Second, if $l \in L$ and $l' \in L^*$, then $l' \in L^{(i)}$ for some $i \geq 0$,

and hence $l\,l' \in L^{(i+1)}$ by definition of concatentation of languages. This completes the first step. Now suppose that L' is such that $1 + L\,L' \subseteq L'$. We are to show that $L^* \subseteq L'$. We show by induction on $i \geq 0$ that $L^{(i)} \subseteq L'$, from which the result follows immediately. If $i = 0$, then it suffices to show that $\varepsilon \in L'$. But this follows from the assumption that $1 + L\,L' \subseteq L'$, which implies that $1 \subseteq L'$. To show that $L^{(i+1)} \subseteq L'$, we observe that, by definition, $L^{(i+1)} = L\,L^{(i)}$. By induction $L^{(i)} \subseteq L'$, and hence $L\,L^{(i)} \subseteq L'$, since $L\,L' \subseteq L'$ by assumption.

Having proved that L^* is the smallest language L' such that $1 + L\,L' \subseteq L'$, it is not hard to prove that L^* satisfies the recurrence $L^* = 1 + L\,L^*$. We just proved the right to left containment. For the converse, it suffices to observe that $1 + L\,(1 + L\,L^*) \subseteq 1 + L\,L^*$, for then the result follows by minimality the result. This is easily established by a simple case analysis.

Exercise 1
Give a full proof of the fact that $L^* = 1 + L\,L^*$.

Finally, a word about implementation. We will assume in what follows that the alphabet is given by the type `char` of characters, that strings are elements of the type `string`, and that regular expressions are defined by the following `datatype` declaration:

```
datatype regexp =
  Zero | One | Char of char |
  Plus of regexp * regexp |
  Times of regexp * regexp |
  Star of regexp
```

We will also work with lists of characters, values of type `char list`, using ML notation for primitive operations on lists such as concatenation and extension. Occasionally we will abuse notation and not distinguish (in the informal discussion) between a string and a list of characters. In particular we will speak of a character list as being a member of a language, when in fact we mean that the corresponding string is a member of that language.

27.2 Specifying the Matcher

Let us begin by devising a specification for the regular expression matcher. As a first cut we write down a type specification. We seek to define a func-

tion `match` of type `regexp -> string -> bool` that determines whether or not a given string matches a given regular expression. More precisely, we wish to satisfy the following specification:

> For every regular expression r and every string s, `match` r s terminates, and evaluates to `true` iff $s \in L(r)$.

We saw in chapter 1 that a natural way to define the procedure `match` is to use a technique called *continuation passing*. We defined an auxiliary function `match_is` with the type

```
regexp -> char list -> (char list -> bool) -> bool
```

that takes a regular expression, a list of characters (essentially a string, but in a form suitable for incremental processing), and a continuation, and yields a boolean. The idea is that `match_is` takes a regular expression r, a character list cs, and a continuation k, and determines whether or not some initial segment of cs matches r, passing the remaining characters cs' to k in the case that there is such an initial segment, and yields `false` otherwise. Put more precisely,

> For every regular expression r, character list cs, and continuation k, if $cs = cs'@cs''$ with $cs' \in L(r)$ and $k\ cs''$ evaluates to `true`, then `match_is` r cs k evaluates `true`; otherwise, `match_is` r cs k evaluates to `false`.

Unfortunately, this specification is too strong to ever be satisfied by any program! Can you see why? The difficulty is that if k is not guaranteed to terminate for all inputs, then there is no way that `match_is` can behave as required. For example, if there is no input on which k terminates, the specification requires that match_is return `false`. It should be intuitively clear that we can never implement such a function. Instead, we must restrict attention to *total* continuations, those that always terminate with `true` or `false` on any input. This leads to the following revised specification:

> For every regular expression r, character list cs, and total continuation k, if $cs = cs'\,cs''$ with $cs' \in L(r)$ and $k\ cs''$ evaluates to `true`, then `match_is` r cs k evaluates to `true`; otherwise, `match_is` r cs k evaluates to `false`.

Observe that this specification makes use of an implicit existential quantification. Written out in full, we might say "For all ..., if there exists cs' and cs'' such that $cs = cs'\,cs''$ with ..., then ...". This observation makes clear that we must *search* for a suitable splitting of cs into two parts such that the first part is in $L(r)$ and the second is accepted by k. There may, in general, be many ways to partition the input to as to satisfy both of these requirements; we need only find one such way. Note, however, that if $cs = cs'@cs''$ with $cs' \in L(r)$ but $k\ cs''$ yielding `false`, we must reject this partitioning and search for another. In other words we cannot simply accept *any* partitioning whose initial segment matches r, but rather only those that also induce k to accept its corresponding final segment. We may return `false` only if there is *no* such splitting, not merely if a particular splitting fails to work.

Suppose for the moment that `match_is` satisfies this specification. Does it follow that `match` satisfies the original specification? Recall that the function `match` is defined as follows:

```
fun match r s =
    match_is r
             (String.explode s)
             (fn nil => true | false)
```

Notice that the initial continuation is indeed total, and that it yields `true` (accepts) iff it is applied to the null string. Therefore `match` satisfies the following property obtained from the specification of `mathc_is` by plugging in the initial continuation:

> For every regular expression r and string s, if $s \in L(r)$, then `match` r s evaluates to `true`, and otherwise `match` r s evaluates to `false`.

This is precisely the property that we desire for `match`. Thus `match` is correct (satisfies its specification) if `match_is` is correct.

So far so good. But does `match_is` satisfy its specification? If so, we are done. How might we check this? Recall the definition of `match_is` given in the overview:

```
fun match_is Zero _ k = false
  | match_is One cs k = k cs
  | match_is (Char c) nil k = false
```

```
| match_is (Char c) (d::cs) k =
  if c=d then k cs else false
| match_is (Times (r1, r2)) cs k =
  match_is r1 cs (fn cs' => match_is r2 cs' k)
| match_is (Plus (r1, r2)) cs k =
  match_is r1 cs k  orelse  match_is r2 cs k
| match_is (Star r) cs k =
  k cs  orelse
  match_is r cs (fn cs' => match_is (Star r) cs' k)
```

Since `match_is` is defined by a recursive analysis of the regular expression r, it is natural to proceed by induction on the structure of r. That is, we treat the specification as a conjecture about `match_is`, and attempt to prove it by structural induction on r.

We first consider the three base cases. Suppose that r is $\mathbf{0}$. Then no string is in $L(r)$, so `match_is` must return `false`, which indeed it does. Suppose that r is $\mathbf{1}$. Since the null string is an initial segment of every string, and the null string is in $L(\mathbf{1})$, we must yield `true` iff $k\ cs$ yields `true`, and `false` otherwise. This is precisely how `match_is` is defined. Suppose that r is `a`. Then to succeed cs must have the form `a`cs' with $k\ cs'$ evaluating to `true`; otherwise we must fail. The code for `match_is` checks that cs has the required form and, if so, passes cs' to k to determine the outcome, and otherwise yields `false`. Thus `match_is` behaves correctly for each of the three base cases.

We now consider the three inductive steps. For $r = r_1 + r_2$, we observe that some initial segment of cs matches r and causes k to accept the corresponding final segment of cs iff either some initial segment matches r_1 and drives k to accept the rest or some initial segment matches r_2 and drives k to accept the rest. By induction `match_is` works as specified for r_1 and r_2, which is sufficient to justify the correctness of `match_is` for $r = r_1 + r_2$.

For $r = r_1\, r_2$, the proof is slightly more complicated. By induction `match_is` behaves according to the specification if it is applied to either r_1 or r_2, provided that the continuation argument is total. Note that the continuation k' given by `fn cs' => match_is r2 cs' k` is total, since by induction the inner recursive call to `match_is` always terminates. Suppose that there exists a partitioning $cs = cs'@cs''$ with $cs' \in L(r)$and $k\ cs''$ evaluating to `true`. Then $cs' = cs'_1\, cs'_2$ with $cs'_1 \in L(r_1)$ and $cs'_2 \in L(r_2)$, by definition of $L(r_1\, r_2)$. Consequently, `match_is` r_2 $(cs'_2\, cs'')$ k evaluates to

`true`, and hence `match_is` r_1 $cs'_1\, cs'_2\, cs''$ k' evaluates to `true`, as required. If, however, no such partitioning exists, then one of three situations occurs:

1. either no initial segment of cs matches r_1, in which case the outer recursive call yields `false`, as required, or

2. for *every* initial segment matching r_1, no initial segment of the corresponding final segment matches r_2, in which case the inner recursive call yields `false` on every call, and hence the outer call yields `false`, as required, or

3. every pair of successive initial segments of cs matching r_1 and r_2 successively results in k evaluating to `false`, in which case the inner recursive call always yields `false`, and hence the continuation k' always yields `false`, and hence the outer recursive call yields `false`, as required.

Be sure you understand the reasoning involved here, it is quite tricky to get right!

We seem to be on track, with one more case to consider, $r = r_1{}^*$. This case would appear to be a combination of the preceding two cases for alternation and concatenation, with a similar argument sufficing to establish correctness. But there is a snag: the second recursive call to `match_is` leaves the regular expression unchanged! Consequently we cannot apply the inductive hypothesis to establish that it behaves correctly in this case, and the obvious proof attempt breaks down.

What to do? A moment's thought suggests that we proceed by an inner induction on the length of the string, based on the idea that if some initial segment of cs matches $L(r)$, then either that initial segment is the null string (base case), or $cs = cs'@cs''$ with $cs' \in L(r_1)$ and $cs'' \in L(r)$ (induction step). We then handle the base case directly, and handle the inductive case by assuming that `match_is` behaves correctly for cs'' and showing that it behaves correctly for cs. But there is a flaw in this argument — the string cs'' need not be shorter than cs in the case that cs' is the null string! In that case the inductive hypothesis does not apply, and we are once again unable to complete the proof.

This time we can use the failure of the proof to obtain a counterexample to the specification! For if $r = \mathbf{1}^*$, for example, then `match_is` r cs k does not terminate! In general if $r = r_1{}^*$ with $\varepsilon \in L(r_1)$, then `match_is` r

cs k fails to terminate. In other words, `match_is` does *not* satisfy the specification we have given for it. Our conjecture is false!

Our failure to establish that `match_is` satisfies its specification lead to a counterexample that refuted our conjecture and uncovered a genuine bug in the program — the matcher may not terminate for some inputs. What to do? One approach is to explicitly check for looping behavior during matching by ensuring that each recursive calls matches *some* non-empty initial segment of the string. This will work, but at the expense of cluttering the code and imposing additional run-time overhead. You should write out a version of the matcher that works this way, and check that it indeed satisfies the specification we've given above.

An alternative is to observe that the proof goes through under the additional assumption that no iterated regular expression matches the null string. Call a regular expression *r standard* iff whenever r'^* occurs within r, the null string is not an element of $L(r')$. It is easy to check that the proof given above goes through under the assumption that the regular expression r is standard.

This says that the matcher works correctly for standard regular expressions. But what about the non-standard ones? The key observation is that *every regular expression is equivalent to one in standard form*. By "equivalent" we mean "accepting the same language". For example, the regular expressions $r + \mathbf{0}$ and r are easily seen to be equivalent. Using this observation we may avoid the need to consider non-standard regular expressions. Instead we can pre-process the regular expression to put it into standard form, then call the matcher on the standardized regular expression.

The required pre-processing is based on the following definitions. We will associate with each regular expression r two standard regular expressions $\delta(r)$ and r^- with the following properties:

1. $L(\delta(r)) = 1$ iff $\varepsilon \in L(r)$ and $L(\delta(r)) = 0$ otherwise.
2. $L(r^-) = L(r) \setminus 1$.

With these equations in mind, we see that every regular expression r may be written in the form $\delta(r) + r^-$, which is in standard form.

The function δ mapping regular expressions to regular expressions is defined by induction on regular expressions by the following equations:

$$\begin{array}{rcl} \delta(\mathbf{0}) & = & \mathbf{0} \\ \delta(\mathbf{1}) & = & \mathbf{1} \\ \delta(\mathtt{a}) & = & \mathbf{0} \\ \delta(r_1 + r_2) & = & \delta(r_1) \oplus \delta(r_2) \\ \delta(r_1\, r_2) & = & \delta(r_1) \otimes \delta(r_2) \\ \delta(r^*) & = & \mathbf{1} \end{array}$$

Here we define $\mathbf{0} \oplus \mathbf{1} = \mathbf{1} \oplus \mathbf{0} = \mathbf{1} \oplus \mathbf{1} = \mathbf{1}$ and $\mathbf{0} \oplus \mathbf{0} = \mathbf{0}$ and $\mathbf{0} \otimes \mathbf{1} = \mathbf{1} \otimes \mathbf{0} = \mathbf{0} \otimes \mathbf{0} = \mathbf{0}$ and $\mathbf{1} \otimes \mathbf{1} = \mathbf{1}$.

Exercise 2
Show that $L(\delta(r)) = 1$ *iff* $\varepsilon \in L(r)$.

The definition of r^- is given by induction on the structure of r by the following equations:

$$\begin{array}{rcl} \mathbf{0}^- & = & \mathbf{0} \\ \mathbf{1}^- & = & \mathbf{0} \\ \mathtt{a}^- & = & \mathbf{0} \\ (r_1 + r_2)^- & = & r_1^- + r_2^- \\ (r_1\, r_2)^- & = & \delta(r_1)\, r_2^- + r_1\, \delta(r_2) + r_1^-\, r_2^- \\ (r^*)^- & = & \delta(r) + r^{-*} \end{array}$$

The only tricky case is the one for concatenation, which must take account of the possibility that r_1 or r_2 accepts the null string.

Exercise 3
Show that $L(r^-) = L(r) \setminus 1$.

27.3 Sample Code

Here is the code for this chapter.

Chapter 28

Persistent and Ephemeral Data Structures

This chapter is concerned with *persistent* and *ephemeral* abstract types. The distinction is best explained in terms of the *logical future* of a value. Whenever a value of an abstract type is created it may be subsequently acted upon by the operations of the type (and, since the type is abstract, by no other operations). Each of these operations may yield (other) values of that abstract type, which may themselves be handed off to further operations of the type. Ultimately a value of some other type, say a string or an integer, is obtained as an observable outcome of the succession of operations on the abstract value. The sequence of operations performed on a value of an abstract type constitutes a logical future of that type — a computation that starts with that value and ends with a value of some observable type. We say that a type is *ephemeral* iff every value of that type has at most one logical future, which is to say that it is handed off from one operation of the type to another until an observable value is obtained from it. This is the normal case in familiar imperative programming languages because in such languages the operations of an abstract type destructively modify the value upon which they operate; its original state is irretrievably lost by the performance of an operation. It is therefore inherent in the imperative programming model that a value have at most one logical future. In contrast, values of an abstract type in functional languages such as ML may have many different logical futures, precisely because the operations do not "destroy" the value upon which they operate, but rather create fresh values of that type to yield as results. Such values are said to be *persistent*

because they persist after application of an operation of the type, and in fact may serve as arguments to further operations of that type.

Some examples will help to clarify the distinction. The primitive list types of ML are persistent because the performance of an operation such as cons'ing, appending, or reversing a list does not destroy the original list. This leads naturally to the idea of multiple logical futures for a given value, as illustrated by the following code sequence:

```
(* original list *)
val l = [1,2,3]
val m1 = hd l
(* first future of l *)
val n1 = rev (tl m1)
(* second future of l *)
val m2 = l @ [4,5,6]
```

Notice that the original list value, `[1,2,3]`, has two distinct logical futures, one in which we remove its head, then reverse the tail, and the other in which we append the list `[4,5,6]` to it. The ability to easily handle multiple logical futures for a data structure is a tremendous source of flexibility and expressive power, alleviating the need to perform tedious bookkeeping to manage "versions" or "copies" of a data structure to be passed to different operations.

The prototypical ephemeral data structure in ML is the reference cell. Performing an assignment operation on a reference cell changes it irrevocably; the original contents of the cell are lost, even if we keep a handle on it.

```
val r = ref 0
(* original cell *)
val s = r
val _ = (s := 1)
val x = !r
(* 1! *)
```

Notice that the contents of (the cell bound to) `r` changes as a result of performing an assignment to the underlying cell. There is only one future for this cell; a reference to its original binding does not yield its original contents.

More elaborate forms of ephemeral data structures are certainly possible. For example, the following declaration defines a type of lists whose tails are mutable. It is therefore a singly-linked list, one whose predecessor relation may be changed dynamically by assignment:

```
datatype 'a mutable_list =
  Nil |
  Cons of 'a * 'a mutable_list ref
```

Values of this type are ephemeral in the sense that some operations on values of this type are destructive, and hence are irreversible (so to speak!). For example, here's an implementation of a destructive reversal of a mutable list. Given a mutable list l, this function reverses the links in the cell so that the elements occur in reverse order of their occurrence in l.

```
local
   fun ipr (Nil, a) = a
     | ipr (this as (Cons (_, r as ref next)), a) =
       ipr (next, (r := a; this))
in
   (* destructively reverse a list *)
   fun inplace_reverse l = ipr (l, Nil)
end
```

As you can see, the code is quite tricky to understand! The idea is the same as the iterative reverse function for pure lists, except that we re-use the nodes of the original list, rather than generate new ones, when moving elements onto the accumulator argument.

The distinction between ephemeral and persistent data structures is essentially the distinction between functional (effect-free) and imperative (effect-ful) programming — functional data structures are persistent; imperative data structures are ephemeral. However, this characterization is oversimplified in two respects. First, it is possible to implement a persistent data structure that exploits mutable storage. Such a use of mutation is an example of what is called a *benign effect* because for all practical purposes the data structure is "purely functional" (*i.e.*, persistent), but is in fact implemented using mutable storage. As we will see later the exploitation of benign effects is crucial for building efficient implementations of persistent data structures. Second, it is possible for a persistent data type

to be used in such a way that persistence is not exploited — rather, every value of the type has at most one future in the program. Such a type is said to be *single-threaded*, reflecting the linear, as opposed to branching, structure of the future uses of values of that type. The significance of a single-threaded type is that it may as well have been implemented as an ephemeral data structure (*e.g.*, by having observable effects on values) without changing the behavior of the program.

28.1 Persistent Queues

Here is a signature of persistent queues:

```
signature QUEUE = sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val insert : 'a * 'a queue -> 'a queue
  val remove : 'a queue -> 'a * 'a queue
end
```

This signature describes a structure providing a representation type for queues, together with operations to create an empty queue, insert an element onto the back of the queue, and to remove an element from the front of the queue. It also provides an exception that is raised in response to an attempt to remove an element from the empty queue. Notice that removing an element from a queue yields both the element at the front of the queue, and the queue resulting from removing that element. This is a direct reflection of the persistence of queues implemented by this signature; the original queue remains available as an argument to further queue operations.

By a *sequence* of queue operations we shall mean a succession of uses of `empty`, `insert`, and `remove` operations in such a way that the queue argument of one operation is obtained as a result of the immediately preceding queue operation. Thus a sequence of queue operations represents a single-threaded time-line in the life of a queue value. Here is an example of a sequence of queue operations:

```
val q0 : int queue = empty
```

```
val q1 = insert (1, q0)
val q2 = insert (2, q1)
val (h1, q3) = remove q2     (* h1 = 1, q3 = q1 *)
val (h2, q4) = remove q3     (* h2 = 2, q4 = q0 *)
```

By contrast the following operations do not form a single thread, but rather a branching development of the queue's lifetime:

```
val q0 : int queue = empty
val q1 = insert (1, q0)
val q2 = insert (2, q0)     (* NB: q0, not q1! *)
val (h1, q3) = remove q1    (* h1 = 1, q3 = q0 *)
val (h2, q4) = remove q3    (* raise Empty *)
val (h2, q4) = remove q2    (* h2 = 2,, q4 = q0 *)
```

In the remainder of this section we will be concerned with single-threaded sequences of queue operations.

How might we implement the signature `QUEUE`? The most obvious approach is to represent the queue as a list with, say, the head element of the list representing the "back" (most recently enqueued element) of the queue. With this representation enqueueing is a constant-time operation, but dequeuing requires time proportional to the number of elements in the queue. Thus in the worst case a sequence of n enqueue and dequeue operations will take time $O(n^2)$, which is clearly excessive. We can make dequeue simpler, at the expense of enqueue, by regarding the head of the list as the "front" of the queue, but the time bound for n operations remains the same in the worst case.

Can we do better? A well-known "trick" achieves an $O(n)$ worst-case performance for any sequence of n operations, which means that each operation takes $O(1)$ steps if we *amortize* the cost over the entire sequence. Notice that this is a *worst-case* bound for the *sequence*, yielding an *amortized* bound for *each operation* of the sequence. This means that some operations may be relatively expensive, but, in compensation, many operations will be cheap.

How is this achieved? By combining the two naive solutions sketched above. The idea is to represent the queue by *two*lists, one for the back "half" consisting of recently inserted elements in the order of arrival, and one for the front "half" consisting of soon-to-be-removed elements in *reverse* order of arrival (*i.e.*, in order of removal). We put "half" in quotes

because we will not, in general, maintain an even split of elements between the front and the back lists. Rather, we will arrange things so that the following representation invariants holds true:

1. The elements of the queue listed in order of removal are the elements of the front followed by the elements of the back in reverse order.

2. The front is empty only if the back is empty.

These invariants are maintained by using a "smart constructor" that creates a queue from two lists representing the back and front parts of the queue. This constructor ensures that the representation invariant holds by ensuring that condition (2) is always true of the resulting queue. The constructor proceeds by a case analysis on the back and fron parts of the queue. If the front list is non-empty, or both the front and back are empty, the resulting queue consists of the back and front parts as given. If the front is empty and the back is non-empty, the queue constructor yields the queue consisting of an empty back part and a front part equal to the reversal of the given back part. Observe that this is sufficient to ensure that the representation invariant holds of the resulting queue in all cases. Observe also that the smart constructor either runs in constant time, or in time proportional to the length of the back part, according to whether the front part is empty or not.

Insertion of an element into a queue is achieved by cons'ing the element onto the back of the queue, then calling the queue constructor to ensure that the result is in conformance with the representation invariant. Thus an insert can either take constant time, or time proportional to the size of the back of the queue, depending on whether the front part is empty. Removal of an element from a queue requires a case analysis. If the front is empty, then by condition (2) the queue is empty, so we raise an exception. If the front is non-empty, we simply return the head element together with the queue created from the original back part and the front part with the head element removed. Here again the time required is either constant or proportional to the size of the back of the queue, according to whether the front part becomes empty after the removal. Notice that if an insertion or removal requires a reversal of k elements, then the next k operations are constant-time. This is the fundamental insight as to why we achieve $O(n)$ time complexity over any sequence of n operations. (We will give a more rigorous analysis shortly.)

Here's the implementation of this idea in ML:

```
structure Queue :> QUEUE = struct
  type 'a queue = 'a list * 'a list
  fun make_queue (q as (nil, nil)) = q
    | make_queue (bs, nil) = (nil, rev bs)
    | make_queue (q as (bs, fs)) = q
  val empty = make_queue (nil, nil)
  fun insert (x, (back,front)) =
     make_queue (x::back, front)
  exception Empty
  fun remove (_, nil) = raise Empty
    | remove (bs, f::fs) = (f, make_queue (bs, fs))
end
```

Notice that we call the "smart constructor" `make_queue` whenever we wish to return a queue to ensure that the representation invariant holds. Consequently, some queue operations are more expensive than others, according to whether or not the queue needs to be reorganized to satisfy the representation invariant. However, each such reorganization makes a corresponding number of subsequent queue operations "cheap" (constant-time), so the overall effort required evens out in the end to constant-time per operation. More precisely, the running time of a sequence of n queue operations is now $O(n)$, rather than $O(n^2)$, as it was in the naive implementation. Consequently, each operation takes $O(1)$ (constant) time "on average," *i.e.*, when the total effort is evenly apportioned among each of the operations in the sequence. Note that this is a *worst-case* time bound for each operation, *amortized over the entire sequence*, not an *average-case* time bound based on assumptions about the distribution of the operations.

28.2 Amortized Analysis

How can we prove this claim? First we given an informal argument, then we tighten it up with a more rigorous analysis. We are to account for the total work performed by a sequence of n operations by showing that any sequence of $n$operations can be executed in cn steps for some constant c. Dividing by n, we obtain the result that each operations takes c steps when amortized over the entire sequence. The key is to observe first that

the work required to execute a sequence of queue operations may be apportioned to the elements themselves, then that only a constant amount of work is expended on each element. The "life" of a queue element may be divided into three stages: it's arrival in the queue, it's transit time in the queue, and it's departure from the queue. In the worst case each element passes through each of these stages (but may "die young", never participating in the second or third stage). Arrival requires constant time to add the element to the back of the queue. Transit consists of being moved from the back to the front by a reversal, which takes constant time per element on the back. Departure takes constant time to pattern match and extract the element. Thus at worst we require three steps per element to account for the entire effort expended to perform a sequence of queue operations. This is in fact a conservative upper bound in the sense that we may need less than $3n$ steps for the sequence, but asymptotically the bound is optimal — we cannot do better than constant time per operation! (You might reasonably wonder whether there is a worst-case, non-amortized constant-time implementation of persistent queues. The answer is "yes", but the code is far more complicated than the simple implementation we are sketching here.)

This argument can be made rigorous as follows. The general idea is to introduce the notion of a *charge scheme* that provides an upper bound on the actual cost of executing a sequence of operations. An upper bound on the charge will then provide an upper bound on the actual cost. Let $T(n)$ be the cumulative time required (in the worst case) to execute a sequence of n queue operations. We will introduce a *charge function*, $C(n)$, representing the *cumulative charge* for executing a sequence of n operations and show that $T(n) \leq C(n) = O(n)$. It is convenient to express this in terms of a function $R(n) = C(n) - T(n)$ representing the cumulative *residual*, or *overcharge*, which is the amount that the charge for n operations exceeds the actual cost of executing them. We will arrange things so that $R(n) \geq 0$ and that $C(n) = O(n)$, from which the result follows immediately.

Down to specifics. By charging 2 for each insert operation and 1 for each remove, it follows that $C(n) \leq 2n$ for any sequence of n inserts and removes. Thus $C(n) = O(n)$. After any sequence of $n \geq 0$ operations have been performed, the queue contains $0 \leq b \leq n$ elements on the back "half" and $0 \leq f \leq n$ elements on the front "half". We claim that for every $n \geq 0$, $R(n) = b$. We prove this by induction on $n \geq 0$. The condition clearly holds after performing 0 operations, since $T(0) = 0$, $C(0) = 0$,

and hence $R(0) = C(0) - T(0) = 0$. Consider the $n+1$st operation. If it is an insert, and $f > 0$, $T(n+1) = T(n) + 1$, $C(n+1) = C(n) + 2$, and hence $R(n+1) = R(n) + 1 = b + 1$. This is correct because an insert operation adds one element to the back of the queue. If, on the other hand, $f = 0$, then $T(n+1) = T(n) + b + 2$ (charging one for the cons and one for creating the new pair of lists), $C(n+1) = C(n) + 2$, so $R(n+1) = R(n) + 2 - b - 2 = b + 2 - b - 2 = 0$. This is correct because the back is now empty; we have used the residual overcharge to pay for the cost of the reversal. If the $n+1$st operation is a remove, and $f > 0$, then $T(n+1) = T(n) + 1$ and $C(n+1) = C(n) + 1$ and hence $R(n+1) = R(n) = b$. This is correct because the remove doesn't disturb the back in this case. Finally, if we are performing a remove with $f = 0$, then $T(n+1) = T(n) + b + 1$, $C(n+1) = C(n) + 1$, and hence $R(n+1) = R(n) - b = b - b = 0$. Here again we use of the residual overcharge to pay for the reversal of the back to the front. The result follows immediately since $R(n) = b \geq 0$, and hence $C(n) \geq T(n)$.

It is instructive to examine where this solution breaks down in the multi-threaded case (*i.e.*, where persistence is fully exploited). Suppose that we perform a sequence of n insert operations on the empty queue, resulting in a queue with n elements on the back and none on the front. Call this queue q. Let us suppose that we have n independent "futures" for q, each of which removes an element from it, for a total of $2n$ operations. How much time do these $2n$ operations take? Since each independent future must reverse all n elements onto the front of the queue before performing the removal, the entire collection of $2n$ operations takes $n + n^2$ steps, or $O(n)$ steps per operation, breaking the amortized constant-time bound just derived for a single-threaded sequence of queue operations. Can we recover a constant-time amortized cost in the persistent case? We can, provided that we *share* the cost of the reversal among *all* futures of q — as soon as one performs the reversal, they all enjoy the benefit of its having been done. This may be achieved by using a benign side effect to *cache* the result of the reversal in a reference cell that is shared among all uses of the queue. We will return to this once we introduce *memoization* and *lazy evaluation*.

28.3 Sample Code

Here is the code for this chapter.

Chapter 29

Options, Exceptions, and Continuations

In this chapter we discuss the close relationships between option types, exceptions, and continuations. They each provide the means for handling failure to produce a value in a computation. Option types provide the means of explicitly indicating in the type of a function the possibility that it may fail to yield a "normal" result. The result type of the function forces the caller to dispatch explicitly on whether or not it returned a normal value. Exceptions provide the means of implicitly signalling failure to return a normal result value, without sacrificing the requirement that an application of such a function cannot ignore failure to yield a value. Continuations provide another means of handling failure by providing a function to invoke in the case that normal return is impossible.

29.1 The n-Queens Problem

We will explore the trade-offs between these three approaches by considering three different implementations of the n-queens problem: find a way to place n queens on an $n \times n$ chessboard in such a way that no two queens attack one another. The general strategy is to place queens in successive columns in such a way that it is not attacked by a previously placed queen. Unfortunately it's not possible to do this in one pass; we may find that we can safely place $k < n$ queens on the board, only to discover that there is no way to place the next one. To find a solution we must reconsider earlier

decisions, and work forward from there. If all possible reconsiderations of all previous decisions all lead to failure, then the problem is unsolvable. For example, there is no safe placement of three queens on a 3x3 chessboard. This trial-and-error approach to solving the n-queens problem is called *backtracking search*.

A solution to the n-queens problem consists of an $n \times n$ chessboard with n queens safely placed on it. The following signature defines a chessboard abstraction:

```
signature BOARD =
sig
  type board
  val new : int -> board
  val complete : board -> bool
  val place : board * int -> board
  val safe : board * int -> bool
  val size : board -> int
  val positions : board -> (int * int) list
end
```

The operation `new` creates a new board of a given dimension $n \geq 0$. The operation `complete` checks whether the board contains a complete safe placement of n queens. The function `safe` checks whether it is safe to place a queen at row i in the next free column of a board B. The operation `place` puts a queen at row i in the next available column of the board. The function `size` returns the size of a board, and the function `positions` returns the coordinates of the queens on the board.

The board abstraction may be implemented as follows:

```
structure Board :> BOARD =
struct
  (* rep: size, next free column, number placed, placements
     inv: size>=0, 1<=next free<=size,
     length(placements) = number placed
  *)
  type board = int * int * int * (int * int) list
  fun new n = (n, 1, 0, nil)
  fun size (n, _, _, _) = n
```

```
  fun complete (n, _, k, _) = (k=n)
  fun positions (_, _, _, qs) = qs
  fun place ((n, i, k, qs),j) =
     (n, i+1, k+1, (i,j)::qs)
  fun threatens ((i,j), (i',j')) =
      i=i' orelse j=j' orelse
      i+j = i'+j' orelse
      i-j = i'-j'
  fun conflicts (q, nil) =
      false
    | conflicts (q, q'::qs) =
      threatens (q, q') orelse conflicts (q, qs)
  fun safe ((_, i, _, qs), j) =
      not (conflicts ((i,j), qs))
end
```

The representation type contains "redundant" information in order to make the individual operations more efficient. The representation invariant ensures that the components of the representation are properly related to one another (*e.g.*, the claimed number of placements is indeed the length of the list of placed queens, and so on.)

Our goal is to define a function

```
val queens : int -> Board.board option
```

such that if $n \geq 0$, then `queens` n evaluates either to `NONE` if there is no safe placement of n queens on an $n \times n$ board, or to `SOME` B otherwise, with B a complete board containing a safe placement of n queens. We will consider three different solutions, one using option types, one using exceptions, and one using a failure continuation.

29.2 Solution Using Options

Here's a solution based on option types:

```
  (* addqueen bd yields SOME bd' with bd' a
     complete safe placement extending bd,
```

```
    if one exists, and yields NONE otherwise
*)
fun addqueen bd =
    let
        fun try j =
            if j > Board.size bd then
               NONE
            else if Board.safe (bd, j) then
               case addqueen (Board.place (bd, j))
                 of NONE => try (j+1)
                  | r as (SOME bd') => r
            else
               try (j+1)
    in
       if Board.complete bd then
          SOME bd
       else
          try 1
    end
fun queens n = addqueen (Board.new n)
```

The characteristic feature of this solution is that we must explicitly check the result of each recursive call to `addqueen` to determine whether a safe placement is possible from that position. If so, we simply return it; if not, we must reconsider the placement of a queen in row j of the next available column. If no placement is possible in the current column, the function yields `NONE`, which forces reconsideration of the placement of a queen in the preceding row. Eventually we either find a safe placement, or yield `NONE` indicating that no solution is possible.

29.3 Solution Using Exceptions

The explicit check on the result of each recursive call can be replaced by the use of exceptions. Rather than have `addqueen` return a value of type `Board.board option`, we instead have it return a value of type `Board.board`, if possible, and otherwise raise an exception indicating failure. The case analysis on the result is replaced by a use of an exception handler. Here's the code:

```
exception Fail
(* addqueen bd yields bd', where bd' is a complete safe
   placement extending bd, if one exists, and raises Fail otherwise
*)
fun addqueen bd =
    let
        fun try j =
            if j > Board.size bd then
               raise Fail
            else if Board.safe (bd, j) then
               addqueen (Board.place (bd, j))
               handle Fail => try (j+1)
            else
               try (j+1)
    in
       if Board.complete bd then
          bd
       else
          try 1
   end
fun queens n =
    SOME (addqueen (Board.new n))
    handle Fail => NONE
```

The main difference between this solution and the previous one is that both calls to `addqueen` must handle the possibility that it raises the exception `Fail`. In the outermost call this corresponds to a complete failure to find a safe placement, which means that `queens` must return `NONE`. If a safe placement is indeed found, it is wrapped with the constructor `SOME` to indicate success. In the recursive call within `try`, an exception handler is required to handle the possibility of there being no safe placement starting in the current position. This check corresponds directly to the case analysis required in the solution based on option types.

What are the trade-offs between the two solutions?

1. The solution based on option types makes explicit in the type of the function `addqueen` the possibility of failure. This forces the programmer to explicitly test for failure using a case analysis on the result of the call. The type checker will ensure that one cannot use a

`Board.board option` where a `Board.board` is expected. The solution based on exceptions does not explicitly indicate failure in its type. However, the programmer is nevertheless forced to handle the failure, for otherwise an uncaught exception error would be raised at run-time, rather than compile-time.

2. The solution based on option types requires an explicit case analysis on the result of each recursive call. If "most" results are successful, the check is redundant and therefore excessively costly. The solution based on exceptions is free of this overhead: it is biased towards the "normal" case of returning a board, rather than the "failure" case of not returning a board at all. The implementation of exceptions ensures that the use of a handler is more efficient than an explicit case analysis in the case that failure is rare compared to success.

For the n-queens problem it is not clear which solution is preferable. In general, if efficiency is paramount, we tend to prefer exceptions if failure is a rarity, and to prefer options if failure is relatively common. If, on the other hand, static checking is paramount, then it is advantageous to use options since the type checker will enforce the requirement that the programmer check for failure, rather than having the error arise only at run-time.

29.4 Solution Using Continuations

We turn now to a third solution based on continuation-passing. The idea is quite simple: an exception handler is essentially a function that we invoke when we reach a blind alley. Ordinarily we achieve this invocation by raising an exception and relying on the caller to catch it and pass control to the handler. But we can, if we wish, pass the handler around as an additional argument, the *failure continuation* of the computation. Here's how it's done in the case of the n-queens problem:

```
(* addqueen bd yields bd', where bd' is a complete safe
   placement extending bd, if one exists, and otherwise,
   yields the value of fc ()
*)
fun addqueen (bd, fc) =
```

```
    let
        fun try j =
            if j > Board.size bd then
               fc ()
            else if Board.safe (bd, j) then
               addqueen
                 (Board.place (bd, j),
                  fn () => try (j+1))
            else
               try (j+1)
    in
       if Board.complete bd then
          SOME bd
       else
          try 1
    end
fun queens n =
    addqueen (Board.new n, fn () => NONE)
```

Here again the differences are small, but significant. The initial continuation simply yields `NONE`, reflecting the ultimate failure to find a safe placement. On a recursive call we pass to `addqueen` a continuation that resumes search at the next row of the current column. Should we exceed the number of rows on the board, we invoke the failure continuation of the most recent call to `addqueen`.

The solution based on continuations is very close to the solution based on exceptions, both in form and in terms of efficiency. Which is preferable? Here again there is no easy answer, we can only offer general advice. First off, as we've seen in the case of regular expression matching, failure continuations are more powerful than exceptions; there is no obvious way to replace the use of a failure continuation with a use of exceptions in the matcher. However, in the case that exceptions would suffice, it is generally preferable to use them since one may then avoid passing an explicit failure continuation. More significantly, the compiler ensures that an uncaught exception aborts the program gracefully, whereas failure to invoke a continuation is not in itself a run-time fault. Using the right tool for the right job makes life easier.

29.5 Sample Code

Here is the code for this chapter.

Chapter 30

Higher-Order Functions

Higher-order functions — those that take functions as arguments or return functions as results — are powerful tools for building programs. An interesting application of higher-order functions is to implement *infinite sequences* of values as (total) functions from the natural numbers (non-negative integers) to the type of values of the sequence. We will develop a small package of operations for creating and manipulating sequences, all of which are higher-order functions since they take sequences (functions!) as arguments and/or return them as results. A natural way to define many sequences is by recursion, or self-reference. Since sequences are functions, we may use recursive function definitions to define such sequences. Alternatively, we may think of such a sequence as arising from a "loopback" or "feedback" construct. We will explore both approaches.

Sequences may be used to simulate digital circuits by thinking of a "wire" as a sequence of bits developing over time. The ith value of the sequence corresponds to the signal on the wire at time i. For simplicity we will assume a perfect waveform: the signal is always either high or low (or is undefined); we will not attempt to model electronic effects such as attenuation or noise. Combinational logic elements (such as and gates or inverters) are operations on wires: they take in one or more wires as input and yield one or more wires as results. Digital logic elements (such as flip-flops) are obtained from combinational logic elements by feedback, or recursion — a flip-flop is a recursively-defined wire!

30.1 Infinite Sequences

Let us begin by developing a sequence package. Here is a suitable signature defining the type of sequences:

```
signature SEQUENCE =
sig
  type 'a seq = int -> 'a
  (* constant sequence *)
  val constantly : 'a -> 'a seq
  (* alternating values *)
  val alternately : 'a * 'a -> 'a seq
  (* insert at front *)
  val insert : 'a * 'a seq -> 'a seq
  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val zip : 'a seq * 'b seq -> ('a * 'b) seq
  val unzip : ('a * 'b) seq -> 'a seq * 'b seq
  (* fair merge *)
  val merge : ('a * 'a) seq -> 'a seq
  val stretch : int -> 'a seq -> 'a seq
  val shrink : int -> 'a seq -> 'a seq
  val take : int -> 'a seq -> 'a list
  val drop : int -> 'a seq -> 'a seq
  val shift : 'a seq -> 'a seq
  val loopback : ('a seq -> 'a seq) -> 'a seq
end
```

Observe that we expose the representation of sequences as functions. This is done to simplify the definition of recursive sequences as recursive functions. Alternatively we could have hidden the representation type, at the expense of making it a bit more awkward to define recursive sequences. In the absence of this exposure of representation, recursive sequences may only be built using the `loopback` operation which constructs a recursive sequence by "looping back" the output of a sequence transformer to its input. Most of the other operations of the signature are adaptations of familiar operations on lists. Two exceptions to this rule are the functions `stretch` and `shrink` that dilate and contract the sequence by a given time

parameter — if a sequence is expanded by k, its value at i is the value of the original sequence at i/k, and dually for shrinking.

Here's an implementation of sequences as functions.

```
structure Sequence :> SEQUENCE =
struct
  type 'a seq = int -> 'a
  fun constantly c n = c
  fun alternately (c,d) n =
      if n mod 2 = 0 then c else d
  fun insert (x, s) 0 = x
    | insert (x, s) n = s (n-1)
  fun map f s = f o s
  fun zip (s1, s2) n = (s1 n, s2 n)
  fun unzip (s : ('a * 'b) seq) =
      (map #1 s, map #2 s)
  fun merge (s1, s2) n =
      (if n mod 2 = 0 then s1 else s2) (n div 2)
  fun stretch k s n = s (n div k)
  fun shrink k s n = s (n * k)
  fun drop k s n = s (n+k)
  fun shift s = drop 1 s
  fun take 0 _ = nil
    | take n s = s 0 :: take (n-1) (shift s)
  fun loopback loop n = loop (loopback loop) n
end
```

Most of this implementation is entirely straightforward, given the ease with which we may manipulate higher-order functions in ML. The only tricky function is `loopback`, which must arrange that the output of the function `loop` is "looped back" to its input. This is achieved by a simple recursive definition of a sequence whose value at n is the value at n of the sequence resulting from applying the loop to this very sequence.

The sensibility of this definition of `loopback` relies on two separate ideas. First, notice that we may *not* simplify the definition of `loopback` as follows:

```
(* bad definition *)
fun loopback loop = loop (loopback loop)
```

The reason is that any application of `loopback` will immediately loop forever! In contrast, the original definition is arranged so that application of `loopback` immediately returns a function. This may be made more apparent by writing it in the following form, which is entirely equivalent to the definition given above:

```
fun loopback loop =
    fn n => loop (loopback loop) n
```

This format makes it clear that loopback immediately returns a function when applied to a loop functional.

Second, for an application of `loopback` to a loop to make sense, it must be the case that the loop returns a sequence without "touching" the argument sequence (*i.e.*, without applying the argument to a natural number). Otherwise accessing the sequence resulting from an application of loopback would immediately loop forever. Some examples will help to illustrate the point.

First, let's build a few sequences without using the `loopback` function, just to get familiar with using sequences:

```
val evens : int seq = fn n => 2*n
val odds : int seq = fn n => 2*n+1
val nats : int seq = merge (evens, odds)
fun fibs n =
    (insert
      (1, insert
           (1, map (op +)
                   (zip (drop 1 fibs, fibs)))))(n)
```

We may "inspect" the sequence using `take` and `drop`, as follows:

```
take 10 nats           (* [0,1,2,3,4,5,6,7,8,9] *)
take 5 (drop 5 nats)   (* [5,6,7,8,9] *)
take 5 fibs            (* [1,1,2,3,5] *)
```

Now let's consider an alternative definition of `fibs` that uses the loopback operation:

```
fun fibs_loop s =
    insert (1, insert (1,
                map (op +) (zip (drop 1 s, s))))
val fibs = loopback fibs_loop;
```

The definition of `fibs_loop` is exactly like the original definition of `fibs`, except that the reference to `fibs` itself is replaced by a reference to the argument `s`. Notice that the application of `fibs_loop` to an argument `s` does not inspect the argument `s`!

One way to understand loopback is that it solves a system of equations for an unknown sequence. In the case of the second definition of fibs, we are solving the following system of equations for f:

$$\begin{aligned} f0 &= 1 \\ f1 &= 1 \\ f(n+2) &= f(n+1) + f(n) \end{aligned}$$

These equations are derived by inspecting the definitions of `insert`, `map`, `zip`, and `drop` given earlier. It is obvious that the solution is the Fibonacci sequence; this is precisely the sequence obtained by applying `loopback` to `fibs_loop`.

Here's an example of a loop that, when looped back, yields an undefined sequence — any attempt to access it results in an infinite loop:

```
fun bad_loop s n = s n + 1
val bad = loopback bad_loop
val _ = bad 0    (* infinite loop! *)
```

In this example we are, in effect, trying to solve the equation $sn = sn + 1$ for s, which has no solution (except the totally undefined sequence). The problem is that the "next" element of the output is defined in terms of the next element itself, rather than in terms of "previous" elements. Consequently, no solution exists.

30.2 Circuit Simulation

With these ideas in mind, we may apply the sequence package to build an implementation of digital circuits. Let's start with wires, which are represented as sequences of levels:

```
datatype level = High | Low | Undef
type wire = level seq
type pair = (level * level) seq
val Zero : wire = constantly Low
val One : wire = constantly High
(* clock pulse with given duration of each pulse *)
fun clock (freq:int):wire =
    stretch freq (alternately (Low, High))
```

We include the "undefined" level to account for propagation delays and settling times in circuit elements.

Combinational logic elements (gates) may be defined as follows. We introduce an explicit unit time propagation delay for each gate — the output is undefined initially, and is then determined as a function of its inputs. As we build up layers of circuit elements, it takes longer and longer (proportional to the length of the longest path through the circuit) for the output to settle, exactly as in "real life".

```
(* apply two functions in parallel *)
infixr **;
fun (f ** g) (x, y) = (f x, g y)
(* hardware logical and *)
fun logical_and (Low, _) = Low
  | logical_and (_, Low) = Low
  | logical_and (High, High) = High
  | logical_and _ = Undef
fun logical_not Undef = Undef
  | logical_not High = Low
  | logical_not Low = High
fun logical_nop l = l
(* a nor b = not a and not b *)
val logical_nor =
    logical_and o (logical_not ** logical_not)
type unary_gate = wire -> wire
type binary_gate = pair -> wire
fun gate f w 0 = Undef
(* logic gate with unit propagation delay *)
```

```
  | gate f w i = f (w (i-1))
val delay : unary_gate = gate logical_nop   (* unit delay *)
val inverter : unary_gate = gate logical_not
val nor_gate : binary_gate = gate logical_nor
```

It is a good exercise to build a one-bit adder out of these elements, then to string them together to form an n-bit ripple-carry adder. Be sure to present the inputs to the adder with sufficient pulse widths to ensure that the circuit has time to settle!

Combining these basic logic elements with recursive definitions allows us to define digital logic elements such as the RS flip-flop. The propagation delay inherent in our definition of a gate is fundamental to ensuring that the behavior of the flip-flop is well-defined! This is consistent with "real life" — flip-flop's depend on the existence of a hardware propagation delay for their proper functioning. Note also that presentation of "illegal" inputs (such as setting both the R and the S leads high results in metastable behavior of the flip-flop, here as in real life Finally, observe that the flip-flop exhibits a momentary "glitch" in its output before settling, exactly as in the hardware case. (All of these behaviors may be observed by using `take` and `drop` to inspect the values on the circuit.)

```
fun RS_ff (S : wire, R : wire) =
    let
        fun X n = nor_gate (zip (S, Y))(n)
        and Y n = nor_gate (zip (X, R))(n)
    in
        Y
    end
(* generate a pulse of b's n wide, followed by w *)
fun pulse b 0 w i = w i
  | pulse b n w 0 = b
  | pulse b n w i = pulse b (n-1) w (i-1)
val S = pulse Low 2 (pulse High 2 Zero);
val R = pulse Low 6 (pulse High 2 Zero);
val Q = RS_ff (S, R);
val _ = take 20 Q;
val X = RS_ff (S, S);            (* unstable! *)
val _ = take 20 X;
```

It is a good exercise to derive a system of equations governing the RS flip-flop from the definition we've given here, using the implementation of the sequence operations given above. Observe that the delays arising from the combinational logic elements ensure that a solution exists by ensuring that the "next" element of the output refers only the "previous" elements, and not the "current" element.

Finally, we consider a variant implementation of an RS flip-flop using the loopback operation:

```
fun loopback2 (f : wire * wire -> wire * wire) =
    unzip (loopback (zip o f o unzip))
fun RS_ff' (S : wire, R : wire) =
    let
        fun RS_loop (X, Y) =
            (nor_gate (zip (S, Y)),
             nor_gate (zip (X, R)))
    in
        loopback2 RS_loop
    end
```

Here we must define a "binary loopback" function to implement the flip-flop. This is achieved by reducing binary loopback to unary loopback by composing with `zip` and `unzip`.

30.3 Sample Code

Here is the code for this chapter.

Chapter 31

Memoization

In this chapter we will discuss *memoization*, a programming technique for cacheing the results of previous computations so that they can be quickly retrieved without repeated effort. Memoization is fundamental to the implementation of lazy data structures, either "by hand" or using the provisions of the SML/NJ compiler.

31.1 Cacheing Results

We begin with a discussion of memoization to increase the efficiency of computing a recursively-defined function whose pattern of recursion involves a substantial amount of redundant computation. The problem is to compute the number of ways to parenthesize an expression consisting of a sequence of n multiplications as a function of n. For example, the expression $2*3*4*5$ can be parenthesized in 5 ways:

$((2*3)*4)*5$, $(2*(3*4))*5$, $(2*3)*(4*5)$, $2*(3*(4*5))$, $2*((3*4)*5)$.

A simple recurrence expresses the number of ways of parenthesizing a sequence of n multiplications:

```
fun sum f 0 = 0
  | sum f n = (f n) + sum f (n-1)
fun p 1 = 1
  | p n = sum (fn k => (p k) * (p (n-k))) (n-1)
```

where `sum` fncomputes the sum of values of a function $f(k)$ with $1 \leq k \leq n$. This program is *extremely* inefficient because of the redundancy in the pattern of the recursive calls.

What can we do about this problem? One solution is to be clever and solve the recurrence. As it happens this recurrence has a closed-form solution (the Catalan numbers). But in many cases there is no known closed form, and something else must be done to cut down the overhead. In this case a simple cacheing technique proves effective. The idea is to maintain a table of values of the function that is filled in whenever the function is applied. If the function is called on an argument n, the table is consulted to see whether the value has already been computed; if so, it is simply returned. If not, we compute the value and store it in the table for future use. This ensures that no redundant computations are performed. We will maintain the table as an array so that its entries can be accessed in constant time. The penalty is that the array has a fixed size, so we can only record the values of the function at some pre-determined set of arguments. Once we exceed the bounds of the table, we must compute the value the "hard way". An alternative is to use a dictionary (*e.g.*, a balanced binary search tree) which has no *a priori* size limitation, but which takes logarithmic time to perform a lookup. For simplicity we'll use a solution based on arrays.

Here's the code to implement a memoized version of the parenthesization function:

```
local
  val limit = 100
  val memopad = Array.array (100, NONE)
in
  fun p' 1 = 1
    | p' n = sum (fn k => (p k) * (p (n-k))) (n-1)
  and p n =
      if n < limit then
         case Array.sub of
              SOME r => r
            | NONE =>
              let
                  val r = p' n
              in
```

```
                Array.update (memopad, n, SOME r);
                r
            end
      else
         p' n
end
```

The main idea is to modify the original definition so that the recursive calls consult and update the memopad. The "exported" version of the function is the one that refers to the memo pad. Notice that the definitions of `p` and `p'` are mutually recursive!

31.2 Laziness

Lazy evaluation is a combination of delayed evaluation and memoization. Delayed evaluation is implemented using *thunks*, functions of type `unit -> 'a`. To delay the evaluation of an expression *exp* of type `'a`, simply write `fn () =>` *exp*. This is a value of type `unit -> 'a`; the expression *exp* is effectively "frozen" until the function is applied. To "thaw" the expression, simply apply the thunk to the null tuple, `()`. Here's a simple example:

```
val thunk =
    fn () => print "hello"    (* nothing printed *)
val _ = thunk ()             (* prints hello *)
```

While this example is especially simple-minded, remarkable effects can be achieved by combining delayed evaluation with memoization. To do so, we will consider the following signature of suspensions:

```
signature SUSP =
sig
  type 'a susp
  val force : 'a susp -> 'a
  val delay : (unit -> 'a) -> 'a susp
end
```

The function `delay` takes a suspended computation (in the form of a thunk) and yields a suspension. It's job is to "memoize" the suspension

so that the suspended computation is evaluated at most once — once the result is computed, the value is stored in a reference cell so that subsequent forces are fast. The implementation is slick. Here's the code to do it:

```
structure Susp :> SUSP =
struct
  type 'a susp = unit -> 'a
  fun force t = t ()
  fun delay (t : 'a susp) =
      let
          exception Impossible
          val memo : 'a susp ref =
              ref (fn () => raise Impossible)
          fun t' () =
              let val r = t ()
              in memo := (fn () => r); r end
      in
          memo := t';
          fn () => (!memo)()
      end
end
```

It's worth discussing this code in detail because it is rather tricky. Suspensions are just thunks; `force` simply applies the suspension to the null tuple to force its evaluation. What about `delay`? When applied, `delay` allocates a reference cell containing a thunk that, if forced, raises an internal exception. This can never happen for reasons that will become apparent in a moment; it is merely a placeholder with which we initialize the reference cell. We then define another thunk `t'` that, when forced, does three things:

1. It forces the thunk `t` to obtain its value `r`.
2. It replaces the contents of the memopad with the constant function that immediately returns `r`.
3. It returns `r` as result.

We then assign `t'` to the memo pad (hence obliterating the placeholder), and return a thunk `dt` that, when forced, simply forces the contents of the memo pad. Whenever `dt` is forced, it immediately forces the contents of

the memo pad. However, the contents of the memo pad changes as a result of forcing it so that subsequent forces exhibit different behavior. Specifically, the *first* time `dt` is forced, it forces the thunk `t'`, which then forces `t` its value `r`, "zaps" the memo pad, and returns `r`. The *second* time `dt` is forced, it forces the contents of the memo pad, as before, but this time the it contains the constant function that immediately returns `r`. Altogether we have ensured that `t` is forced at most once by using a form of "self-modifying" code.

Here's an example to illustrate the effect of delaying a thunk:

```
val t = Susp.delay (fn () => print "hello")
val _ = Susp.force t    (* prints hello *)
val _ = Susp.force t    (* silent *)
```

Notice that `hello` is printed once, not twice! The reason is that the suspended computation is evaluated at most once, so the message is printed at most once on the screen.

31.3 Lazy Data Types in SML/NJ

The lazy datatype declaration[1]

```
datatype lazy 'a stream = Cons of 'a * 'a stream
```

expands into the following pair of type declarations

```
datatype 'a stream! = Cons of 'a * 'a stream
withtype 'a stream = 'a stream! Susp.susp
```

The first defines the type of stream *values*, the result of forcing a stream computation, the second defines the type of stream *computations*, which are suspensions yielding stream values. Thus streams are represented by suspended (unevaluated, memoized) computations of stream values, which are formed by applying the constructor `Cons` to a value and another stream.

The value constructor `Cons`, when used to build a stream, automatically suspends computation. This is achieved by regarding `Cons` *e* as shorthand for `Cons (Susp.susp (fn () =>` *e*`)`. When used in a pattern, the value constructor `Cons` induces a use of `force`. For example, the binding

[1] Please see chapter 15 for a description of the SML/NJ lazy data type mechanism.

```
val Cons (h, t) = e
```

becomes

```
val Cons (h, t) = Susp.force e
```

which forces the right-hand side before performing pattern matching.

A similar transformation applies to non-lazy function definitions — the argument is forced before pattern matching commences. Thus the "eager" tail function

```
fun stl (Cons (_, t)) = t
```

expands into

```
fun stl! (Cons (_, t)) = t
and stl s = stl! (Susp.force s)
```

which forces the argument as soon as it is applied.

On the other hand, lazy function definitions defer pattern matching until the result is forced. Thus the lazy tail function

```
fun lstl (Cons (_, t)) = t
```

expands into

```
fun lstl! (Cons (_, t)) =
    t
and lstl s =
    Susp.delay (fn () => lstl! (Susp.force s))
```

which a suspension that, when forced, performs the pattern match.

Finally, the recursive stream definition

```
val rec lazy ones = Cons (1, ones)
```

expands into the following recursive function definition:

```
val rec ones = Susp.delay (fn () => Cons (1, ones))
```

Unfortunately this is not quite legal in SML since the right-hand side involves an application of a a function to another function. This can either be provided by extending SML to admit such definitions, or by extending the `Susp` package to include an operation for building recursive suspensions such as this one. Since it is an interesting exercise in itself, we'll explore the latter alternative.

31.4 Recursive Suspensions

We seek to add a function to the `Susp` package with signature

```
val loopback : ('a susp -> 'a susp) -> 'a susp
```

that, when applied to a function f mapping suspensions to suspensions, yields a suspension s whose behavior is the same as $f(s)$, the application of f to the resulting suspension. In the above example the function in question is

```
fun ones_loop s = Susp.delay (fn () => Cons (1, s))
```

We use `loopback` to define `ones` as follows:

```
val ones = Susp.loopback ones_loop
```

The idea is that `ones` should be equivalent to `Susp.delay (fn () => Cons (1, ones))`, as in the original definition and which is the result of evaluating `Susp.loopback ones_loop`, assuming `Susp.loopback` is implemented properly.

How is `loopback` implemented? We use a technique known as *backpatching*. Here's the code

```
fun loopback f =
    let
        exception Circular
        val r = ref (fn () => raise Circular)
        val t = fn () => (!r)()
    in
        r := f t ; t
    end
```

First we allocate a reference cell which is initialized to a placeholder that, if forced, raises the exception `Circular`. Then we define a thunk that, when forced, forces the contents of this reference cell. This will be the return value of `loopback`. But before returning, we assign to the reference cell the result of applying the given function to the result thunk. This "ties the knot" to ensure that the output is "looped back" to the input. Observe that if the loop function touches its input suspension before yielding an output suspension, the exception `Circular` will be raised.

Here is the code for this chapter.

31.5 Sample Code

Chapter 32

Data Abstraction

An *abstract data type (ADT)* is a type equipped with a set of operations for manipulating values of that type. An ADT is implemented by providing a *representation type* for the values of the ADT and an implementation for the operations defined on values of the representation type. What makes an ADT abstract is that the representation type is *hidden* from clients of the ADT. Consequently, the *only* operations that may be performed on a value of the ADT are the given ones. This ensures that the representation may be changed without affecting the behavior of the client — since the representation is hidden from it, the client cannot depend on it. This also facilitates the implementation of efficient data structures by imposing a condition, called a *representation invariant*, on the representation that is *preserved* by the operations of the type. Each operation that takes a value of the ADT as argument may *assume* that the representation invariant holds. In compensation each operation that yields a value of the ADT as result must *guarantee* that the representation invariant holds of it. If the operations of the ADT preserve the representation invariant, then it must truly be invariant — no other code in the system could possibly disrupt it. Put another way, any violation of the representation invariant may be localized to the implementation of one of the operations. This significantly reduces the time required to find an error in a program.

32.1 Dictionaries

To make these ideas concrete we will consider the abstract data type of *dictionaries*. A dictionary is a mapping from *keys* to *values*. For simplicity we take keys to be strings, but it is possible to define a dictionary for any ordered type; the values associated with keys are completely arbitrary. Viewed as an ADT, a dictionary is a type `'a dict` of dictionaries mapping strings to values of type `'a` together with `empty`, `insert`, and `lookup` operations that create a new dictionary, insert a value with a given key, and retrieve the value associated with a key (if any). In short a dictionary is an implementation of the following signature:

```
signature DICT =
sig
  type key = string
  type 'a entry = key * 'a
  type 'a dict
  exception Lookup of key
  val empty : 'a dict
  val insert : 'a dict * 'a entry -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

Notice that the type `'a dict` is not specified in the signature, whereas the types `key` and `'a entry` are defined to be `string` and `string * 'a`, respectively.

32.2 Binary Search Trees

A simple implementation of a dictionary is a *binary search tree*. A binary search tree is a binary tree with values of an ordered type at the nodes arranged in such a way that for every node in the tree, the value at that node is greater than the value at any node in the left child of that node, and smaller than the value at any node in the right child. It follows immediately that no two nodes in a binary search tree are labelled with the same value. The binary search tree property is an example of a representation invariant on an underlying data structure. The underlying structure is a

binary tree with values at the nodes; the representation invariant isolates a set of structures satisfying some additional, more stringent, conditions.

We may use a binary search tree to implement a dictionary as follows:

```
structure BinarySearchTree :> DICT =
struct
  type key = string
  type 'a entry = key * 'a
  (* Rep invariant: 'a tree is a binary search tree *)
  datatype 'a tree =
    Empty |
    Node of 'a tree * 'a entry * 'a tree
  type 'a dict = 'a tree
  exception Lookup of key
  val empty = Empty
  fun insert (Empty, entry) =
      Node (Empty, entry, Empty)
    | insert (n as Node (l, e as (k,_), r), e' as (k',_)) =
      (case String.compare (k', k)
         of LESS => Node (insert (l, e'), e, r)
          | GREATER => Node (l, e, insert (r, e'))
          | EQUAL => n)
  fun lookup (Empty, k) = raise (Lookup k)
    | lookup (Node (l, (k, v), r), k') =
      (case String.compare (k', k)
         of EQUAL => v
          | LESS => lookup (l, k')
          | GREATER => lookup (r, k'))
end
```

Notice that `empty` is defined to be a valid binary search tree, that `insert` yields a binary search tree if its argument is one, and that `lookup` relies on its argument being a binary search tree (if not, it might fail to find a key that in fact occurs in the tree!). The structure `BinarySearchTree` is sealed with the signature `DICT` to ensure that the representation type is held abstract.

32.3 Balanced Binary Search Trees

The difficulty with binary search trees is that they may become unbalanced. In particular if we insert keys in ascending order, the representation is essentially just a list! The left child of each node is empty; the right child is the rest of the dictionary. Consequently, it takes $O(n)$ time in the worse case to perform a lookup on a dictionary containing n elements. Such a tree is said to be *unbalanced* because the children of a node have widely varying heights. Were it to be the case that the children of every node had roughly equal height, then the lookup would take $O(\lg n)$ time, a considerable improvement.

Can we do better? Many approaches have been suggested. One that we will consider here is an instance of what is called a *self-adjusting tree*, called a *red-black tree* (the reason for the name will be apparent shortly). The general idea of a self-adjusting tree is that operations on the tree may cause a reorganization of its structure to ensure that some invariant is maintained. In our case we will arrange things so that the tree is *self-balancing*, meaning that the children of any node have roughly the same height. As we just remarked, this ensures that lookup is efficient.

How is this achieved? By imposing a clever representation invariant on the binary search tree, called the *red-black tree* condition. A red-black tree is a binary search tree in which every node is colored either red or black (with the empty tree being regarded as black) and such that the following properties hold:

1. The children of a red node are black.

2. For any node in the tree, the number of black nodes on any two paths from that node to a leaf is the same. This number is called the *black height* of the node.

These two conditions ensure that a red-black tree is a balanced binary search tree. Here's why. First, observe that a red-black tree of black height h has at least $2^h - 1$ nodes. We may prove this by induction on the structure of the red-black tree. The empty tree has black-height 1 (since we consider it to be black), which is at least $2^1 - 1$, as required. Suppose we have a red node. The black height of both children must be h, hence each has at most $2^h - 1$ nodes, yielding a total of $2 \times (2^h - 1) + 1 = 2^{h+1} - 1$ nodes, which is at least $2^h - 1$. If, on the other hand, we have a black node, then the black

height of both children is $h-1$, and each have at most $2^{h-1}-1$ nodes, for a total of $2 \times (2^{h-1}-1)+1 = 2^h - 1$ nodes. Now, observe that a red-black tree of height h with n nodes has black height at least $h/2$, and hence has at least $2^{h/2}-1$ nodes. Consequently, $lg(n+1) \geq h/2$, so $h \leq 2 \times lg(n+1)$. In other words, its height is logarithmic in the number of nodes, which implies that the tree is height balanced.

To ensure logarithmic behavior, all we have to do is to maintain the red-black invariant. The empty tree is a red-black tree, so the only question is how to perform an insert operation. First, we insert the entry as usual for a binary search tree, with the fresh node starting out colored red. In doing so we do not disturb the black height condition, but we might introduce a *red-red violation*, a situation in which a red node has a red child. We then remove the red-red violation by propagating it upwards towards the root by a constant-time transformation on the tree (one of several possibilities, which we'll discuss shortly). These transformations either eliminate the red-red violation outright, or, in logarithmic time, push the violation to the root where it is neatly resolved by recoloring the root black (which preserves the black-height invariant!).

The violation is propagated upwards by one of four *rotations*. We will maintain the invariant that there is at most one red-red violation in the tree. The insertion may or may not create such a violation, and each propagation step will preserve this invariant. It follows that the parent of a red-red violation must be black. Consequently, the situation must look like this. This diagram represents four distinct situations, according to whether the uppermost red node is a left or right child of the black node, and whether the red child of the red node is itself a left or right child. In each case the red-red violation is propagated upwards by transforming it to look like this. Notice that by making the uppermost node red we may be introducing a red-red violation further up the tree (since the black node's parent might have been red), and that we are preserving the black-height invariant since the great-grand-children of the black node in the original situation will appear as children of the two black nodes in the re-organized situation. Notice as well that the binary search tree conditions are also preserved by this transformation. As a limiting case if the red-red violation is propagated to the root of the entire tree, we re-color the root black, which preserves the black-height condition, and we are done re-balancing the tree.

Let's look in detail at two of the four cases of removing a red-red vio-

lation, those in which the uppermost red node is the left child of the black node; the other two cases are handled symmetrically. If the situation looks like this, we reorganize the tree to look like this. You should check that the black-height and binary search tree invariants are preserved by this transformation. Similarly, if the situation looks like this, then we reorganize the tree to look like this (precisely as before). Once again, the black-height and binary search tree invariants are preserved by this transformation, and the red-red violation is pushed further up the tree.

Here is the ML code to implement dictionaries using a red-black tree. Notice that the tree rotations are neatly expressed using pattern matching.

```
structure RedBlackTree :> DICT =
struct
  type key = string
  type 'a entry = string * 'a
  (* Inv: binary search tree + red-black conditions *)
  datatype 'a dict =
    Empty |
    Red of 'a entry * 'a dict * 'a dict |
    Black of 'a entry * 'a dict * 'a dict
  val empty = Empty
  exception Lookup of key
  fun lookup (dict, key) =
      let
          fun lk (Empty) = raise (Lookup key)
            | lk (Red tree) = lk' tree
            | lk (Black tree) = lk' tree
          and lk' ((key1, datum1), left, right) =
              (case String.compare(key,key1)
               of EQUAL => datum1
                | LESS => lk left
                | GREATER => lk right)
      in
          lk dict
      end
  fun restoreLeft
        (Black (z, Red (y, Red (x, d1, d2), d3), d4)) =
      Red (y, Black (x, d1, d2), Black (z, d3, d4))
    | restoreLeft
```

```
        (Black (z, Red (x, d1, Red (y, d2, d3)), d4)) =
      Red (y, Black (x, d1, d2), Black (z, d3, d4))
    | restoreLeft dict = dict
  fun restoreRight
        (Black (x, d1, Red (y, d2, Red (z, d3, d4)))) =
      Red (y, Black (x, d1, d2), Black (z, d3, d4))
    | restoreRight
        (Black (x, d1, Red (z, Red (y, d2, d3), d4))) =
      Red (y, Black (x, d1, d2), Black (z, d3, d4))
    | restoreRight dict = dict
  fun insert (dict, entry as (key, datum)) =
      let
          (* val ins : 'a dict->'a dict insert entry *)
          (* ins (Red _) may have red-red at root *)
          (* ins (Black _) or ins (Empty) is red/black *)
          (* ins preserves black height *)
          fun ins (Empty) = Red (entry, Empty, Empty)
            | ins (Red (entry1 as (key1, datum1), left, right)) =
              (case String.compare (key, key1)
                 of EQUAL => Red (entry, left, right)
                  | LESS => Red (entry1, ins left, right)
                  | GREATER => Red (entry1, left, ins right))
            | ins (Black (entry1 as (key1, datum1), left, right)) =
              (case String.compare (key, key1)
                 of EQUAL => Black (entry, left, right)
                  | LESS => restoreLeft (Black (entry1, ins left, right))
                  | GREATER => restoreRight (Black (entry1, left, ins right)))
      in
          case ins dict
            of Red (t as (_, Red _, _)) => Black t (* re-color *)
             | Red (t as (_, _, Red _)) => Black t (* re-color *)
             | dict => dict
      end
end
```

It is worthwhile to contemplate the role played by the red-black invariant in ensuring the correctness of the implementation and the time complexity of the operations.

32.4 Abstraction *vs.* Run-Time Checking

You might wonder whether we could equally well use run-time checks to enforce representation invariants. The idea would be to introduce a "debug flag" that, when set, causes the operations of the dictionary to check that the representation invariant holds of their arguments and results. In the case of a binary search tree this is surely possible, but at considerable expense since the time required to check the binary search tree invariant is proportional to the size of the binary search tree itself, whereas an insert (for example) can be performed in logarithmic time. But wouldn't we turn off the debug flag before shipping the production copy of the code? Yes, indeed, but then the benefits of checking are lost for the code we care about most! (To paraphrase Tony Hoare, it's as if we used our life jackets while learning to sail on a pond, then tossed them away when we set out to sea.) By using the type system to enforce abstraction, we can confine the possible violations of the representation invariant to the dictionary package itself, and, moreover, we need not turn off the check for production code because there is no run-time penalty for doing so.

A more subtle point is that it may not always be possible to enforce data abstraction at run-time. Efficiency considerations aside, you might think that we can always replace static localization of representation errors by dynamic checks for violations of them. But this is false! One reason is that the representation invariant might not be computable. As an example, consider an abstract type of *total* functions on the integers, those that are guaranteed to terminate when called, without performing any I/O or having any other computational effect. It is a theorem of recursion theory that no run-time check can be defined that ensures that a given integer-valued function is total. Yet we can define an abstract type of total functions that, while not admitting every possible total function on the integers as values, provides a useful set of such functions as elements of a structure. By using these specified operations to create a total function, we are in effect encoding a proof of totality in the code itself.

Here's a sketch of such a package:

```
signature TIF = sig
  type tif
  val apply : tif -> (int -> int)
  val id : tif
```

```
  val compose : tif * tif -> tif
  val double : tif
  ⋮
end
structure Tif :> TIF = struct
  type tif = int->int
  fun apply t n = t n
  fun id x = x
  fun compose (f, g) = f o g
  fun double x = 2 * x
  ⋮
end
```

Should the application of such some value of type `Tif.tif` fail to terminate, we know where to look for the error. No run-time check can assure us that an arbitrary integer function is in fact total.

Another reason why a run-time check to enforce data abstraction is impossible is that it may not be possible to tell from looking at a given value whether or not it is a legitimate value of the abstact type. Here's an example. In many operating systems processes are "named" by integer-value process identifiers. Using the process identifier we may send messages to the process, cause it to terminate, or perform any number of other operations on it. The thing to notice here is that any integer at all is a possible process identifier; we cannot tell by looking at the integer whether it is indeed valid. No run-time check on the value will reveal whether a given integer is a "real" or "bogus" process identifier. The only way to know is to consider the "history" of how that integer came into being, and what operations were performed on it. Using the abstraction mechanisms just described, we can enforce the requirement that a value of type `pid`, whose underlying representation is `int`, is indeed a process identifier. You are invited to imagine how this might be achieved in ML.

32.5 Sample Code

Here is the code for this chapter.

Chapter 33

Representation Independence and ADT Correctness

This chapter is concerned with proving correctness of ADT implementations by exhibiting a simulation relation between a reference implementation (taken, or known, to be correct) and a candidate implementation (whose correctness is to be established). The methodology generalizes Hoare's notion of abstraction functions to an arbitrary relation, and relies on Reynolds' notion of parametricity to conclude that related implementations engender the same observable behavior in all clients.

33.1 Sample Code

Here is the code for this chapter.

Chapter 34

Modularity and Reuse

1. Naming conventions.
2. Exploiting structural subtyping (`type t` convention).
3. Impedance-matching functors.

34.1 Sample Code

Here is the code for this chapter.

Chapter 35

Dynamic Typing and Dynamic Dispatch

This chapter is concerned with dynamic typing in a statically typed language. It is commonly thought that there is an "opposition" between statically-typed languages (such as Standard ML) and dynamically-typed languages (such as Scheme). In fact, dynamically typed languages are a special case of statically-typed languages! We will demonstrate this by exhibiting a faithful representation of Scheme inside of ML.

35.1 Sample Code

Here is the code for this chapter.

Chapter 36

Concurrency

In this chapter we consider some fundamental techniques for concurrent programming using CML.

36.1 Sample Code

Here is the code for this chapter.

Part V

Appendices

The Standard ML Basis Library

The *Standard ML Basis Library* is a collection of modules providing a basic collection of abstract types that are shared by all implementations of Standard ML. All of the primitive types of Standard ML are defined in structures in the Standard Basis. It also defines a variety of other commonly-used abstract types.

Most implementations of Standard ML include module libraries implementing a wide variety of services. These libraries are usually not portable across implementations, particularly not those that are concerned with the internals of the compiler or its interaction with the host computer system. Please refer to the documentation of your compiler for information on its libraries.

Compilation Management

All program development environments provide tools to support building systems out of collections of separately-developed modules. These tools usually provide services such as:

1. *Source code management* such as version and revision control.
2. *Separate compilation and linking* to support simultaneous development and to reduce build times.
3. *Libraries of re-usable modules* with consistent conventions for identifying modules and their components.
4. *Release management* for building and disseminating systems for general use.

Different languages, and different vendors, support these activities in different ways. Some rely on generic tools, such as the familiar Unix tools, others provide proprietary tools, commonly known as IDE's (integrated development environments).

Most implementations of Standard ML rely on a combination of generic program development tools and tools specific to that implementation of the language. Rather than attempt to summarize all of the known implementations, we will instead consider the SML/NJ *Compilation Manager (CM)* as a representative program development framework for ML. Other compilers provide similar tools; please consult your compiler's documentation for details of how to use them.

36.2 Overview of CM

36.3 Building Systems with CM

36.4 Sample Code

Here is the code for this chapter.

Sample Programs

A number of example programs illustrating the concepts discussed in the preceding chapters are available in the Sample Code directory on the world-wide web.

Bibliography

[1] Emden R. Gansner and John H. Reppy, editors. *The Standard ML Basis Library*. Cambridge University Press, 2000.

[2] Peter Lee. Standard ML at Carnegie Mellon. Available within CMU at `http://www.cs.cmu.edu/afs/cs/local/sml/common/smlguide`.

[3] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.