Programming in Standard ML

Robert Harper

School of Computer Science

Carnegie Mellon University

Spring, 1998

These notes are intended as a brief introduction to Standard ML (1997 dialect) for the experienced programmer. They began as lecture notes for *15-212: Fundamental Principles of Computer Science II*, the second semester of the introductory sequence in the undergraduate computer science curriculum at Carnegie Mellon University. They have subsequently been used in several other courses at Carnegie Mellon, and at a number of universities around the world. These notes are intended to supersede my *Introduction to Standard ML*, which has been widely circulated over the last ten years.

The Definition of Standard ML (Revised) by Robin Milner, Mads Tofte, Robert Harper, and David MacQueen (MIT Press, 1997) constitutes the official 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 Standard ML. The two most popular introductory programming textbooks based on Standard ML are: Lawrence Paulson, *ML for the Working Programmer (Second Edition)*, MIT Press, 1997, and Jeffrey Ullman, *Elements of ML Programming*, Prentice-Hall, 1994.

There are several implementations of Standard ML available for a variety of hardware and software platforms. *Standard ML of New Jersey* is a comprehensive research implementation, and is the most widely used. Harlequin's *MLWorks* is a commercial implementation that provides a substantial set of program development and analysis tools. Other implementations include two other research implementations, *MLKit* and *Moscow ML*, and another commercial implementation, *Poly ML*, from Abstract Hardware Ltd. *Concurrent ML* is an extension of Standard ML with primitives for concurrent programming; it is available as part of the *Standard ML of New Jersey* compiler. (For users at Carnegie Mellon, see the CMU local guide for information about using Standard ML.)

These notes are a work in progress. I am making regular updates, so please check back for changes. The most recent revision was made onTuesday, May 05, 1998 12:32 PM. Corrections, comments and suggestions are welcome.

For users who are not able to browse this web site, I have prepared a complete draft (in Postscript

format) for downloading. This copy is updated infrequently; please refer to the web pages for the latest revisions.

[Table of Contents][Overview of Standard ML][Core Language][Module Language]
[Programming Techniques][Sample Programs][Basis Library]

Table of Contents

[Home][Next]

Last edit: *Monday, April 27, 1998 03:12 PM*

Programming in Standard ML

- samplecode/optexccont.sml
- samplecode/regexp.sml
- samplecode/repinv.sml
- samplecode/memo.sml
- samplecode/seq.sml
- samplecode/streams.sml

- Basis Library

[Home][Next]

Overview of Standard ML

[Back][Home][Next]

Last edit: *Thursday, June 25, 1998 11:36 AM*

Sample Code for this Chapter

Standard ML is a type-safe programming language that embodies many innovative ideas in programming language design. It is a statically-typed language, with a user-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 (*e.g.*, for handling I/O or implementing mutable data structures). It facilitates programming with recursive data structures (such as trees and lists) by encouraging 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 mechanism for enforcing abstraction, imposing hierarchical structure, and building generic modules. It is portable across platforms and implementations because it has a precise definition given by a formal operational semantics that defines both the static and dynamic semantics of the language. It provides a portable standard basis library that defines a rich collection of commonly-used types and routines.

These features are supported by all implementations of Standard ML, but many go beyond the standard to provide experimental language features, more extensive libraries, and handy program development tools. Details can be found with the documentation for your compiler, but here's a brief overview of what you might expect. Most implementations provide an interactive system supporting on-line entry and execution of ML programs and providing access to tools for compiling, linking, and analyzing the behavior of programs. A few compilers are "batch only", relying on the ambient operating system to manage the construction of large programs from compiled parts. Nearly every compiler is capable of generating native machine code, even in the interactive system, but some optionally generate byte codes for a portable abstract machine. Most implementations support separate compilation and incremental recompilation based on automatically-generated or manually-constructed component dependency specifications. Some implementations provide interactive 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 experimental language extensions, notably mechanisms for concurrent programming (using message-passing or locking), richer forms of modularity constructs, and support for "lazy" data structures.

To develop a feel for the language and how it is used, let us consider a small, but non-trivial, program to implement a regular expression package for checking whether a given string matches a given regular expression. We'll structure the implementation into two modules, an implementation of

regular expressions themselves and an implementation of a matching algorithm for them. The structure of the system is neatly expressed using *signatures* that describe the components of these two modules.

```
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 expressions, together with operations for parsing and unparsing (formatting) them. The definition of the abstract syntax 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 6 clauses, corresponding to the regular expressions **0**, **1**, ***a***, ***r1*** + ***r2***, ***r1 r2***, and r^*. 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 signature `MATCHER` describes a module that implements a matching algorithm for regular expressions. The matcher is a function `match` that takes a regular expression and yields a function that takes a string and determines whether or not that string matches that regular 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.

Now let's look at the high-level structure of an implementation of a regular expression matcher. It

consists of two major components: an implementation of regular expressions, and an implementation of the matcher. Implementations of signatures are called *structures* in ML; the implementation of the regular expression matcher consists of two structures. Since the implementation of the matcher depends on an implementation of regular expressions, but is independent of any particular implementation of regular expressions, we use a *parameterized module*, or *functor*, to implement it. Here's the high-level structure we're considering:

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

functor Matcher (structure RegExp : REGEXP) :> MATCHER =
...

structure Matcher :> MATCHER = Matcher (structure RegExp =
RegExp)
```

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

The functor identifier `Matcher` is bound to a structure that takes an implementation of `REGEXP` as parameter. We may think of `Matcher` as a kind of function mapping structures to structures. The result signature of the functor specifies that the implementation must conform to the requirements of the signature `MATCHER`, and ensures that only what is specified in that signature is visible of any instance of this functor (obtained by applying it to an implementation of `REGEXP`). A specific matcher is provided by applying the functor `Matcher` to the stucture `RegExp` to obtain an implementation of `MATCHER`.

Once the system is built, we may use it 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 this to `Matcher.match`. Here's an example:

```
val regexp = Matcher.RegExp.parse "((a + %).(b + %))*"
val matches = Matcher.match regexp

matches "aabba"
matches "abac"
```

We use the convention that "@" stands for the empty regular expression and "%" stands for the regular expression accepting only the null string. Concatentation is indicated by a ".", alternation by "+", and iteration by "*".

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

matches "aabba"
matches "abac"
```

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

```
open Matcher Matcher.RegExp

val regexp = parse "((a + %).(b + %))*"
val matches = match regexp

matches "aabba"
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 these structures. Here's an overview of the implementation of regular expressions:

```
structure RegExp :> REGEXP = struct

  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp | Times of regexp * regexp |
    Star of regexp

  ... implementation of the tokenizer ...

  fun tokenize s = tokenize_exp (String.explode s)

  ... implementation of the parser components ...

  fun parse s =
      let
          val (r, s') = parse_exp (tokenize (String.explode
s))
      in
          case s'
            of nil => r
             | _ => raise SyntaxError "Unexpected input.\n"
      end
      handle LexicalError => raise SyntaxError "Illegal
input.\n"
```

```
    ... implementation of the formatter ...

    fun format r =
        String.implode (format_exp r)

end
```

The implementation is bracketed by the keywords `struct` and `end`. The type `regexp` is implemented precisely as specified by a `datatype` declaration. The parser works by "exploding" the string into a list of characters (making it easier to process them character-by-character), transforming the character list into a list of "tokens" (abstract symbols representing lexical atoms), and finally parsing the resulting list of tokens. If there is remaining input after the parse, or if the tokenizer encountered an illegal token, an appropriate syntax error is signalled. The formatter works by calling an associated function that yields a list of characters, then "imploding" that list into a string.

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
```

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.)

Now here's an overview of the parser. It is a simple recursive-descent parser implementing the standard precedence conventions for regular expressions (iteration binds most tightly, then concatentation, then alternation). The parser is defined by four mutually-recursive functions, `parse_exp`, `parse_term`, `parse_factor`, and `parse_atom`. These implement a recursive descent parser that dispatches on the head of the token list to determine how to proceed. The code is essentially a direct transcription of the obvious LL(1) grammar for regular expressions capturing the binding conventions described earlier.

```
fun parse_exp ts =
    let
        val (r, ts') = parse_term ts
    in
        case ts'
          of (PlusSign :: ts'') =>
             let
                 val (r', ts''') = parse_exp ts''
             in
                 (Plus (r, r'), ts''')
             end
           | _ => (r, ts')
    end

and parse_term ts = ... (elided) ...

and parse_factor ts =
    let
        val (r, ts') = parse_atom ts
    in
        case ts'
        of (Asterisk :: ts'') => (Star r, ts'')
         | _ => (r, ts')
    end

and parse_atom nil = raise SyntaxError ("Atom expected\n")
  | parse_atom (AtSign :: ts) = (Zero, ts)
  | parse_atom (Percent :: ts) = (One, ts)
  | parse_atom ((Literal c) :: ts) = (Char c, ts)
  | parse_atom (LParen :: ts) =
    let
        val (r, ts') = parse_exp ts
    in
        case ts'
          of (RParen :: ts'') => (r, ts'')
           | _ => raise SyntaxError ("Right-parenthesis
expected\n")
    end
```

Once again 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 main idea is to implement the matcher by a recursive analysis of the given regular expression. The main difficulty is to account for concatenation --- to match a string against the regular expression ***r1 r2*** we must match some initial segment against ***r1***, then match the corresponding final segment against ***r2***. 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 functor parametric in the regular expression structure:

```
functor Matcher (structure RegExp : REGEXP) :> MATCHER =
struct

  structure RegExp = RegExp

  open RegExp

  fun match_is Zero _ k = false
    | match_is One cs k = k cs
    | 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)

  fun match r s =
      match_is r (String.explode s) (fn nil => true |
false)

end
```

Note that we must incorporate the parameter structure into the result structure, 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 functions, 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 aside*: the code given above contains a subtle error. Can you find it? If not, see the chapter on proof-directed debugging 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*, ideas for building reliable and robust programs. I hope you enjoy it!

Sample Code for this Chapter

[Back][Home][Next]

Core Language

[Back][Home][Next]

Last edit: *Friday, April 24, 1998 11:20 PM*

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, *etc*. These mechanisms are the subject of this part of the notes. The second part, the *module language*, comprises the mechanisms for structuring programs into separate units and is described in the next part of these notes.

[Back][Home][Next]

Types, Values, and Effects

[Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:56 PM*

Sample Code for this Chapter

Computation in familiar programming languages such as C is based on the *imperative* model of computation described in terms of an abstract machine. The meaning of a C program is a *state transition function* that transforms the *initial* state of the abstract machine into a *final state*. The transitions consist of modifications to the memory of the abstract machine (including the registers), and having an effect on the external world (through I/O devices). The constructs of C have the flavor of commands: do something, then do something else for a while, then do something else.

Computation in ML is of an entirely different nature. In ML we compute by *calculation of expressions*, rather than *execution of instructions*. (Later in the course we will see that these two viewpoints may be reconciled, but for the time being it is best to keep a clear distinction in mind.) The calculation model is a direct 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 a value of *x*. We proceed by "plugging in" the given value for *x*, and then using the ordinary rules of arithmetic to determine the value of the polynomial. The ML model of computation is essentially just a generalization of this idea, but rather than restrict ourselves to arithmetic operations on the reals, we admit a richer variety of values and a richer variety of primitive operations on them. Much later we will generalize this model a bit further to admit effects on memory and the external world, leading to a reconciliation with the imperative model of computation with which you are familiar.

The unit of evaluation in ML is the *expression*. Every expression in Standard ML

1. … has a *type.*
2. … may or may not have a *value.*
3. … may or may not engender an *effect.*

Roughly speaking, the type of an expression in ML is a description of the sort of value it yields, should it yield a value at all. For example, if an expression has type `int`, then its values are going to be integers, and similarly, an expression of type `real` has real numbers (in practice, floating point numbers) as values. Every expression is required to have a type; otherwise, it is rejected as *ill-typed* (with a suitable explanatory message). A *well-typed* expression is evaluated (by a process of calculation) to determine its value, if indeed it has one. An expression can fail to have a value in several ways, one of which is to incur a run-time error (such as arithmetic overflow), and another of which is to compute infinitely without yielding a value. The *soundness* of the ML type system ensures that if the expression has a value, then the "shape" of that value is determined by the type of the expression. Thus, a well-typed expression of type int cannot evaluate to a string or a floating point number; it must be an integer. As we will see (much) later it is also possible for evaluation to engender an *effect* on the computing environment, for example by writing to the window system or

requesting input from a file. For the time being we will ignore effects.

What is a type? There are many possible answers, depending on what you wish to emphasize. Here we will emphasize the role of types as determining the set of well-formed programs. Generally speaking, a type consists of

1. a *type name* standing for that type,
2. a collection of *values* of that type, and
3. a collection of *operations* on values of that type.

In other words, a type consists of a name for the type, some ways to create values of that type, and some ways for computing with values of that type.

To start off with, let's consider the type of integers. Its name is, appropriately enough, `int`. Values of type `int` are the *integer numerals* `0`, `1`, `~1`, `2`, `~2`, and so on. Notice that unary negation in SML is written using a tilde (`~`), rather than a minus sign (`-`). Operations on integers include addition and subtraction, `+` and `-`, and the operations `div` and `mod` for dividing and calculating remainders. (See the Standard ML Basis Library chapter on integers for a complete description.)

Values are one form of *atomic* expression; others will be introduced later. *Compound* expressions include atomic expressions, and also include expressions built by applying an operator to other compound expressions. The formation of expressions is governed by a set of *typing rules* that define the types of atomic expressions and determine the types of compound expressions in terms of the types of their constituent expressions.

The typing rules are generally quite intuitive since they are consistent with our experience in mathematics and in other 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 SML, 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`. Writing *exp* : *typ* to indicate that the expression *exp* has the type *typ*, we have

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

Why? In the case of the value `3`, this is an *axiom*: integer numerals have integer type, by definition. What about the expression `3+4`? Well, the addition operation takes two arguments (written on either side of the plus sign), each of which must be an integer. Since both arguments are of type `int`, it

follows that the entire expression is of type `int`. For more complex cases we proceed analogously, deducing that `(3+4) div (2+3)`: `int`, for example, by observing that `(3+4)`: `int` and `(2+3)`: `int`.

This kind of reasoning 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, `(3+4) div 5 : int` because

1. `(3+4):int`
   1.1 `3 : int`
   1.2 `4 : int`
2. `5 : int`

Implicit in this derivation is the rule for formation of `div` expressions: it has type `int` if both of its arguments have type `int`. Steps (1) and (2) justify the assertion `(3+4) div 5 : int` by demonstrating that the arguments each have type `int`. Recursively, we must justify that `(3+4)`: `int`, which follows from the subsidiary steps to step (1). Here we rely on the rule that the addition of two expressions has type `int` if both of its arguments do.

Evaluation of expressions is governed by a similar set of rules, called *evaluation rules*, that determine how the value of a compound expression is determined as a function of the values of its constituent expressions. Implicit in this description is the *call-by-value* principle, which states that the arguments to an operation are evaluated *before* the operation is applied. (While this may seem intuitively obvious, it's worth mentioning that not all languages adhere to this principle.)

We write *exp* => *val* to indicate that the expression *exp* has value *val*. Informally, it is easy to see that

```
5 => 5
2+3 => 5
(2+3) div (1+4) => 1
```

These assertions can be justified by *evaluation derivations*, which are similar in form to typing derivations. For example, we may justify the assertion `(3+2) div 5 => 1` by the derivation

1. `(3+2)=> 5`
   1.1 `3 => 3`
   1.2 `2 => 2`
2. `5 => 5`

Some things are left implicit in this derivation. First, it is an axiom that every value (in this case, a numeral) evaluates to itself; values are fully-evaluated expressions. Second, the rules of addition are used to determine that adding `3` and `2` yields `5`.

What other types are there? Here are few more *base types*, summarized briefly by their values and operations:

Type name: `real`
Values: `3.14`, `~2.17`, `0.1E6`, ...

Operations: `+`, `-`, `*`, `/`, `=`, `<`, ...

Type name: `char`
Values: `#"a"`, `#"b"`, ...
Operations: `ord`, `char`, `=`, `<`, ...

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 others (in fact, infinitely many others!), but these are enough to get us started. (See the Basis Library for a complete description of the primitive types of SML, including the ones given above.) Notice that some of the arithmetic operations for real numbers are "spelled" the same way as for integers. For example, we may write `3.1+2.7` to perform a floating point addition of two floating point numbers. On the other hand division, which is properly defined for reals, is written as `3.1/2.7` to distinguish it from the integer division operation `div`.

With these types in hand, we now have enough rope to hang ourselves by forming *ill-typed* expressions. For example, the following expressions are ill-typed:

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

The last expression may seem surprising, at first. The primitive arithmetic operations are *overloaded* in the sense that they apply *either* to integers *or* to reals, *but not both at once.* To gain some intuition, recall that at the hardware level there are two distinct arithmetic units, the integer (or fixed point) unit and the floating point unit. Each has its own separate hardware for addition, and we may not mix the two in a single instruction. Of course the compiler might be expected to sort this out for you, but then there are difficulties with round-off and overflow since different compilers might choose different combinations of conversions and operations. SML leaves this to the programmer to avoid ambiguity and problems with portability between implementations.

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.

Notice 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. While it may, at first glance, appear that this is a violation of the call-by-value principle mentioned above, the explanation is that the conditional is not a primitive function, but rather a *derived form* that is explained in terms of other constructs of the language.

A common "mistake" is to write an expression like this

```
if exp = true then exp1 else exp2
```

If you think about it for a moment, this expression is just a longer way of writing

```
if exp then exp1 else exp2
```

Similarly,

```
if exp = false then exp1 else exp2
```

can be abbreviated to

```
if not exp then exp1 else exp2
```

or, better yet, just

```
if exp then exp2 else exp1
```

Neither of these examples is really a mistake, but it is rarely clearer to test a Boolean value for equality with true or false than to simply perform a conditional test on the value itself.

Sample Code for this Chapter

[Home][Up][Next]

Variables and Declarations

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

Just as in any other programming language, values may be assigned to variables that may be used in an expression to stand for that value. However, in sharp contrast to more familiar languages, *variables in SML do not vary* (!). Values are *bound* to variables using *value bindings*; once a variable is bound to a value, it is bound for life. There is no possibility of changing the binding of a variable after it has been bound. In this respect variables in SML are more akin to variables in mathematics than to variables in languages such as C. Similarly, types may be bound to *type variables* using *type bindings*; the type variable so defined stands for the type bound to it and can never stand for any other type.

A binding (either value or type) introduces a "new" variable, distinct from all other variables of that class, for use within its range of significance, or *scope*. Scoping in SML is *lexical*, meaning that the range of significance of a variable 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 will have *global scope*, meaning that the range of significance of the variable is the "rest" of the program --- the part that lexically follows the binding. We will introduce mechanisms for delimiting the scopes of variables shortly.

Any type may be give 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 variable `float`, which subsequently is synonymous with `real`. The second introduces *two* type variables, `count` and `average`, which stand for `int` and `real`, respectively. In general a type binding introduces one or more new type variables *simultaneously* in the sense that the definitions of the type variables may not involve any of the type variables being defined. Thus a binding such as

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

nonsensical (if taken in isolation) since the type variables `float` and `average` are introduced simultaneously, and hence cannot refer to one another. The syntax for type bindings is `type` *var*1 = *typ*1 `and` ... `and` *var*n = *typn*, where each *vari* is a type variable and each *typi* is a type expression.

Similarly, value variables are bound to values using *value bindings*. 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` *var1* `:` *typ1* = *exp1* `and` ... `and` *varn* `:` *typn* = *expn*, where each *vari* is a variable, each *typi* is a type expression, and each *expi* is an expression.

As you have no doubt surmised, value bindings are type-checked by comparing the type of the right-hand side with the specified type to ensure that they coincide. If a mismatch occurs, the value binding is rejected as ill-formed. Well-typed bindings are evaluated according to the *bind-by-value* rule: the right-hand side of the binding is evaluated, and the resulting value (if any) is bound to the given variable.

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 variable `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 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 both occurrences 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.

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 *dec1 ... decn*, where *n* is at least 2. The scopes of these declarations are *nested* within one another: the scope of *dec1* includes *dec2 ... decn*, the scope of *dec2* includes *dec3 ... decn*, 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 shadows 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.)

The scope of a variable 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 (in effect) 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 shadows 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.

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

1. Type checking must take account of the declared type of a variable.
2. 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 = sqrt(2)
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 = 2.14…
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, an no change is made to the value environment. Subsequently, whenever we encounter the type variable `float` in a type expression, it is replaced by `real` in accordance with the type binding above.

Earlier we introduced two relations, the typing relation, *exp* `:` *typ*, and the evaluation relation, *exp* `=>` *val*. These two-place relations were sufficient for variable-free expressions, but in the presence of declarations these relations must be extended to account for the type and value environments. This is achieved by expanding the typing relation into a three-place relation *typenv* `|-` *exp* `:` *typ*, where *typenv* is a type environment, *exp* is an expression and *typ* is a type. (The *turnstile* symbol, "|-", is a punctuation mark separating the type environment from the expression and its type.) The type of a variable is determined by consulting the type environment; in particular, we have the following typing axiom:

```
... val x : int ... |- x : int
```

Similarly, the evaluation relation is enriched to take account of the value environment. We write *valenv* |- *exp* => *val* to indicate that *exp* evaluates to *val* in the value environment *valenv*. Variables are governed by the following axiom:

```
... val x = val ... |- x => val
```

There is an obvious similarity between the two relations.

Sample Code for this Chapter

[Back][Home][Up][Next]

Functions

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

So far Standard ML is just a glorified calculator supporting operations of various primitive types and allowing intermediate results to be bound to identifiers. What makes it possible to do more than just calculate the values of expressions is the possibility to *abstract* the data from the pattern of the computation so that the same computation may be easily repeated for various data values. For example, if we calculate the expression `2*(3+4)`, the data might be the values `2`, `3`, and `4`, and the pattern of calculation might be written in skeletal form as `( ) * ( ( ) + ( ))` with "holes" where the data used to be. We say "might be" because it's not at all clear, given the original expression, what is the data and what is the pattern. For example, we might regard `2` as the data and `( ) * (3+4)` as the pattern, or even regard `*` and `+` as the data and `2 ( ) (3 ( ) 4)` as the pattern! What is important here is that the original expression can be recovered by filling the holes with the missing data items and, moreover, different expressions can be obtained by filling the same hole with different data items. Thus, an expression with a "hole" in it is may be thought of as a *function* that, when *applied* to an *argument value* determines its result by filling the hole with the argument.

This view of functions is similar to our experience from high school algebra. In elementary algebra we manipulate polynomials such as *x^2 + 2x + 1* as a kind of expression denoting a real number, but with the variable *x* representing an unknown quantity. We may also think of a polynomial as a function of the real numbers: given a real number *x*, a polynomial determines another real number *y* computed by some combination of arithmetic operations. In fact, we sometimes write equations such as *y = x^2 + 2x + 1* or *y(x) = x^2 + 2x + 1* to denote the function 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 since we may regard the polynomial *x^2 + 2xy + y^2* as 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.

It is usually left implicit that the variables *x* and *y* range over the real numbers, and that *f*, *g*, and *h* are functions mapping real numbers to real numbers. To be fully explicit, we sometimes write something like

f : *R* -> *R* : *x in R* |--> *x^2 + 2x + 1*

to indicate that *f* is a function on the reals mapping an element *x* of *R* to the element *x^2 + 2x + 1* of *R*. This notation has the virtue of separating the binding of the function to a name (*f*) from the description of its behavior (*x in R* |--> *x^2 + 2x + 1*). This makes clear that functions are a kind of

"value" in mathematics (namely, a set of ordered pairs satisfying the usual uniqueness and existence conditions), and that the variable *f* is bound to that value 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 *Df* is its first derivative, also a function on the real line.

The treatment of functions in Standard ML is very similar to our mathematical experience, except that we stress the *algorithmic* aspects of functions (*how* they determine values from arguments), as well as the *extensional* aspects (*what* they compute). Just as in mathematics a function in Standard ML is a kind of value, namely a value of *function type*. A function type has the form *typ*`->`*typ'*, where *typ* is the *domain type* (the type of arguments to the function), and *typ'* is the *range type* (the type of results). We compute with a function by *applying* it to an *argument* value of its domain type and calculating the *result* value of its range type. Function values are *lambda expressions* of the form `fn` *var* `:` *typ* `=>` *exp*; the variable *var* is called the *parameter*, and the expression *exp* is called its *body*. It has type *typ*`->`*typ'*, where *exp* has type *typ'* under the assumption that *var* has type *typ*. The result of applying such a function to an argument value *val* is determined by temporarily adding the binding `val` *var* `=` *val* to the environment, and evaluating *exp* to a value *val'*. The temporary binding is then removed, and the result value, *val*, is returned as the value of the application.

For example, `sqrt` is a (built-in) function of type `real->real` that may be applied to a real number to obtain its square root; for example, the expression `sqrt 2.0` evaluates to `1.414...`. Observe that function application is written by juxtaposition: we simply write the argument next to the function. We can, if we wish, parenthesize the argument, writing `sqrt 2.0` for the sake of clarity; this is especially useful for expressions like `sqrt (sqrt 2.0)`. The function `sqrt` is special in that it is a built-in, or *primitive*, operation of the language. We may also define functions as templates using a notation similar to that introduced above. For example, the fourth root function on the reals may be written in Standard ML using *lambda notation* as follows:

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

Notice that we don't (at this stage) give this function a name, rather we simply define its behavior by a template specifying how it calculates its result from its argument. This template defines a function of type `real->real` since it maps real numbers to real numbers. It may be applied to an argument by writing, for example,

```
(fn x : real => sqrt (sqrt x)) (4.0)
```

to calculate the fourth root of `4.0`. The calculation proceeds by binding the variable `x` to the argument `4.0`, then evaluating the expression `sqrt (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. (There is a subtle issue about the temporary binding of `x` that we will return to later.)

We may give a function a name using the declaration forms introduced in the previous chapter. For example, we may bind the fourth root function to the identifer `fourthroot` as follows:

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

We may then write `fourthroot 4.0` to compute the fourth root of `4.0`. This notation quickly

becomes tiresome to write down, so Standard ML provides a special form of function binding that alleviates the burden. In practice we write

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

rather than the more verbose `val` declaration above. But it has (almost) precisely the same meaning: the `fun` binding binds a lambda expression to an identifier.

These examples raise a few additional points about functions in Standard ML. First of all, the general form of an application expression is *exp exp'*, where *exp* is an expression that evaluates to a function, and *exp'* is an expression that evaluates to its argument. Standard ML is a *call-by-value* language: the argument to a function is evaluated before the function is applied. (You may reasonably wonder what is the alternative. In a so-called *call-by-name* language the argument is passed in unevaluated form to the function, and is only evaluated if the function requires it to be. This behavior is expressible in Standard ML by other means, which we shall return to later.) Thus, when to evaluate an expression such as `fourthroot 2.0`, we proceed as follows:

1. Evaluate `fourthroot` to the function value `fn x : real => sqrt (sqrt x)`.
2. Evaluate the argument `2.0` to its value `2.0`
3. Bind `x` to the value `2.0`.
4. Evaluate `sqrt (sqrt x)` by a subsidiary calculation to `1.189...`.
   a. Evaluate `sqrt` to a function value (in this case the primitive square root function).
   b. Evaluate the argument expression `(sqrt x)` to its value, `1.414...` (by a subsidiary calculation).
      i. Evaluate `sqrt` to a function value (in this case the primitive square root function).
      ii. Evaluate `x` to its value, `2.0`.
      iii. Compute the square root of `2.0`, yielding `1.414...`.
   c. Compute the square root of `1.414...`, yielding `1.189...`.
5. Drop the binding for the variable `x`.

Second of all, notice that we evaluate *both* the function and argument positions of an application expression --- both the function and argument are arbitrary 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, 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. The point here is that functions are *first-class values*, meaning that they may be obtained as the value of an arbitrary 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 later in these notes.

So far, we've only considered functions on the real numbers, but we may also define functions of other types. For example,

```
fun pal (s:string):string = s ^ (rev 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`.

There are a few subtleties that we must be aware of when thinking about functions. The first is: *the name of the parameter is not important*. Consequently, it may be systematically renamed without changing the meaning of the function, *provided that* we don't rename it in such a way as to clash with some other name that is currently in scope. An example will illustrate the point:

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

These two functions are completely equivalent; they differ only in the name of the parameter (in one case, `x`, in the other, `y`). The second subtlety is the *static scope principle*: a use of a variable refers to the *nearest enclosing* binding of that variable in the text of the program. Just as one value binding can shadow another, so can parameters of functions shadow value bindings (or other parameters). Here's an example:

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

The first function, `h`, introduces a parameter `x` that shadows the outer value binding; the value binding has no effect on the meaning of the function `h`. The second function, `i`, makes use of the variable `x` introduced by the `val` binding; from within the body of `i` this is the nearest enclosing binding occurrence of `x` in the program. (The parameter `x` of the function `h` does not enclose the definition of the function `i`.) The use of `x` within the function `i` introduces some constraints on the possible renamings of the parameter of `i`. Specifically, we may certainly rename `y` to `z` without changing the meaning of the function `i`, but we may not rename `y` to `x` without changing the meaning completely. That is, the function `j` has the *same* meaning as the function `i`, but the function `k` has a *different* meaning:

```
fun j(z:real):real = x+z
fun k(x:real):real = x+x
```

While these may seem like minor technical issues, it is essential that you master these ideas now to avoid confusion later on!

We close this section with a brief summary of function types:

Type name: typ->typ'
Values: primitives, `fn` *var* `:` *typ* `=>` *exp*
Operations: application *exp exp'*

Once we develop some additional machinery we will return to the function type to discuss recursive functions.

Sample Code for this Chapter

[Back][Home][Up][Next]

Products and Patterns

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:56 PM*

Sample Code for this Chapter

A characteristic feature of ML is the the ease with which we may handle *aggregate data structures* such as tuples, arrays, lists, and trees. The simplest form of aggregate is the *tuple*, value of *product* type. Product types have the form

$typ_1 * ... * typ_n$,

where *n* is at least 2. Values of this type are *n-tuples* of the form

$(val_1, ..., val_n)$,

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

Thus the following are well-formed bindings:

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

More generally, a *tuple expression* has the form

$(exp_1, ..., exp_n)$,

where each exp_i is an expression (not necessarily a value). Evaluation of tuple expressions proceeds *from left to right*, yielding the tuple value $(val_1, ..., val_n)$, where each exp_i evaluates to val_i (for each $1<=i<=n$). Thus the binding

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

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

Tuples may be decomposed into their constituent parts using *pattern matching*. This is expressed using a generalized form of value binding in which the left-hand side is not merely a variable, but a *pattern* involving zero or more variables. The general form of a value binding is

`val` *pat* = *exp*,

where *pat* is a *pattern* and *exp* is any expression.

What sorts of patterns are there? We've already seen the basic form of pattern, namely a *variable pattern*, written *var*`:`*typ*. Another form of pattern is the *tuple pattern*, which has the form `(`pat_1`,` ... `,` pat_n`)`, where each pat_i is a pattern. (We will introduce other forms of pattern later in these notes.)

Just as every expression must have a type, so must every pattern. The type of a pattern is determined by a rule governing each form of pattern. The variable pattern *var*`:`*typ* is of type *typ*, and the tuple pattern `(`pat_1`,` ... `,` pat_n`)` is of type typ_1`*`...`*`typ_n, where pat_i is a pattern of type typ_i for each *i*. Thus the pattern `(n:int,r:real,s:string)` is of type `int*real*string`, as might be expected.

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 by the compiler. Thus the following bindings are well-typed (given the bindings above):

```
val (m:int, n:int) = pair
val (m:int, r:real, s:string) = triple
val ((m:int,n:int), (r:real, s:real)) = pair_of_pairs
val (m:int, n:int, r:real, s:real) = quadruple
```

In contrast, the following are ill-typed:

```
val (m:int,n:int,r:real,s:real) = pair_of_pairs
val (m:int, r:real) = pair
val (m:int, r:real) = triple
```

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). Then, we proceed according to the rules of *pattern matching* to determine the bindings for the individual variables in the pattern. This process is quite intuitive. For example, the binding

```
val (m:int,r:real,s:string) = triple
```

binds `m` to `2`, `r` to `2.0`, and `s` to `"2.0"`.

Formally, we go through a process of reduction to atomic value bindings, where an atomic binding is one whose pattern is a variable pattern. The binding

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

reduces to the sequence of bindings

```
val pat1 = val1
...
val patn = valn
```

This decomposition is repeated until all bindings are atomic, at which point the process terminates having arrived at the value environment determined by the original binding. Notice that we rely on the fact that values of *n*-tuple type are *n*-tuples! This is a crucial property of the type system of ML, which determines the shapes of well-typed values based on their types.

For example, the evaluation of the binding

```
val ((m:int,n:int), (r:real, s:real)) = pair_of_pairs
```

proceeds by first evaluating the expression `pair_of_pairs` to `((2,3),(2.0,3.0))`, then decomposing the pattern `((m:int,n:int), (r:real, s:real))` in two major stages, as follows:

1. Reduce the binding

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

   to the sequence of bindings

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

2. Reduce the latter bindings to the sequence of atomic bindings

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

At this point we have determined the bindings for the individual variables in the pattern.

The *null tuple* is a tuple with zero elements. It is written `()`, which is consistent with the *n*-tuple notation. Its type, however, is written `unit`, indicating that it is has but a single element. The null-tuple pattern is, of course, also written `()`. Aside from regularity, the main reason for having a null tuple in the language is to provide a "default" value for expressions that have no interesting value (but, presumably, an interesting effect). We'll have more to say about this later in these notes.

When tuples get large, it gets hard to remember which position is which. *Records* are tuples whose components are *labeled* with an identifier. A *record type* has the form

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

where *n* is at least 2. 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

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

This pattern has type $\{lab_1:typ_1, \ldots, lab_n:typ_n\}$ provided that pat_i has type typ_i for each *i*. The important thing to note about record expressions is that *the order of the fields determines the order of evaluation*, but that for record values, *the order of the fields is irrelevant.* Once the fields have been evaluated, you can write them in any order you like, but the compiler will adhere to the order you choose to write unevaluated fields.

Some examples will help clarify the use of record types.

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

val mailto_rwh : hyperlink =
    { protocol="mailto", address="rwh@cs.cmu.edu",
display="Robert Harper" }
val plcore_home : hyperlink =
    { protocol="http", address="//cs.cmu.edu/~rwh/plcore",
display="Programming Languages Core Course" }

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

(The over-use of strings here is quite obvious; in due course we'll have sufficient mechanism to do a better job.)

In practice one often wishes to select only one or two fields from a tuple or record value, the others being irrelevant to the computation at hand. It would be tedious in the extreme to be forced to bind a variable to each of possibly dozens of irrelevant fields, just so that you could access one of them. *Wild card patterns* are used to handle these situations. The basic form of wild card is written as an underscore, _. It is an atomic pattern that does not generate any bindings; wild card bindings are simply eliminated (after evaluation of the right-hand side).

```
val (m:int, _, r:real, _) = quadruple
val (_, (x:real, y:real)) = pair_of_pairs
val { protocol=port, address=_, display=_ } = mailto_rwh
```

In each case we have elided certain fields using the wild card pattern. The matching process proceeds as before, including *evaluation of the right-hand side of the binding*, but bindings whose pattern is the wild card are dropped. For example, the first binding above generates in one step the bindings

```
val m:int = 2
```

```
val _ = 3
val r:real = 2.0
val _ = 3.0
```

At the next step the bindings for the wild card are dropped, yielding bindings for `m` and `r` alone.

It is important to remember that the right-hand side of a binding is *always* evaluated, regardless of the use of wild card patterns! Thus a binding of the form `val _ =` *exp* always leads to the evaluation of *exp*, but then its value is thrown away. (This could be useful when *exp* has an effect, as we'll see much later in these notes.)

You will by now have asked yourself "what is the type of a wild card pattern?". Good question. The answer is: *whatever type is necessary to ensure that the overall binding is well-typed.* This is undoubtedly not a fully satisfying answer, because it doesn't tell you how this information is determined. We will have more to say on this when we discuss type inference below.

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 = port, ... } = plcore_home
```

The pattern `{ protocol = port, ... }` stands for the pattern `{ protocol=port, address=_, display=_ }` used earlier. In effect the compiler replaces the ellipsis with however many wild card entries are required in order to complete the record pattern. 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 eschew the use of ellipsis.

Sample Code for this Chapter

[Back][Home][Up][Next]

Clausal Function Definitions

Last edit: *Monday, April 27, 1998 02:54 PM*

Sample Code for this Chapter

A function may bind more than one argument by using a pattern, rather than a variable, in the argument position. Function expressions may 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 (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 labeled `url`). 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 record type $typ_1*...*typ_n$, and each *i* between 1 and *n*, there is a function `#i` of type $typ_1*...*typ_n$`->`typ_i defined as follows:

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

where `x` occurs in the *i*th 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.

Tuple types have the property that all values of that type have the same shape; 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`; it is a compile-time type error to attempt to do otherwise.

Other types have values of more than one "shape"; 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*. For the time being we will think of match failure as a fatal

run-time error, but later on we will see that the extent of the failure can be controlled.

Here are some simple 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.

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 definition.* The general form of a function is

`fn` pat_1 => exp_1 | ... | pat_n => exp_n

where 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 *rule*; the entire assembly of rules is called a *match*.

The typing rules for matches ensure consistency of the clauses. Specifically,

1. Each pattern in the match must have the same type *typ*.
2. Each expression in the match must have the same type *typ'*, given the types of the variables in the patterns.

The type of a function whose body is a match satisfying these requirements is *typ->typ'*. Note that there is no requirement that *typ* and *typ'* coincide!

Application of functions with multiple clauses to a value *val* proceeds by considering each clause *in the order written.* At stage *i* 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 replaced by the values determined by the pattern matching process. 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. Here's an example.

```
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`.)

Using `fun` notation we may define `recip` as follows:

```
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 an arrow.

Heterogeneous types abound. Perhaps the must fundamental one is the type `bool` of booleans. Its values are `true` and `false`. Functions may be defined on booleans using clausal definitions that dispatch on `true` and `false`. For example, the negation function is defined clausally as follows:

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

In fact, this function is pre-defined in ML.

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

`case` *exp* `of` pat_1 => exp_1 | ... | pat_n => exp_n

is short for the application

(`fn` pat_1 => exp_1 | ... | pat_n => exp_n) *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.

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.

Conceptually, equality and comparison operations on the types `int`, `char`, and `string` are defined by infinite (or, at any rate, enormously large) matches, but in practice they are built into the language

as primitives. (The ordering on `char` and `string` are the lexicographic orderings.) Thus we may write

```
fun is_alpha c:char =
    (#"a" <= c andalso c <= #"z") orelse (#"A" <= c andalso
c <= #"Z")
```

to test for alphabetic characters.

All this talk of success and failure of pattern matching brings up the issue of *exhaustiveness* and *redundancy* in a match. A clause in a match is *redundant* if any value matching its pattern must have matched the pattern of a preceding clause in the match. A redundant rule can never be reached during execution. It is *always* an error to have a redundant clause in a match. Redundant clauses often arise accidentally. For example, the second clause of the following function definition is redundant for annoyingly subtle reasons:

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

The mistake is to have capitalized `True` so that it is no longer the boolean-typed constant pattern, but is rather a variable that matches any value of Boolean type. Hence the second clause is redundant. Reversing the order of clauses can also lead to redundancy, as in the following mistaken definition of `recip`:

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

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

A match (as a whole) is *exhaustive* if every possible value of the domain type of the match must match some clause of that match. In other words, pattern matching against an exhaustive pattern cannot fail at run-time. The clauses in the (original) definition of `recip` are exhaustive because they cover every possible integer value. The match comprising the body of the following function is not exhaustive:

```
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.

It is *often*, but not always, an error to have an inexhaustive match. The reason is that the type system

cannot record many invariants (such as the property that `is_numeric` is only called with a character representing a decimal digit), and hence the compiler will issue a warning about inexhaustive matches. It is a good idea to check each such warning to ensure that you have not accidentally omitted a clause from the match.

Any match can be made exhaustive by the inclusion of a *catch-all* clause of the form

```
_ => exp
```

where *exp* is an expression of the appropriate type. It is a bad idea to simply stick such a clause at the end of every match in order to eliminate "inexhaustive pattern" warnings. By doing so you give up the possibility that the compiler may warn you of a legitimate error (due to a forgotten case) in your program. The compiler is your friend! Use it to your advantage!

Sample Code for this Chapter

[Back][Home][Up][Next]

Recursive Functions

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:56 PM*

Sample Code for this Chapter

It's time to return to function definitions. So far we've only considered very simple functions (such as the reciprocal function) whose value is computed more or less directly using the primitives of the language. You may well be wondering at this stage how to define functions that require some form of iteration to compute. In familiar imperative languages iteration is accomplished using `while` and `for` loops; in ML it is accomplished using *recursion*.

Informally, a function defined by recursion is one that computes the result of a call by "calling itself". To accomplish this, the function must be given a name by which it can refer to itself. This is achieved using a *recursive value binding.* Recursive value bindings have almost the same form as ordinary, non-recursive value bindings, except that the binding is qualified with the adjective "`rec`" by writing `val rec` *pat* `=` *exp*. Here's an example:

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

This is a recursive definition of the factorial function, which is ordinarily defined in textbooks by the recursion equations

0! = 1
n! = n(n-1)! (n>=0)*

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 clearly a close correspondence between the ML notation and the mathematical notation for the factorial function.

How are recursive value bindings type-checked? The answer may appear, at first reading, to be paradoxical: *assume* that the function has the type specified, then *check* that the definition is consistent with this assumption. In the case of `factorial` we *assume* that `factorial` has type `int->int`, then *check* that its definition

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

has type `int->int`. To do so we must check that each pattern has type `int`, and that each corresponding 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`. (Be certain that you understand this reasoning! It is essential for what follows.)

How are applications of recursive value bindings evaluated? The rules are almost the same as before. We need only observe that the binding for the function may have to be retrieved many times during evaluation (once for each recursive call). For example, to evaluate `factorial 3`, we retrieve the definition of `factorial`, then pattern match the argument against the pattern of each clause. Clearly `3` does not match `0`, but it does match `n:int`, binding `n` to `3` in the process. We then evaluate `n * factorial (n-1)` relative to this binding for `n`. To do so we retrieve the binding for `factorial` a second time, and to apply it to `2`. Once again we consider each clause in turn, failing to match `0`, but succeeding to match `n:int`. This introduces a *new* binding for `n` that shadows the previous binding so that `n` now evaluates to `2`. We then proceed once again to evaluate `n * factorial (n-1)`, this time with `n` bound to `2`. Once again we retrieve the binding for `factorial`, then bind `n` to `1`, shadowing the two previous bindings, then evaluating `n * factorial (n-1)` with this binding for `n`. We retrieve the binding for `factorial` one last time, then apply it to `0`. This time we match the pattern `0` and yield `1`. We then (in four steps) compute the result, `6`, by completing the pending multiplications.

The `factorial` function illustrates an important point about recursive function definitions. Notice that the recursive call in the definition of `factorial` occurs as the argument of a multiplication. This means that in order for the multiplication to complete, we must first complete the calculation of the recursive call to `factorial`. In rough outline the computation of `factorial 3` 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`

(The strings of multiplications are implicitly right-associated.) 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.

The foregoing definition of `factorial` should be contrasted with the following definition:

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

fun factorial n:int = fact_helper (n, 1)
```

We define `factorial` using a *helper function* `fact_helper` that takes an additional parameter, called an *accumulator*, that records the running partial result of the computation. This corresponds to reducing the prefix of the pending computations in the trace given above by "left-associating" the multiplications. (In fact the technique is only applicable to associative binary operations for precisely this reason.)

The important thing to observe about `fact_helper` is that it is *tail recursive*, meaning that the recursive call is the last step of evaluation of an application of it to an argument. The following evaluation trace illustrates the point:

1. `factorial 3`
2. `fact_helper (3, 1)`
3. `fact_helper (2, 3)`
4. `fact_helper (1, 6)`
5. `fact_helper (0, 6)`
6. `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. In this sense tail recursive definitions are analogous to loops in imperative languages: they merely iterate a computation, and do not require allocation of storage during execution. For this reason tail recursive procedures are sometimes called *iterative*.

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. A *specification* of the result of the function stated in terms of its arguments. This specification will usually involve *assumptions* about the arguments that are sufficient to establish that the function behaves correctly.
2. An *induction principle* that justifies the correctness of the recursive function based on the pattern of its recursive calls. In simple cases this is ordinary mathematical induction, but in more complicated situations a more sophisticated approach is often required.

These ideas may be illustrated by considering the first definition of `factorial` given above. A reasonable specification for `factorial` is as follows:

> *if n>=0 then* `factorial` *n evaluates to n!*

Notice that the specification expresses the assumption that the argument, *n*, is non-negative, and asserts that the application of `factorial` to *n* terminates with the expected answer.

To check that satisfies this specification, 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!*, and so by the definition

`factorial` *n* evaluates to the value of *n*m! = (m+1)*m! = (m+1)! = n!*, as required. This completes the proof.

That was easy. What about the second definition of `factorial`? We focus on the behavior of `fact_helper`. A suitable specification is

> *if n>=0 then* `fact_helper` *(n,r) evaluates to n!*r*

Once again we proceed by mathematical induction on *n*; you can easily check that `fact_helper` satisfies this specification. It follows that the second definition of `factorial` in terms of `fact_helper` satisfies the specification of `factorial` given above, since we may take *r=1* in the specification of `fact_helper`.

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

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

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

```
(* for n>=0, fib n evaluates to 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 show that the running time`fib` of is exponential in its argument, which is clearly awful for such a simple function.

Here's a better solution: for each *n>=0* compute not only the *n*th Fibonacci number, but also the *(n-1)* st as well. (For *n=0* we define the "-1"st 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 fibb 0 = (1, 0)
  | fibb 1 = (1, 1)
  | fibb (n:int) =
    let
        val (a:int, b:int) = fibb (n-1)
    in
        (a+b, a)
    end
```

You might feel satisfied with this solution since it runs in time linear in *n*. But in fact there's a *constant-time* algorithm to compute the *n*th Fibonacci number! In other words the recurrence

$F_0 = 1$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

has a closed-form solution. (See Knuth's *Concrete Mathematics* (Addison-Wesley 1989) for a derivation.) 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.

It is often useful to define two functions simultaneously, each of which calls itself and/or the other to compute its result. Such fnctions 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. Later in these notes we will see more compelling examples of mutually-recursive functions.

Sample Code for this Chapter

[Back][Home][Up][Next]

Type Inference

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:56 PM*

Sample Code for this Chapter

So far (with a few exceptions) we've programmed in what is known as an*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've no doubt 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 contextual information.

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 are allowed to write just `fn s => s ^ "\n"`, leaving ML to insert "`:string`" for you. When is it allowable to omit this information? It is difficult to give a precise answer without introducing quite a lot of machinery, but we can give some hints of when you can and when you cannot omit types. A remark fact about ML is that the answer is "almost always", with the exception of a few trouble spots that we now discuss.

The main difficulty is with the arithmetic operators, which are *overloaded*, by which we mean that the same syntax is used for integer and floating point arithmetic operations. This creates a problem for type inference because it is not possible to unambiguously reconstruct type information for a function such as `fn n => n+n` because there is no way to tell whether the addition operation is integer or floating point addition. We could equally well regard this expression as abbreviating `fn n:int => n+n`, with type `int->int`, or `fn n:real => n+n`, with type `real->real`. In some cases the surrounding context determines which is meant. For example, in the expression `(fn n => n+n)(0)` the only sensible interpretation is to regard the parameter `n` to have type `int`. A related source of difficulty is the (infrequently used) "sharp" notation for records. Absent information from the context, the type of the function `fn r => #name(r)` cannot be determined: all that is known of the type of `r` is that it has a `name` field; neither the type of that field nor the labels and types of the remaining fields are determined. Therefore this function will be rejected as ambiguous because there is not enough information to determine the domain type of the function.

These examples illustrate situations where ambiguity leads to difficulties. But you shouldn't conclude from this that type inference must fail unless the missing type information can be uniquely determined! In many (indeed, most) cases there is no *unique* way to infer omitted type information, but there is nevertheless a *best* way. (The examples in the preceding paragraph merely serve to point out that sometimes there is no best way, either. But these are the exceptions, rather than the rule.)

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 result without performing any computation on it. You might suspect that this expression has to be rejected since its type is ambiguous, but this would be unfortunate since the expression makes perfectly good sense for *any* choice of the type of `x`. This is in sharp contrast to examples such as the function `fn x=>x+1`, for which only *two* choices for the type of `x` are possible (namely, `int` and `real`), with no way to choose between them. The choice of `int` or `real` affects the behavior of the function: in one case it performs an integer addition, in the other a floating point addition. In the case of the identity function, however, we may choose any type at all for `x`, without affecting the execution behavior of the function --- the function is said to be *polymorphic* because its execution behavior is uniform in the type of `x`. 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` ("alpha"), `'b` ("beta"), `'c` ("gamma"), *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. 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 capture 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`, an hence may be chosen arbitrarily. 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->'a`. In each of these cases 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):'a=>x` with type `'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. Notice as well that 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 since the function returns its first parameter when invoked. The type scheme 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 those pesky examples involving arithmetic primitives or record selection operations) *has a principal type scheme*. That is, there is always (well, with very few exceptions) 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)`, then 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 of use selects 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 checking error.
2. *Underconstrained*: there are many solutions. There are two sub-cases: *ambiguous* (due to overloading), or *polymorphic* (there is a "best" solution).
3. *Uniquely determined:* there is precisely one solution. This corresponds to an unambiguous type inference problem.

The free type variables of the system of equations determines the "degree" of polymorphism in the expression: the constraints have a solution for any choice of types to substitute for these variables.

The characterization of type inference as a constraint satisfaction procedure suggests an explanation 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 considers the constraints in some order, and at some point discovers an inconsistency. It is fundamentally impossible to attribute this inconsistency to any one feature of the program: all that is know is that the constraint set as a whole is unsatisfiable. 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.

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.

We finish this section with a technical issue that arises from time to time. As we remarked above, the treatment of `val` bindings ensures that a bound variable has precisely the same types 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, 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 declaration

```
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 this case. This particular example is not very impressive since it's hard to imagine using the former, rather than the latter, declaration in a practical situation, but occasionally something similar does arise, with an attendant loss of polymorphism.

Why this limitation? 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.

Sample Code for this Chapter

[Back][Home][Up][Next]

Lists

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

We have already noted that aggregate data structures are especially easy to handle in ML. Our first examples were tuple and record types. The *list* types provide another example of an aggregate data structure in ML. Informally, the values of type *typ* `list` are the finite lists of values of type *typ*. But what is a list? The values of type *typ* `list` are defined 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 argument *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`:

```
nil : typ list
op :: : typ * typ list -> typ list
```

(The notation `op ::` is used to *refer* to the "cons" operator as a function, rather than to *use* it to form a list, which requires infix notation.) Two things are notable here:

1. The "cons" operation takes an argument of type *typ* `list`, and yields a result of type *typ* `list`. This reflects the "recursive" nature of the type *typ* `list`.
2. Both operations 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.

A consequence of the inductive definition of the list type is that values of type *typ* `list` have the form

h_1 `::(` h_2 `::` ... `::(` h_n `::nil)`...`)`

for some *n>=0*. (When *n* is zero, this is, by convention, the empty list, `nil`.) The operator `::` is *right-associative*, so we may omit the parentheses and just write

h_1 `::` h_2 `::` ... `::` h_n `::nil`.

As a further convenience this list may be abbreviated using *list notation*:

`[` h_1 `,` h_2 `,` ... `,` h_n `]`

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.

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 `length` of 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]
```

It is not tail recursive. In fact, its time complexity 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)* of on a list of length *n*.

T(0) = O(1)
T(n+1) = T(n) + O(1)

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 rev_helper (nil, a) = a
        | rev_helper (h::t, a) = rev_helper (t, h::a)
in
      fun rev l = rev_helper (l, nil)
end
```

The pattern is the same as before, except that by re-associating the uses of `::` we reverse the list! The helper function reverses its first argument and prepends it to its second argument. That is, `rev_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 `rev_helper` runs in time proportional to the length of its first argument, and hence `rev` runs in time proportional to the length of the list.

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

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

The proof is by structural induction on the list *l*. If *l* is `nil`, then `rev_helper (`*l*`, `*a*`)` evaluates to *a*, which is as required. If *l* is *h*`::`*t*, then by inductive hypothesis evaluates to the result of appending *h*`::`*a* to the reversal of *t*, which is easily seen to be the result of appending *a* to the reversal of *h*`::`*t*.

The form of this argument may be summarized as follows:

1. Establish the correctness of the function for the empty list, `nil`.
2. Assuming the correctness of the function for *t*, establish its correctness for *h*`::`*t*.

It follows that the function is correct for all lists *l*, by the inductive definition of the list type. This is called the principle of *structural induction on lists*. We will soon generalize this to other inductively-defined types.

Sample Code for this Chapter

[Back][Home][Up][Next]

Datatype Declarations

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

Lists are one example of the notion of a *recursive datatype*. ML provides a general mechanism, the `datatype` declaration, for introducing recursive types. Earlier we introduced the `type` declarations as an abbreviation mechanism. Giving a type a name is useful documentation and is convenient as an abbreviation, but is otherwise inconsequential. One could replace all uses of the type name by its definition and not effect 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, nor may not, be (mutually) recursive.
2. One or more "new" value constructors for each of the type constructors introduced by the declaration.

The type constructor 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.

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 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
```

which determines whether or not the first `suit` outranks the second.

Datatypes may also be *parameterized* by another type. For example,

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

introduces the unary type constructor `'a option`. 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`.

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 directly 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. Our definition of binary trees is analogous to starting the natural numbers with zero, rather than one. In fact you 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 one ordinarily defines recursive functions. 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 called *structural induction*. The function `height` is said to be defined by induction on the structure of its argument, 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. Suppose we gave the following definition of `size`:

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

What happens? 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` datatype is appropriate for binary trees: those for which every 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 (try it) 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 a variadic tree to be either empty, or a node collecting together a forest to form a tree, and a forest to be 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 = Nil | Cons of 'a tree * 'a forest
```

This example illustrates the introduction of two *mutually recursive datatypes*, which is why we present it here. Mutually recursive datatypes beget mutually recursive functions defined on them. 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 Nil = 0
  | size_forest (Cons (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 datatype declaration for such trees:

```
datatype ’a tree = Empty | Node of ’a branch * ’a branch
```

```
and      'a branch = Branch of 'a * 'a tree
```

Notice that in contrast to our first definition of binary trees in which the branches from a node to its children were *implicit*, these branches are now *explicit* since they are labelled with data items. For example, we can collect up 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)
```

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 it makes no sense to evaluate the expression

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

since the result, if it were to be accepted, would be a "heterogeneous" tree with integers at some nodes and strings at others. Such structures are ruled out in ML as type-incorrect.

In 95% of the cases this apparent restriction is no restriction at all; 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 what about the other 5%? What if one really wanted to have a tree with integers at some nodes and strings at others? How would one represent such a thing in ML? To see 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. This 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!

`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 perhaps 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 * exp | 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. This is yet another reason why the static type discipline of ML is a positive benefit, rather than a hindrance, to programmers.

Sample Code for this Chapter

[Back][Home][Up][Next]

Functionals

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

Functions (values of function type) are *first-class values*, 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 sometimes 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 [*0,x*], viewed as a function of *x*.

Higher-order functions are less familiar tools in programming since most well-known languages have at best 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.* Design patterns are just higher-order functions 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.

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 x is considered to be a reference to the *nearest lexically enclosing declaration of* x. We say "nearest" because of the possibility of shadowing; if we re-declare a variable x, then subsequent uses of x 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 dynamic 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.

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 discussion illustrates why functions in ML are *not* directly analogous to "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 expressive 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 (standard) 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))`. 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
```

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_em nil = 0
  | add_em (h::t) = h + add_em t

fun mul_em nil = 1
  | mul_em (h::t) = h * mul_em t
```

What precisely is the similarity? We will look at it from two points of view. One 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_em` and `mul_em` as follows:

```
fun add_em l = reduce (0, op +, l)
fun mul_em 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 perspective on the commonality between `add_em` and `mul_em` 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_em` and `mul_em` is abstracted by the `reduce` functional, which is then specialized to yield `add_em` and `mul_em`. Said another way, *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, tail-recursive 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.

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 present case 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 precisely the same (apart from one additional function call in the latter case).

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 I'm afraid it is necessary to avoid 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 :: 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 `[v1,...,vn]`, the function `staged_append` yields a function that is equivalent to, but not quite as efficient as, the function

```
fn l => v1 :: v2 :: ... :: vn :: 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.

Sample Code for this Chapter

[Back][Home][Up][Next]

Exceptions

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

In the first chapter of these notes we mentioned that expressions in Standard ML always have a type, may have a value, and may engender 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. What are some other examples? The main examples are these:

1. Exceptions. Evaluation may be aborted by signaling an exceptional condition.
2. Mutation. Storage may be allocated and modified during evaluation.
3. I/O. 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 dealt with later in these notes.

A basic use of exceptions in ML is to signal error conditions. 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. It can be proved that ML is safe, from which it follows that such behaviors cannot arise (except through the failure of the compiler to implement the language properly.) In unsafe languages (such as C) these sorts of errors can and do arise, typically because of the (mis)use of a primitive operation on a value that does not lie in its domain of definition. For example, in C we may cast an integer as a function pointer, then invoke it by applying it to an argument. The behavior of such a program that cannot be predicted at the level of the language itself since it relies on the details of the memory layout and the interpretation of data as code. To ensure safety, and hence freedom from mysterious run-time faults, ML ensures that the primitive operations may only be applied to appropriate arguments. This is achieved in part by the static type discipline, which rules out expressions that are manifestly inappropriate (*e.g.*, adding a string to an integer or casting an integer as a function), and partly 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.*

For example, 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
```

Thus an exception is a form of "answer" to the question "what is the value 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 from one another. 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.

At this point 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 OK 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 mistake by the programmer, evidently), the result is nevertheless well-defined because ML checks for pattern match failure, rather than leaving the behavior of the application undefined. In other words: 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.

These are all examples of the use of pre-defined exceptions to 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.

Here's an example. 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 ways. 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 as follows:

```
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.

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 = makestring (checked_factorial input)
    in
        print result
    end
    handle Factorial => print "Out of range.\n"
```

The expression *exp* `handle` *match* is 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 *exn*, 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 *exn*, then evaluation resumes with the expression part of that clause. If no pattern matches, the exception *exn* 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 *exn*.

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 = makestring (checked_factorial input)
        val _ = print result
    in
        factorial_driver ()
    end
    handle EndOfFile => print "Done.\n"
         | SyntaxError =>
           let val _ = print "Syntax error.\n" in
factorial_driver () end
         | Factorial =>
           let val _ = print "Out of range.\n" 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 that they *force* you to consider the exceptional case (if you don't, you'll get an uncaught exception at run-time), and that they *allow* you to segregate the special case from the normal case in the code (rather than clutter the code with explicit checks).

Another typical use of exceptions is to implement *backtracking*, a programming technique based on exhaustive search of a state space. A very simple, *albeit* 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.

Exceptions can also carry values. For example, we might associate with a `SyntaxError` exception a string indicating the precise nature of the error. For example, we might write

```
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. Such an exception is introduced by the declaration

```
exception SyntaxError of string
```

which 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
^ "\n"
```

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.

Exception constructors raise the question of the status of exceptions in the language. 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 an exception. 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.*

Sample Code for this Chapter

[Back][Home][Up][Next]

References

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:56 PM*

Sample Code for this Chapter

Evaluation of an expression may terminate with a value and may along the way engender an effect upon its environment. Our first example of an effect was the possibility of raising an exception, which we explored in detail in the preceding chapter. The next important example of an effect is 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, storage effects are sometimes denigrated by referring to them as *side effects*, by analogy with the unintended effects of some medications. While it is surely excessive to dismiss storage effects as completely undesirable, it is advantageous to minimize the use of storage effects except where clearly appropriate to the task. We will explore some of the basic techniques for using storage effects later in this chapter, but first we introduce the mechanisms for supporting mutable storage in ML.

To support mutable storage the execution model of programs is modified to include an implicit *memory* consisting of a finite set of *mutable cells* containing data items of a fixed type. A mutable cell may be thought of as a kind of container in which a data value is stored. During the course of evaluation the content of a cell may be retrieved or may be replaced by any other value of the same type. Mutation introduces a strongly temporal aspect to evaluation: we speak of the *current* contents of a cell as the value *most recently* assigned to it. This is to be contrasted with the bindings of values to variables, which never change once made and hence have a permanent quality; the binding of a variable is a uniquely-determined value that does not change during evaluation. Since cells are used by issuing "commands" to modify and retrieve their contents, programming with cells is sometimes called *imperative programming*.

Since cells may have their contents changed during evaluation it is imperative that we take careful account of the *identity* of cells. When are two cells the same? When are they different? The guiding principle is that two cells (of the same type) are distinct if there is a program that can tell them apart; otherwise they are equal. How can we tell cells apart? By doing the only things we can ever do with cells: retrieve their contents or set their contents to specified values. Given two integer cells, we can determine whether they are the same cell or not by first checking if they have distinct contents. If so, then they are distinct cells. If not, we must distinguish between two "copies" of a single cell, or two cells that happen to have the same content. To do this, bind the current contents of one cell to a variable, and set that cell's value to an integer different from the saved contents. If the other cell's value is now the newly-assigned value, then the two cells are the same, otherwise they are different.

This principle of equality is called *identity of indiscernables*: two things are equal if we cannot tell them apart. The test we just outlined extends to cells of other types, but is a rather roundabout way to test for cell identity. In practice we work with a slightly conservative approximation to cell identity, called *reference (or pointer) equality* --- two cells are equal iff they occupy the same address in

memory. This test is conservative in that it may distinguish two cells that are in fact indiscernable: any two unit-valued cells are indiscernable because there is only one value of unit type, yet pointer equality would distinguish them. To avoid such anomalies we use pointer equality to determine cell identity, relying on the representation of cells as references to memory. For this reason mutable cells in ML are called *reference cells*, or *references*.

Reference cells containing values of type *typ* are themselves values of type *typ* `ref`. They are "first-class" values in the sense that reference cells may be passed as arguments, returned as results, and even stored in other reference cells. Reference cells are *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 a "new cell" we mean a cell that is distinct from all other cells previously allocated; it does not share storage with any of them. The *content* of a cell of type *typ* is retrieved using the function `!` of type *typ* `ref ->` *typ*. Applying `!` to a (reference to a) cell returns the current content of that cell. The content of a cell is modified by the operation `op :=` of type *typ* `*` *typ* `ref -> unit`; it is written using infix syntax with the reference cell as left-hand argument and the new contents as right-hand argument. 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. Cells may be compared for equality using the equality operation, `=`, which has type *typ* `ref *` *typ* `ref -> bool`.

Here are some examples:

```
val r = ref 0
val s = ref 0
val a = r=s
val _ = r := 3
val x = !s + !r
val t = r
val b = s=t
val c = r=t
val _ = t := 5
val y = !s + !r
val z = !t + !r
```

Afterwards, `a` is bound to `false`, `b` to `false`, `c` to `true`, `x` to `3`, `y` to `5`, and `z` to `10`. Be sure you understand exactly why in each case!

The above examples illustrate the problem of *aliasing*. The variables `t` and `r` are both bound to the *same* cell, whereas `s` is bound to a *different* cell. We say that `t` and `r` are *aliases* for the same cell because the one cell is known by two different names. Aliasing is a serious source of bugs in programs since assigning a value to one destroys the contents of the other. Avoiding these kinds of problems requires careful reasoning about the possibility of two variables being bound to the same reference cell. A classic example is a program to "rotate" the contents of three cells: given reference cells *a*, *b*, and *c*, with initial contents *x*, *y*, and *z*, set their contents to *y*, *z*, and *x*, respectively. Here's a candidate implementation:

```
fun rot3 (a, b, c) =
    let
        val t = !a
    in
```

```
        a := !b; b := !c; c := t
    end
```

This code works fine if *a*, *b*, and *c* are distinct reference cells. But suppose that *a* and *c* are the same cell. Afterwards the contents of *a*, *b*, and *c* are *y*, *y*, and *x*! A correct implementation must work even in the presence of aliasing. Here's a solution that works correctly in all cases:

```
fun rot3 (a, b, c) =
    let
        val (x, y, z) = (!a, !b, !c)
    in
        a := y; b := z; c := x
    end
```

Notice that we use immutable variables to temporarily hold the initial contents of the cells while their values are being updated.

This example illustrates the use of the semicolon to sequence evaluation of expressions purely for their effect. The expression

exp$_1$ `;` *exp*$_2$

is shorthand for

`let val _ =` *exp*$_1$ `in` *exp*$_2$ `end`

The expression *exp*$_1$ is evaluated only for its effect; its return value is thrown away by the wildcard binding. The value of the entire expression is the value of *exp*$_2$ after evaluation of *exp*$_1$ for effect. The cumulative effect of the sequential composition is the effect of evaluating *exp*$_1$ followed by the effect of evaluating *exp*$_2$.

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 or Pascal 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`; in Pascal one must use reference parameters to achieve a similar effect. Which is preferable is largely 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 relatively seldom used in ML, 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. For example, the second implementation of `rot3` above might be written using ref

patterns as follows:

```
fun rot3 (a, b, c) =
    let
        val (ref x, ref y, ref z) = (a, b, c)
    in
        a := y; b := z; c := x
    end
```

In practice it is common to use both explicit de-referencing and ref patterns, depending on the situation.

Using references it is possible to mimic the style of programming used in imperative languages such as C or C++ or Java. For example, we might define the factorial function 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 since its argument is always the null-tuple.

It is 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; quite the opposite is the case. We will now discuss some important uses of state in ML.

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.

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 principle 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 principle 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 *pcls*. 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 = Nil | Cons of 'a * 'a pcl ref
```

The "tail" of a `Cons` node is a reference cell so that we may assign to it to implement backpatching. Here's an example:

```
fun hd (Cons (h, _)) = h          (* auxiliary functions *)
fun tl (Cons (_, t)) = t

val ones = Cons (1, ref Nil)     (* create a preliminary
acyclic structure *)

val _ = (tl ones) := ones        (* backpatch to form the
cycle *)
```

Initially the variable `ones` is bound to the acyclic pcl with one node whose head element is `1`. We then assign that very node to the predecessor (tail) of that node, resulting in a circular pcl with one node. Observe that `hd ones`, `hd !(tl ones)`, `hd !(tl !(tl ones))`, *etc* all evaluate to `1`. Notice that we must explicitly refer to the contents of the tail of each node since it is a reference cell!

Let us define the *length* of a pcl to be the number of distinct nodes occurring in it. An interesting exercise is to define a `length` function for pcls that makes *no* use of auxiliary storage (*i.e.*, no list of previously-encountered nodes) and runs in time proportional to the number of cells in the pcl. *Hint:* think of the fable of the tortoise and the hare. If they run a long race on an oval track, what is sure to happen, and when? Does this suggest an algorithm?

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 values of type *typ*. The basic operations on arrays are these:

`array : int * 'a -> 'a array`	*create array of given size with given initial value*
`size : 'a array -> int`	*number of elements in a given array*
`sub : 'a array * int -> 'a`	*access element; raises Subscript exception if out of bounds access is attempted*
`update : 'a array * int * 'a -> unit`	*change the contents of a given array element; raises Subscript for out of bounds access*

These are just the basic operations on arrays; consult the Basis Library document for further details. Immutable arrays are also available. The type `'a vector` is similar to the type `'a array`, except that vectors are immutable, whereas arrays are mutable.

One simple use of arrays is for *memoization*. Here's a function to compute the *n*th 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 *f 0* + ... + *f n*.)

```
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`, `C2`, ..., `C 9`, and the computation of `C i` engenders the computation of `C 1`, ..., `C (i-1)` for each 1<=*i*<=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.

Sample Code for this Chapter

[Back][Home][Up][Next]

Input & Output

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

Standard ML Basis Library 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 the IO section of the *Standard ML Basis Library* for more details. There is no standard library for graphical user interfaces; each implementation provides its own package. See your vendor's documentation for details.

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*, `canInput` determines whether or not a call to `input` would immediately yield up to *n* characters. If the `input` operation would block, `canInput` yields `NONE`; otherwise it yields `SOME` *k*, with 0<=*k*<=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*.

Sample Code for this Chapter

[Back][Home][Up][Next]

Lazy Data Structures

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:55 PM*

Sample Code for this Chapter

As we saw earlier, a `datatype` declaration is used to introduce a new type whose elements are generated by a given set of value constructors. The value constructors may be used to create values of the type (by applying them to values of suitable type), and to decompose values of the type (by using them in patterns). Value constructors, like all other functions in ML, are evaluated *eagerly*, meaning that the arguments to the constructor are evaluated before the constructor is applied. For example, to attach an element to the front of a list, we first determine the value of the element and the value of the list before building a new list with that element as head and that list as tail. This policy is based on the intuitively appealing idea of a list as a kind of value that we manipulate by using the list constructors as functions and as patterns.

An alternative is to view a data structure as being perpetually in the process of creation, rather than as a result of a completed computation. According to this view a list may be thought of as a "partial", or "suspended", computation that, when provoked, computes just far enough to determine whether the end of the list has been reached, or , if not, to produce the next element of the list together with a suspended computation to compute the remainder of the list. An added benefit of this viewpoint is that it is then possible to define *infinite lists* (better known as *streams*) that continually generate the next element, without ever reaching the end of the list. This view of data structures as being in the process of creation conflicts with the eager evaluation strategy just described since under the eager approach all expressions are fully evaluated before they are used, whereas we would like to evaluate them only as much as absolutely necessary to allow the overall computation to proceed. This is called, appropriately enough, *lazy evaluation*.

Standard ML does not support lazy evaluation as a primitive notion; it can be implemented "by hand" using methods that are described later in these notes. However, Standard ML of New Jersey (from version 110.5) does provide for lazy evaluation through an extension of the `datatype` and `val rec` declaration forms. We will illustrate these mechanisms by defining a type `'a stream` of streams of values of type `'a`. Based on the discussion above you might imagine that a stream is just an infinite list, but it is important to keep the two concepts separate. Lists are eager types whose values are generated by finitely-many applications of `::` to the empty list, `nil`. Streams are lazy types whose values are determined by suspended computations that generate the next element of the stream (and another computation to generate the remainder). The two concepts are, and ought to be kept separate since they serve different purposes and require different modes of reasoning.

First off, the lazy evaluation mechanisms of SML/NJ must be enabled by evaluating the following declarations:

```
Compiler.Control.Lazy.enabled := true;
open Lazy;
```

We may then define a type of streams as follows:

```
datatype lazy 'a stream = Cons of 'a * 'a stream
```

The keyword "`lazy`" indicates that values of type `'a stream` are suspended computations that, when evaluated, yield a value of the form `Cons (x, c)`, where `x` is a value of type `'a`, and `c` is another value of type `'a stream`, *i.e.*, another computation of such a value.

How might a value of type `'a stream` be created? Since the description of values of this type we've just given is clearly "circular", we must employ a recursive value binding to create one. Here's a definition 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 defining a value of a lazy type, which means that it must be kept as an incomplete computation, rather than fully evaluated at the time the binding is created. What computation is bound to `ones`? It's the computation that, when evaluated, yields `Cons (1, ones)`, a stream whose head element is `1` and whose tail is the very same computation again. Thus if we evaluate the tail of `ones` we will, once again, obtain the same value, and so on *ad infinitum*.

How can we take apart values of stream type? By pattern matching, of course! For example, we may evaluate the binding

```
val Cons (h, t) = ones
```

to extract the head and tail of the stream `ones`. To perform the pattern match we must first force the evaluation of `ones` to obtain `Cons (1, ones)`, then pattern match to bind `h` to `1` and `t` to `ones`. Had the pattern been "deeper", further evaluation would be forced, 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 partial computations up to the depth required by the pattern*.

We may define functions to extract the head and tail of a stream as follows:

```
fun shd (Cons (h, _)) = h

fun stl (Cons (_, s)) = s
```

Both of these functions force the computation of the stream when applied so that they may extract the head and tail elements. In the case of the head element it is clear that the stream computation *must* be

forced in order to determine its value, but a moment's thought reveals that it is *not* strictly necessary to force the computation of a stream to extract it's tail! Why is that? Since the tail of a stream is itself a stream, it may be thought of as a suspended computation. But which suspended computation is it? According to the definition just given, it is the suspended stream computation extracted from the second component of the value of the given stream. But another definition is possible: it is the suspended computation that, *when forced*, yields the second component of the result of forcing the stream computation. Here's a definition:

```
fun lazy lstl (Cons (_, s)) = s
```

Here the keyword "`lazy`" indicates that an application of `lstl` to a stream does *not* immediately perform pattern matching (hence forcing the argument), but rather merely *sets up* a delayed 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 as follows:

```
val rec lazy s = (print "."; Cons (1, s));
val s' = stl s;                              (* prints "."
*)
val Cons _ = s';                             (* silent *)

val rec lazy s = (print "."; Cons (1, s));
val s'' = lstl s;                            (* silent *)
val Cons _ = s'';                            (* prints "."
*)
```

Notice that since `stl` immediately forces it's argument, the "." is printed when it is applied, whereas it is printed only when the result of applying `lstl` to an argument is itself forced by another pattern match.

It is *extremely important* that you understand the difference between these two definitions! To check your understanding, 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
```

Notice that 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 an application of loop to a stream merely sets up a stream computation, rather than forcing the evaluation of its argument at the time that the loop is applied. This ensures that we are as lazy as possible about evaluating streams. 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. This would be a bit over-eager in the case that the result of applying `smap` were never used in a subsequent computation. Which solution is "right"? It all depends on what you're doing, but as a rule of thumb, it is best to be as lazy as possible when dealing with lazy types.

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)
```

It is worthwhile contemplating how and why this definition works.

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. Here's the code:

```
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 filter 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, sfilter (not o
(divides x)) s)
```

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)          (* might as well be
eager *)
val primes = sieve nats2
```

To inspect the values of a stream it is often useful to use the following function that "takes" *n>=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
```

In addition to supporting demand-driven computation the lazy evaluation primitives of SML/NJ also support *memoization* of the results of a computation. The idea is that a delayed computation is performed *at most once*. If it is never forced by pattern matching, then the delayed computation is never performed at all. If it is ever forced, then the result of forcing that computation is stored in a

memo pad so that if it is forced again, the previous result is returned immediately,without repeating the work that was done previously. 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`.

Sample Code for this Chapter

[Back][Home][Up][Next]

Concurrency

[Back][Home][Up]

Last edit: *Monday, April 27, 1998 02:54 PM*

Sample Code for this Chapter

Concurrent ML (CML) is a non-standard extension of Standard ML with primitives for concurrent programming. It is available as part of the *SML/NJ* compiler only. The eXene Library for programming the X windows system is based on CML. The *MLWorks* system also includes primitives for concurrent programming.

Sample Code for this Chapter

[Back][Home][Up]

Module Language

[Back][Home][Next]

Last edit: *Sunday, April 05, 1998 10:45 PM*

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*.

[Signatures and Structures][Views and Data Abstraction][Hierarchies and Parameterization]

[Back][Home][Next]

Signatures and Structures

[Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:57 PM*

Sample Code for this Chapter

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-2, Modula-3, 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. One thing to point out right away, though, is that the relationship between signatures and structures in ML is *many-to-many*, whereas in some languages (such as Modula-2) the relationship is *one-to-one* or *many-to-one*. This means that in ML a signature may serve as the interface for many different structures, and that a structure may implement many different signatures. This provides a considerable degree of flexibility in the use (and re-use) of components in a system. The price we pay for this flexibility is that we must be quite careful about referring to *the* signature of a structure, since it can have more than one. As we will see, every structure has a *most specific*, or *principal*, signature, with the property that all other signatures for that structure are (in a suitable sense) more restrictive than the principal signature.

Structures

The fundamental unit of modularity in ML is the *structure*. A structure consists of a sequence of declarations comprising the *components* of the structure. A structure may be bound to a *structure variable* using a *structure binding*. The components of a structure are accessed using *long identifiers*, or *paths*. A structure may also be *opened* to incorporate all of its components into the environment.

Here's a simple example of a structure:

```
structure IntLT = struct
  type t = int
  val lt = (op <)
  val eq = (op =)
end
```

This structure has three components, one type and two values, each of which are functions. The type component is named `t` and is bound to the type `int`. The value components are named `lt` and `eq`, and are bound to the corresponding comparison operations on integers. This structure packages up the type `int` with the integer comparison operations < and = to form a module that is then bound to the structure variable `IntLT`.

We may similarly package up the type `int` with comparison operations being divisibility and equality using the following binding:

```
structure IntDiv = struct
  type t = int
  fun lt (m, n) = (n mod m = 0)
  fun eq (m, n) = (op =)
end
```

The structures and may be thought of as two different *interpretations* of the type `int` as an ordered type (*i.e.*, a type supporting a "less than" and an equality operation). In one case we interpret "less than" as the standard ordering on integers, in the other we interpret "less than" as divisibility. The point is *the type does not determine the interpretation*. We use the module system to package up types with operations to provide an interpretation of that type. Many different interpretations may co-exist, provided only that we bind them to distinct structure variables.

The components of a structure are accessed using *paths* (also known as *long identifiers* or *qualified names*). We may only access the components of a *named* structure (one that has been bound to a structure variable). A component named *id* of a structure named *strid* is accessed by the long name *strid*`.`*id*, the structure name followed by the component name, separated by a "dot". For example, `IntLT.lt` designates the `lt` operation of the structure `IntLT`, and `IntDiv.lt` designates the `lt` operation of the structure `IntDiv`. The type of `IntLT.lt` is

```
IntLT.t * IntLT.t -> bool,
```

and the type of `IntDiv.lt` is

```
IntDiv.t * IntDiv.t -> bool.
```

The types of these operations have been "externalized" using long identifiers to refer to the appropriate type `t` for each operation. Since `IntLT.t` and `IntDiv.t` are both bound to the type `int`, it makes sense to write expressions such as `IntLt.lt(3,4)` and `IntDiv.lt(3,4)`.

Since `IntLT.t` and `IntDiv.lt` are both bound to the type `int`, it is technically correct to consider `IntLt.t` to be of type

```
IntDiv.t * IntDiv.t -> bool
```

and also of type

```
int * int -> bool.
```

Were we also to have a structure `StringLT` whose `t` component is bound to the type `string`, then `StringLT.lt` would have type

```
StringLT.t * StringLT.t -> bool
```

and type

```
string * string -> bool
```

but not type

```
IntLT.t * IntLT.t -> bool
```

Packaging a declaration to form a structure does not affect the usual rules of type equivalence --- transparent type definitions remain transparent.

The use of a long identifier to access a component of a structure serves to remind us of the interpretation of the underlying type of the structure. For example, the long identifer `IntLT.lt` reminds us that the comparison is the standard "less than" relation on integers, whereas the long identifier `IntDiv.lt` reminds us that the comparison is divisiblity. Sometimes the use of long identifiers can get out of hand, cluttering the program text, rather than clarifying it. This can be alleviated by *opening* the structure for use in a particular context. For example, rather than writing

```
IntDiv.lt (exp1, exp2) andalso IntDiv.eq (exp3, exp4)
```

we may instead write

```
let
  open IntDiv
in
  lt (exp1, exp2) andalso eq (exp3, exp4)
end
```

This has the effect of incorporating the components of the structure `IntLT` into the environment for the duration of the evaluation of the body of the `let` expression. It is as if we replace "`open IntLT`" by the declarations comprising the structure bound to `IntLT`.

Using `open` has some disadvantages. One is that we cannot simultaneously open two structures with have one or more components with the same names --- the one we open later we will shadow the bindings of the one we open earlier. For example, if we write

```
let
  open IntLT IntDiv        (* open both structures in the
order given *)
in
  ...
end
```

then only the bindings of the second structure, `IntDiv`, are available in the scope of the `let` because they completely shadow the bindings of the first structure, `IntLT`.

Another disadvantage is that it is difficult to determine exactly which bindings are introduced by an `open` declaration. We must refer to the implementation of the opened structure (typically defined somewhere remote from the client code) to understand the effect of the open. A typical bug is to unwittingly shadow an identifier by opening a structure that happens to provide a binding for that identifier, even though we did not intend that it do so. In many cases this will result in a

typechecking error, but in more insidious cases it can lead to subtle run-time bugs. For example, suppose the implementation of the structure makes use of an auxiliary function as follows:

```
structure StringLT = struct
  type t = string
  fun compare (c, d) = Char.< (c, d)
  fun lt (s, t) = ... compare ...
  fun eq (s, t) = ... compare ...
end
```

Opening this structure introduces not only the expected components `t`, `lt`, and `eq`, but also the unexpected auxiliary function `compare`!

To avoid such problems it is usually advisable to avoid `open` entirely. The typical compromise is to introduce a short (typically one letter) name for the structures in question to minimize the clutter of a long path. Thus we might write

```
let
  structure I = IntLT
in
  I.lt (exp1, exp2) andalso I.eq (exp3, exp4)
end
```

rather than opening the structure `IntLT` as suggested above.

The structures and are rather simple examples of the use of the module system. A more substantial example is provided by packaging the implementation of (ephemeral) queues into a structure.

```
structure PersQueue = 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 components of this structure may be accessed by using long identifiers,

```
val q = PersQueue.empty
val q' = PersQueue.insert (1, q)
val q'' = PersQueue.insert (2, q)
val (x'', _) = PersQueue.remove q''        (* 2 *)
val (x', _) = PersQueue.remove q'          (* 1 *)
```

by opening the structure,

```
let
  open PersQueue
in
```

```
  insert (1, empty)
end
```

or by introducing a short name for it

```
let
  structure PQ = PersQueue
in
  PQ.insert (1, PQ.empty)
end
```

The structure `PersQueue` may be thought of as an implementation of the abstract data type of persistent queues. We may build and manipulate queues using the operations `PersQueue.empty`, `PersQueue.insert`, and `PersQueue.remove`. Structures are loosely analogous to classes in languages such as C++ and Java; in particular, abstract types are usually implemented by structures.

Signatures

A *signature* is the type of a structure. It describes a structure by *specifying* each of its components by giving its name and a description of it. Different sorts of components have different specifications. A type component is specified by giving its arity (number of arguments) and (optionally) its definition. A datatype component is specified by its declaration, which defines its value constructors and their types. An exception component is specified by giving the type of the values it carries (if any). A value component is specified by giving its type scheme.

Here is the signature of an ordered type, one that comes equipped with a comparison operations on it.

```
signature ORDERED = sig
  type t
  val lt : t * t -> bool
  val eq : t * t -> bool
end
```

This signature describes a structure that provides a type component named `t` (with no specified definition) and two operations, `lt` and `eq`, of type `t * t -> bool`. Ordinarily we expect that `lt` is reflexive and transitive, and that `eq` is an equivalence relation, but these requirements are not formally expressible in ML.

If we wish we can specify the definition of a type component in a signature. For example, we may define the signature

```
signature INT_ORDERED = sig
  type t = int
  val lt : t * t -> bool
  val eq : t * t -> bool
end
```

which is similar to the signature `ORDERED`, except that the type component `t` is specified to be equivalent to `int`. It therefore describes only those structures that provide an interpretation of `int` as an ordered type. (As we mentioned earler, there can be many such interpretations.)

An important consequence of having type definitions in signatures is that many superficially different signatures are equivalent. For example, the signature `INT_ORDERED` is equivalent to the following signature:

```
signature INT_ORDERED_VARIANT = sig
  type t = int
  val lt : int * int -> bool
  val eq : int * int -> bool
end
```

The reason is that since the type component `t` is defined to be `int`, we may replace it by `int` anywhere that it is used to obtain an equivalent signature. For all practical purposes the signatures `INT_ORDERED` and `INT_ORDERED_VARIANT` are indistinguishable from one another.

Here is a signature describing implementations of persistent queues:

```
signature QUEUE = sig
  type 'a queue
  val empty : 'a queue
  val insert : 'a * 'a queue -> 'a queue
  exception Empty
  val remove : 'a queue -> 'a * 'a queue
end
```

This signature specifies that an implementation of persistent queues provide a one-argument type constructor `'a queue`, the type of queues containing values of type `'a`, an exception `Empty` carrying no value, and the values `empty`, `insert`, and `remove` with types `'a queue`, `'a * 'a queue -> 'a queue`, and `'a queue -> 'a * 'a queue`, respectively.

Signature Matching

The *signature matching* relation is of central importance to the ML module system. Signature matching governs the formation of complex module expressions in the same way that type matching governs the formation of core language expressions. For example, to determine whether a structure binding `structure` *strid* `:` *sigexp* = *strexp* is well-formed, we must check that the *principal* signature of *strexp* matches the *ascribed* signature *sigexp*. The principal signature of a structure expression is the signature that most accurately describes the structure *strexp*; it contains the definitions of all of the types defined in *strexp*, and the types of all of its value components. We then compare the principal signature of *strexp* against the signature *sigexp* to determine whether or not *strexp* satisfies the requirements specified by *sigexp*.

Signature matching consists of a comparison between a *candidate* and a *target* signature. The target expresses a set of requirements that the candidate must fulfill. In the case of a structure binding the candidate is the principal signature of the structure expression, and the target is the ascribed signature of the binding. Roughly speaking, to check that a candidate siganture matches a target signature it is necessary to ensure that the following conditions hold:

1. Every type specification in the target must have a matching type specification in the candidate. If the target specifies a definition for a type, so must the candidate specify an equivalent

definition.
2. Every exception specification in the target must have an equivalent exception specification in the candidate.
3. Every value specification in the target must be matched by a value specification in the candidate with at least as general a type.

Note that the candidate signature may have *more components* than are required by the target, may have *more definitions* of types than are required, and may have value components with *more general* types. The target signature specifies a set of necessary conditions that must be met by the candidate, but the candidate may well be much richer than is required by the target.

To make these ideas precise, we decompose the signature matching relation into two sub-relations, *enrichment* and *realization*, that are defined as follows:

1. A signature *sigexp enriches* a signature *sigexp'* if *sigexp* has at least the components specified in *sigexp'*, with the types of value components being at least as general in *sigexp* as they are in *sigexp'*.
2. A signature *sigexp realizes* a signature *sigexp'* if *sigexp* fulfills at least the type definitions specified in *sigexp'*, but is otherwise identical to *sigexp'*.

In other words *sigexp* enriches *sigexp'* if we can obtain *sigexp'* from *sigexp* by dropping components and specializing types, and *sigexp* realizes *sigexp'* if we can obtain *sigexp'* from *sigexp* by "forgetting" the definitions of some of *sigexp*'s type components. It is immediate that any signature both enriches and realizes itself, and it is not hard to see that enrichment and realization are transitive.

We then say that *sigexp matches sigexp'* if there exists a signature *sigexp''* such that *sigexp* enriches *sigexp''* and *sigexp''* realizes *sigexp'*. Put in more operational terms, to determine whether *sigexp* matches *sigexp'*, we first drop components and specialize types in *sigexp* to obtain a *view sigexp''* of *sigexp* with the same components as *sigexp'*, then check that the type definitions specified by *sigexp'* are provided by the view. Signature matching can fail for several reasons:

1. The target contains a component not present in the candidate.
2. The target contains a value component whose type is not an instance of its type in the candidate.
3. The target defines a type component, that is defined differently or not defined in the candidate.

The first two reasons are failures of enrichment; the third is a failure of realization.

Some examples will clarify these definitions. Let us consider realization first since it is the simpler of the two relations. The signature `INT_ORDERED` realizes the signature `ORDERED` because we may obtain the latter from the former by "forgetting" that the type component `t` in the signature `INT_ORDERED` is defined to be `int`. The converse fails: `ORDERED` does not realize `INT_ORDERED` because `ORDERED` does not define the type component `t` to be `int`. Here is another counterexample to realization. The signature

```
signature LESS_THAN = sig
  type t = int
  val lt : t * t -> bool
end
```

does not realize the signature ORDERED, even though it defines t to be int, simply because the eq component is missing from the signature LESS_THAN.

That's all there is to say about realization. Enrichment is slightly more complicated. The signature ORDERED enriches the signature LESS_THAN because it provides all of the components required by the latter, at precisely the required types. For a more interesting example, consider the signature of monoids,

```
signature MONOID = sig
  type t
  val unit : t
  val mult : t * t -> t
end
```

and the signature of groups,

```
signature GROUP = sig
  type t
  val unit : t
  val mult : t * t -> t
  val inv : t -> t
end
```

The signature GROUP enriches the signature MONOID, as might be expected (since every group is a monoid).

The enrichment relation respects signature equivalence. For example, the signature INT_ORDERED enriches the following signature:

```
signature INT_LESS_THAN = sig
  val lt : int * int -> bool
end
```

Here we have dropped both the t and the eq components of the signature INT_ORDERED, and specified lt to have a superficially different type than is specified in the signature INT_ORDERED. As was pointed out earlier, the signature INT_ORDERED is equivalent to the signature INT_ORDERED_VARIANT, which clearly enriches the signature INT_LESS_THAN. Since enrichment respects signature equivalence, it follows that INT_ORDERED is an enrichement of INT_LESS_THAN.

The enrichment relation also allows the types of value components to be specialized by instantiating polymorphic types. For example, the signature

```
sig
  type t
  val f : 'a -> 'a
end
```

enriches the signature

```
sig
  type t
  val f : t -> t
end
```

simply because the polymorphic type `'a -> 'a` may be specialized to the required type `t -> t` (by taking `'a` to be `t`).

There is one additional case of enrichment to consider. A datatype specification may be regarded as an enrichment of 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

```
sig
  datatype 'a rbt =
    Empty | Red of 'a rbt * 'a * 'a rbt | Black of 'a rbt *
'a * 'a rbt
end
```

is considered to be an enrichment of the signature

```
sig
  type 'a rbt
  val Empty : 'a rbt
  val Red : 'a rbt * 'a * 'a rbt
end
```

which specifies two of the three value constructors of the datatype as ordinary values.

Putting these ideas together, we see that the following signature matches the signature `MONOID`:

```
sig
  type t = int list
  val unit : 'a list
  val mult : 'a list * 'a list -> 'a list
  val aux : 'a list
end
```

Why? First, we drop the component `aux`, and specialize the type of `mult` to `int list * int list -> int list` and the type of `unit` to `int list` by taking `'a` to be `int`, thereby obtaining the intermediate signature

```
sig
  type t = int list
  val unit : int list
  val mult : int list * int list -> int list
end
```

This intermediate signature is equivalent to the signature

```
sig
  type t = int list
  val unit : t
  val mult : t * t -> t
end
```

By neglecting the definition of the type `t` we obtain the signature `MONOID`. Therefore the signature match succeeds.

Signature Ascription

The point of having signatures in the language is to express the requirement that a given structure have a given signature. This is achieved by *signature ascription*, the attachment of a target signature to a structure binding. There are two forms of signature ascription, *transparent* and *opaque*, differing only in the extent to which type definitions are propagated into the scope of the binding. Transparent ascription is written as

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

Opaque ascription is written as

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

The two are distinguished by the use of a colon, "`:`", or the symbol "`:>`" before the ascribed signature.

Here is an example of transparent ascription. We may use transparent ascription on the binding of the structure variable `IntLT` to express the requirement that the structure implement an ordered type. This is achieved as follows:

```
structure IntLT : ORDERED = struct
  type t = int
  val lt = (op <)
  val eq = (op =)
end
```

Transparent ascription is so-called because the definition of `IntLT.t` is not obscured by the ascription; the equation `IntLT.t` = `int` remains valid in the scope of this declaration. Transparent ascription is appropriate here because the signature merely expresses the requirement that the given structure provide a type and two comparison operations. We do not intend that these be the *only* operations on that type. (Had we done so the structure would be useless because there would be no way to create a value of type `IntLT.t`, rendering the structure `IntLT` useless!) The structure `IntLT` may be thought of as a *view* of the type `int` as a type ordered by the standard comparison operations. We may form another view of `int` as an ordered type, but with a different ordering, by making the following binding:

```
structure IntDiv : ORDERED = struct
  type t = int
  fun lt (m, n) = (n mod m = 0)
  val eq = (op =)
end
```

Here's an example of opaque ascription. We may use opaque ascription to specify that a structure implement queues, and, at the same time, specify that *only* the operations in the signature be used to manipulate values of that type. This is achieved as follows:

```
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
```

Opaque ascription is so-called because the definition of `'a Queue.queue` is hidden by the binding; the equivalence of the types `'a Queue.queue` and `'a list * 'a list` is not propagated into the scope of the binding. This is appropriate because we wish to ensure that queues are created and manipulated only by the "official" operations in the signature, and not by any other means. By suppressing the identity of the implementation type we preclude use of any operations on values of that type other than the ones specified in the signature.

Type checking a structure binding proceeds as follows. First we determine the *principal signature* of the structure expression on the right-hand side of the binding. (It is an important property of the language that the principal signature of a structure always exists; there is always a "most accurate" description of any structure.) We then proceed according to whether there is an ascribed signature, and, in case there is, according to whether it is a transparent or opaque ascription. If there is no ascribed signature, the principal signature of the right-hand side is assigned as the signature of the structure variable. If there is an ascribed signature, we match the principal signature against it to determine whether its requirements are met. If not, the binding is rejected as ill-typed. If so, then we assign a signature to the structure variable according to whether the ascription is transparent or opaque. If it is transparent, the structure variable is assigned the *view* of the candidate signature determined by the matching process; if it is opaque, the structure variable is assigned the *ascribed* signature. This means that for a transparent ascription *the definitions in the principal signature of the types occurring in the ascribed signature are propagated into the scope of the binding*, whereas for opaque ascription *only the information explicitly appearing in the ascribed signature is propagated.* In particular if a type is specified in the ascribed signature, but no definition is provided, then the definition of that type is hidden from the clients of that binding, rendering it opaque.

It remains to define the principal signature of a structure expression. There are two forms of structure expression to be considered (at this stage): a structure variable and a `struct` expression. A structure variable has as principal signature the signature assigned to it by the ascription process just described. An `struct` expression is assigned a principal signature by a component-by-component analysis of its constituent declarations. The rules are essentially as follows:

1. Corresponding to a declaration of the form `type ('a`$_1$`,...,'a`$_n$`) t = `*typ*, the principal signature contains the specification `type ('a`$_1$`,...,'a`$_n$`) t = `*typ*.
2. Corresponding to a declaration of the form

   `datatype ('a`$_1$`,...,'a`$_n$`) t = `*con*$_1$ `of` *typ*$_1$ `|` ... `|` *con*$_k$ `of` *typ*$_k$,

   the principal signature contains the specification

   `datatype ('a`$_1$`,...,'a`$_n$`) t = `*con*$_1$ `of` *typ*$_1$ `|` ... `|` *con*$_k$ `of` *typ*$_k$.

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*, were *typ* is the principal type scheme of the expression *exp* (relative to the preceding context).

The complete rules are slightly more complicated than this because they must take account of such features as pattern-matching in value bindings, mutually recursive declarations of functions, and the possibility of shadowing bindings by re-declaration. However, the rules given above are a rough-and-ready approximation that will serve for most purposes; the reader is referred to The Definition of Standard ML for a complete account.

With these rules in mind, it is a good exercise to review the two examples of signature ascription given above. Go through the steps of forming the principal signature, then check that the principal signature matches the ascribed signature, and determine the signature to assign to the structure variable in each case.

Sample Code for this Chapter

[Home][Up][Next]

Views and Data Abstraction

[Back][Home][Up][Next]

Last edit: *Monday, April 27, 1998 02:57 PM*

Sample Code for this Chapter

It is good practice to ascribe a signature to every structure binding in a program to ensure that the signature of the bound structure variable is apparent from the binding. In the preceding chapter we described the elaboration and evaluation of a structure binding with an explicit signature ascription. First the ascribed signature is used to determine a *view* of the principal signature of the right-hand side of the binding, then the view is checked to ensure that it verifies the type sharing requirements of the ascribed signature. If both steps succeed, we assign a signature to the bound structure variable according to whether it is a transparent or opaque ascription --- if it is transparent, we assign the view to the variable, otherwise the ascription. Thus transparent ascription is used to form views of a structure, and opaque ascription is used to form abstractions in which critical type information is hidden from the rest of the program.

The formation of a view also has significance at run-time: a new structure is built consisting of only those components of the right-hand side of the binding mentioned in the ascribed signature, perhaps augmented by zero or more type components to ensure that the signature of the view is well-formed. (For example, if we attempt to extract only the constructors of a datatype, and not the datatype itself, the compiler will implicitly extract the datatype to ensure that the types of the constructors are expressible in the signature. Any type implicitly included in the view is marked as "hidden" to indicate that it was implicitly included as a consequence of the explicit inclusion of some other components of the structure.) Moreover, the types of polymorphic value components may be specialized in the view, corresponding to a form of polymorphic instantiation during signature matching. The result is a structure whose shape is fully determined by the view; no "junk" remains after the ascription. This ensures that access to the components of a structure is efficient (constant-time), and that there are no "space leaks" stemming from the presence of components of a structure that are not mentioned in its signature.

In this chapter we discuss the trade-off's between using views and abstraction in ML by offering some guidelines and examples of their use in practice. How does one decide whether to use transparent or opaque ascription? Generally speaking, transparent ascription is appropriate if the signature is not intended to be exhaustive, but is rather just a specification of some minimum requirements that a module must satisfy. Opaque ascription is appropriate if the signature is intended to be exhaustive, specifying precisely the operations that are available on the type.

Here's a common example of the use of transparent ascription in a program. When defining a module it is often convenient to introduce a numberof auxiliary bindings, especially of "helper functions" that are used internally to the code of the "public" operations. Since these auxiliaries are not intended to be used by clients of the module, it is good practice to localize them to the implementation of the

public operations. This can be achieved by using the `local` construct, as previously discussed in these notes. An alternative is to define the auxiliaries as components of the stucture, relying on transparent ascription to drop the auxiliary components before exporting the public components to clients of the module. Thus we might write something like this:

```
structure IntListOrd : ORDERED =
struct
  type t = int list
  fun aux l = ...
  val lt (l1, l2) = ... aux ...
  val eq (l1, l2) = ... aux ...
end
```

The effect of the signature ascription is to drop the auxiliary component `aux` from the structure during signature matching so that afterwards the binding of `IntListOrd` contains only the components in the signature `ORDERED`. An added bonus of this style of programming is that during debugging and testing we may gain access to the auxiliary by simply "commenting out" the ascription by writing instead

```
structure IntListOrd (* : ORDERED *) =
struct
  type t = int list
  fun aux l = ...
  val lt (l1, l2) = ... aux ...
  val eq (l1, l2) = ... aux ...
end
```

Since the ascription has been suppressed, the auxiliary component `IntListOrd.aux` is accessible for testing. (It would be useful to have a compiler switch that "turns off" signature ascription, rather than having to manually comment out each ascription in the program, but no current compilers support such a feature.)

Now let us consider uses of opaque ascription by reconsidering the implementation of persistent queues using pairs of lists. Here it makes sense to use opaque ascription since the operations specified in the signature are intended to be exhaustive --- the only way to create and manipulate queues is to use the operations `empty`, `insert`, and `remove`. By using opaque signature matching in the declaration of the `Queue` structure, we ensure that the type `Queue.queue` is hidden from the client. Consequently an expression such as `Queue.insert (1, ([],[]))` is ill-typed, even though queues are "really" pairs of lists, because the type `'a list * 'a list` is not equivalent to `'a Queue.queue`. Were we to use transparent ascription this equation would hold, which means that the client would not be constrained to using only the "official" queue operations on values of type `'a Queue.queue`. This violates the principle of *data abstraction*, which states that an abstract type should be completely defined by the operations that may be performed on it.

Why impose such a restriction? One reason is that it ensures that the client of an abstraction is insensitive to changes in the implementation of the abstraction. Should the client's behavior change as a result of a change of implementation of an abstract type, we know right where to look for the error: it can *only* be because of an error in the implementation of the operations of the type. Were abstraction not enforced, the client might (accidentally or deliberately) rely on the implementation details of the abstraction, and would therefore need to be modified whenever the implementation of

the abstraction changes. Whenever such coupling can be avoided, it is desirable to do so, since it allows components of a program to be managed independently of one another.

A closely related reason to employ data abstraction is that it enables us to enforce representation invariants on a data structure. More precisely, it enables us to isolate any violations of a representation invariant to the implementation of the abstraction itself. No client code can disrupt the invariant if abstraction is enforced. For example, suppose that we are implementing a dictionary package using a binary search tree. The implementation might be defined in terms of a library of operations for manipulating generic binary trees called `BinTree`. The implementation of the dictionary might look like this:

```
structure Dict :> STRING_DICT =
  struct
    (* Rep Invariant: binary search tree *)
    type t = string BinTree.tree
    fun insert (k, t) = ...
    fun lookup k = ...
  end
```

Had we used transparent, rather than opaque, ascription of the `STRING_DICT` signature to the `Dict` structure, the type `Dict.t` would be known to clients to be string `BinTree.tree`. But then one could call `Dict.lookup` with any value of type `string BinTree.tree`, not just one that satisfies the representation invariant governing binary search trees (namely, that the strings at the nodes descending from the left child of a node are smaller than those at the node, and those at nodes descending from the right child are larger than those at the node). By using opaque ascription we are isolating the implementation type to the `Dict` package, which means that the only possible violations of the representation invariant are those that arise from errors in the `Dict` package itself; the invariant cannot be disrupted by any other means. The operations themselves may *assume* that the representation invariant holds whenever the function is called, and are obliged to *ensure* that the representation invariant holds whenever a value of the representation type is returned. Therefore any combination of calls to these operations yielding a value of type `Dict.t` must satisfy the invariant.

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 ever 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 checkto 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.

Transparency and opacity may seem, at first glance, to be fundamentally opposed to one another. But in fact transparency is *special case* of opacity! By using type definitions in signatures, we may always express *explicitly* the propagation of type information that is conveyed *implicitly* by transparent ascription. For example, rather than write

```
structure IntLT : ORDERED = struct type t=int ... end
```

we may instead write

```
structure IntLT :> INT_ORDERED = struct type t=int ... end
```

at the expense of introducing a specialized version of the signature `ORDERED` with the type `t` defined to be `int`. This syntactic inconvenience can be ameliorated by using the "`where type`" construct, writing

```
structure IntLT :> ORDERED where type t=int = struct ...
end
```

The signature expression "`ORDERED where type t=int`" is equivalent to the signature `INT_ORDERED` defined above.

Thus transparency is a form of opacity in which we happen to publicize the identity of the underlying types in the ascribed signature. This observation is more important than one might think at first glance. The reason is that it is often the case that we must use a combination of opacity and transparency in a given situation. Here's an example. Suppose that we wished to implement several dictionary packages that differ in the type of keys. The "generic" signature of a dictionary might look like this:

```
signature DICT = sig
  type key
  val lt : key * key -> bool
  val eq : key * key -> bool
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

Notice that we include a type component for the keys, together with operations for comparing them, along with the type of dictionaries itself and the operations on it. Now consider the definition of an integer dictionary module, one whose keys are integers ordered in the usual manner. We might use a declaration like this:

```
structure IntDict :> DICT = struct
  type key = int
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

But this is wrong! The reason is that the opaque ascription, which is intended to hide the implementation type of the abstraction, also obscures the type of keys. Since the only operations on keys in the signature are the comparison functions, we can never insert an element into the dictionary!

What is necessary is to introduce a specialized version of the `DICT` signature in which we publicize

the identity of the `key` type, as follows:

```
signature INT_DICT = DICT where type key = int

structure IntDict :> INT_DICT = struct
  type key = int
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

With this declaration the type `'a IntDict.dict` is abstract, but the type `IntDict.key` is equivalent to `int`. Thus we may correctly write `IntDict.insert (IntDict.empty, 1, "1")` to insert the value `"1"` into the empty dictionary with key `1`. To build a dictionary whose keys are strings, we proceed similarly:

```
signature STRING_DICT = DICT where type key = string

structure StringDict :> STRING_DICT = struct
  type key = string
  val lt : key * key -> bool = (op <)
  val eq : key * key -> bool = (op =)
  datatype 'a dict = Empty | Node of 'a dict * 'a * 'a dict
  val empty = Empty
  fun insert (d, k, e) = ...
  fun lookup (d, k) = ...
end
```

In the next two chapters we will discuss how to build a *generic* implementation of dictionaries that may be instantiated for many different choices of key type.

Sample Code for this Chapter

[Back][Home][Up][Next]

Hierarchies and Parameterization

[Back][Home][Up]

Last edit: *Monday, April 27, 1998 02:57 PM*

Sample Code for this Chapter

In the preceding chapter we considered the following signature of dictionaries with an arbitrary key type:

```
signature DICT = sig
  type key
  val lt : key * key -> bool
  val eq : key * key -> bool
  type 'a dict
  val empty : 'a dict
  val insert : 'a dict * key * 'a -> 'a dict
  val lookup : 'a dict * key -> 'a
end
```

The signatures of dictionaries with particular choices of key type were defined using the "`where type`" construct. For example, the signature declarations

```
signature STRING_DICT = DICT where type key=string
signature INT_DICT = DICT where type key=int
```

define the signatures of dictionaries with string and integer keys, respectively. The motivation for introducing these specialized instances of the `DICT` signature is that we typically wish to hold the implementation type, `'a dict`, of dictionaries abstract, but leave the type of keys concrete, as described earlier.

The signature `DICT` is a bit unsatisfactory because it mixes two different notions in one interface, namely the type, `key`, of keys and its associated comparison operations, `lt` and `eq`, and the type `'a dict` of dictionaries and its associated operations `empty`, `insert`, and `lookup`. It would be cleaner to separate these two aspects of the interface, especially since we shall soon consider the key component to be "generic", with the rest being "specific", to the abstraction. The way to do this in ML is with a *substructure*, 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
end
```

The type of keys and the operation on it are segregated into a *substructure* of the dictionary structure, a component of a structure that is itself a structure. Correspondingly, uses of the type `key` are replaced by references to the `t` component of the substructure `Key`. This leads to a *hierarchical* organization in which we consider the key structure to be subservient to the dictionary operations.

Specialized versions of the signature DICT are build essentially as before, except that we use a long identifier to specify the type of keys:

```
signature STRING_DICT = DICT where type Key.t=string
signature INT_DICT = DICT where type Key.t=int
```

Specific implementations of these specialized instances may be defined as follows:

```
structure StringDict :> STRING_DICT = struct
  structure Key : ORDERED = StringLT
  type 'a dict = Key.t BinTree.tree
  val empty = BinTree.empty
  val insert = ...insert into a BST using Key.lt and
Key.eq...
  val lookup = ...lookup in a BST using Key.lt and
Key.eq...
end

structure IntDict :> INT_DICT = struct
  structure Key : ORDERED = IntLT
  type 'a dict = Key.t BinTree.tree
  val empty = BinTree.empty
  val insert = ...insert into a BST using Key.lt and
Key.eq...
  val lookup = ...lookup in a BST using Key.lt and
Key.eq...
end
```

The difficulty, of course, is that we are repeating the code for dictionaries in each implementation; the elided parts of both structures would be *identical*. The *only* difference between the two dictionary structures lies in the implementation of keys; in one case we choose string operations and in the other we choose integer operations. Since the bulk of the code is the same, it is a pity to have to repeat it for reach choice of key type.

Fortunately, ML provides a convenient means of avoiding such redundancy, called a *functor*. A functor is a *parameterized module*, or a *generic structure*, that is defined in terms of zero or more argument structures with a specified signature. A functor may be *applied*, or *instaniated*, with any structures matching the argument signatures. A functor is therefore a kind of function taking zero or more structures as arguments and yielding a structure as result.

In the case of dictionaries we may define a generic implementation that is parameterized by the type of keys and associated comparison operations. This is achieved by introducing a functor.

```
functor Dict (structure K : ORDERED) :> DICT where type
Key.t=K.t =
struct
   structure Key : ORDERED = K
   type 'a dict = Key.t BinTree.tree
   val empty = BinTree.empty
   val insert = ...insert into a BST using Key.lt and
Key.eq...
   val lookup = ...lookup in a BST using Key.lt and
Key.eq...
end
```

This declaration introduces a functor named `Dict` that takes as argument any structure implementing the signature `ORDERED`, and yields a structure implementing the instance of the signature `DICT` determined by taking the key type of the dictionary to be the type component of its argument, leaving the type of dictionaries abstract. The type checker ensures that the body of the functor matches the specified result signature, under the assumption that the argument has the stated signature. In the case of the `Dict` functor the type checker ensures that the principal signature of the body of the functor (the part between `struct` and `end`) matches the signature

```
DICT where type Key.t=K.t,
```

assuming that the structure `K` has signature `ORDERED`.

The `Dict` functor encapsulates the implementation of dictionaries as a generic structure that is independent of the specific choice of keys. One advantage of this encapsulation is that should we wish to modify the implementation of dictionaries, say to fix an error or to improve performance, we need only modify the `Dict` functor, rather than change every occurrence of the dictionary code spread throughout a large system. This is obviously advantageous for both the original author of the code, and anyone who must maintain it in the future. In fact common data structures such as dictionaries are typically provided as part of a "shrink wrapped" library, and hence are shared among many different programs, thereby increasing code reuse and reducing redundancy.

The `Dict` functor provides a generic implementation of dictionaries. Dictionaries with specific key types may be built by instantiating the `Dict` functor as follows:

```
structure IntDict = Dict (structure K = IntLT)
structure StringDict = Dict (structure K = StringLT)
```

Notice that functor application uses keyword parameter passing --- the parameter is explicitly bound to a structure using a structure binding. In practice the right-hand sides of such bindings are always (long) identifiers; if not, the compiler implicitly inserts bindings to ensure that this is the case. In our discussions we will tacitly assume that the right-hand side of all such bindings are (long) identifiers.

What are the signatures of the structure variables `IntDict` and `StringDict`? Since no signature is ascribed to these bindings, the principal signature of the corresponding right-hand side of the binding is assigned to each variable, in keeping with our previous policies. Since the right-hand side in these examples is a functor application, we must answer the question: what is the principal signature of a functor application? If --- as here --- the result signature of the functor is opaque, the

principal signature is precisely the asribed signature of the functor, but with the structure parameter replaced by its binding (which must be, by our assumption, another structure identifier). Thus the signature assigned to `IntDict` is

```
DICT where type Key.t=IntLT.t
```

which is equivalent to the signature

```
DICT where type Key.t=int
```

since `IntLt.t` = `int`. Similarly, the signature assigned to `StringDict` is

```
DICT where type Key.t=StringLT.t
```

which is equivalent to the signature

```
DICT where type Key.t=string
```

What if the functor has no result signature, or its result signature is transparently ascribed? In that case we assign the intermediate signature of the match as the result signature of the functor, and use that signature as the implied result signature of the functor.

Dictionaries illustrate the use of the ML module system to build generic implementations of abstract types. A generic implementation of priority queues (which support a `remove_min` operation that dequeues the "least" element of the queue relative to a specified ordering) may be built in an exactly analogous manner. Here's a suitable signature of priority queues:

```
signature PRIO_QUEUE = sig
  structure Elt : ORDERED
  type prio_queue
  exception Empty
  val empty : prio_queue
  val insert : Elt.t * prio_queue -> prio_queue
  val remove : prio_queue -> Elt.t * prio_queue
end
```

Notice that `prio_queue` is a type, and not a type constructor, as it was in the case of "plain" queues. This is a reflection of the fact that the operations on a priority queue are not independent of the type of elements (as they are with plain queues), but rely on the comparison operations that are provided with the `Elt` structure.

A generic implementation of priority queues is a functor taking as argument a structure containing the element type together with its associated operations:

```
functor PrioQueue
  (structure E : ORDERED) :> PRIO_QUEUE where type
Elt.t=E.t =
struct
  structure Elt : ORDERED = E
  type prio_queue = ...a heap based on the ordering
```

```
    Elt.lt...
      exception Empty
      val empty = ...the empty heap...
      val insert = ...sift a new element into the heap...
      val remove = ...remove the least element and adjust the
    heap...
    end
```

Specific instances of priority queues may be built as follows:

```
    structure IntPQ = PrioQueue (structure E = IntLT)
    structure StringPQ = PrioQueue (structure E = StringLT)
```

with signatures

```
    PRIO_QUEUE where type Elt.t=int
```

and

```
    PRIO_QUEUE where type Elt.t=string
```

respectively.

The situation becomes more interesting when we wish to combine two or more abstract types to form a third. Suppose we are to implement a (hypothetical) abstract type that employs an ordered type of values that occur both as keys of a dictionary and elements of a priority queue. The signature of this abstract type might look like this

```
    signature ADT = sig
      structure Val : ORDERED
      type adt
      ...operations...
    end
```

The implementation should be generic in the type of values, and also in the implementation of dictionaries and priority queues; we don't want to build the implementation of these auxiliary data structures into the implementation of `ADT`'s. There are two approaches to building an `Adt` functor, each with its advantages and disadvantages. Here's the first approach:

```
    functor Adt
      (structure V : ORDERED) :> ADT where type Val.t=V.t =
    struct
      structure Val : ORDERED = V
      structure D = Dict (structure K = V)
      structure Q = PrioQueue (structure E = V)
      type adt = ...
      ...
    end
```

The functor `Adt` instantiates the `Dict` and `PrioQueue` functors to the structure of values specified

as argument to the Adt functor. This ensures that the type equation

```
D.Key.t = Q.Elt.t = V.t
```

holds inside the body of the functor, so that expressions such as

```
D.insert (Q.remove_min ..., ...)
```

are well-typed. (The structures D and Q are not visible outside of the functor since they do not appear in the result signature; they are local auxiliaries used within the functor.)

This approach works well, but if the Dict or PrioQueue functors are changed, the Adt functor must be recompiled to pick up the new versions. An alternative, which avoids this dependency of the implementation of Adt on the implementations of the Dict and PrioQueue functors, is to treat the dictionary and priority queue structures as additional parameters to the Adt functor. This leads to the following setup:

```
functor Adt'
  (structure V : ORDERED and D : DICT and Q : PRIO_QUEUE)
:>
    ADT where type Value.t=V.t =
struct
  structure Val = V
  type adt = ...implementation type...
  ...implementation of operations...
end
```

To build an instance of the Adt' functor we must first built appropriate instances of the Dict and PrioQueue functors and pass these to Adt':

```
structure IntDict = Dict (structure K=IntLT)
structure IntPQ = Dict (structure K=IntLT)
structure A = Adt' (structure V=IntLt and D=IntDict and
Q=IntPQ)
```

There is a problem, however, with this setup: the functor Adt' is ill-typed! It is no longer true within the body of Adt that the type equation

```
D.Key.t = Q.Elt.t = V.t
```

holds in the body of Adt', even though the equation

```
IntDict.Key.t = IntPQ.Elt.t = IntLT.t = int
```

does hold of the arguments, for we might well choose arguments for which the required equation is invalid. In short, the functor is "too generic", and consequently the body is not type correct.

What to do? The solution is to restrict the parameters to the Adt' functor so that the only possible instances are those that satisfy the required equation. There are two methods for doing this, both equivalent. The first is to explicitly require that the dictionary and priority queue arguments agree on

the value type passed as parameter:

```
functor Adt'
  (structure V : ORDERED
   and D : DICT where type Key.t=V.t
   and Q : PRIO_QUEUE where type Elt.t=V.t) :>
    ADT where type Val.t=V.t =
struct
  ...
end
```

The body of `Adt'` is now type correct since the required type equations hold as a result of our additional assumptions on the arguments.

An alternative is to impose the equational requirement on types in a *post hoc* manner using a *sharing specification*:

```
functor Adt'
  (structure V : ORDERED and D : DICT and Q : PRIO_QUEUE
   sharing type D.Key.t = Q.Elt.t = V.t) :>
    ADT where type Val.t=V.t =
struct
  ...
end
```

The sharing specification stipulates that the given equation must hold of any instance of this functor. Any attempt to instantiate `Adt'` with structures `V`, `D`, and `Q` not satisfying the sharing specification is rejected as ill-formed.

An advantage of sharing specifications is that they provide a direct, symmetric specification of the required type equation without forcing the programmer to explicitly "thread" the common type through the various signatures. In fact sharing specifications encourage concision since they do not require that the common component be "factored out" as it is in the foregoing example. Here is a more concise formulation of the `Adt'` functor in which we drop the first argument entirely, relying only on a `sharing` specification to constraint the dictionary and priority queue structures appropriately.

```
functor Adt'
  (structure D : DICT and Q : PRIO_QUEUE
   sharing type D.Key.t = Q.Elt.t) :>
    ADT where type Val.t=D.Key.t =
struct
  ...
end
```

Notice that the result signature changes slightly to extract the common type from one of the parameters, the choice of which being arbitrary in the presence of the `sharing` specification.

Sample Code for this Chapter

[Back][Home][Up]

Programming Techniques

[Back][Home][Next]

Last edit: *Monday, May 04, 1998 03:29 PM*

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.

[Back][Home][Next]

Induction and Recursion

[Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:29 PM*

Sample Code for This Chapter

This chapter is concerned with the close relationship between *recursion*and *induction* in programming. When defining a recursive function, be sure to write down a clear, concise *specification* of its behavior, then mentally (or on paper) give an *inductive proof* that your code satisfies the specification. What is a specification? It includes (at least) these ingredients:

1. Assumptions about the types and values of the arguments to the function. For example, an integer argument might be assumed to have a non-negative value.
2. Guarantees about the result value, expressed in terms of the argument values, under the assumptions governing the arguments.

What does it mean to prove that your program satisfies the specification? It means to give a rigorous argument that if the arguments satisfy the assumptions on the input, then the program will terminate with a value satisfying the guarantees stated in the specification. In the case of a recursively-defined function the argument invariably has the form of a inductive proof based on an *induction principle* such as mathematical induction for the natural numbers or, more generally, structural induction for other recursively-defined types. The rule of thumb is this

when programming recursively, think inductively

If you keep this rule firmly in mind, you'll find that you are able to get your code right more often without having to resort to the tedium of step-by-step debugging on test data.

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 $n>=0$. We seek to define the function

exp : int -> int

satisfying the specification

if n>=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>=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*p*, where *p* is the value of *exp (n-1)*. Inductively, $p=2^{n-1}$, so $2*p = 2*2^{n-1} = 2^n$, 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 (they are), then we can read off a recurrence describing its execution time as follows:

$T(0) = O(1)$
$T(n+1) = O(1)+ T(n)$

In fact this recurrence could itself be thought of as defining a function in ML simply by rewriting it into ML syntax! However, in most cases 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:

$S(0) = O(1)$
$S(n+1) = O(1) + S(n)$

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 $2^{(n \text{ div } 2)}$, so squaring it yields $2^{(n \text{ div } 2)} * 2^{(n \text{ div } 2)} = 2^{2*(n \text{ div } 2)} = 2^n$, 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:

T(0) = O(1)
T(2n) = O(1) + T(n)
T(2n+1) = O(1) + T(2n) = O(1) + T(n)

Solving this recurrence using standard techniques yields the solution

T(n) = O(lg n)

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>=0, then iterative_fast_exp (n, a) evaluates to $2^n * a$

Here's some code that achieves this specification:

```
fun iterative_fast_exp (0, a) = a
  | iterative_fast_exp (n, a) =
    if n mod 2 = 0 then
```

```
        iterative_fast_exp (n div 2, iterative_fast_exp (n
    div 2, a))
      else
        iterative_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 $2^{(n \text{ div } 2)} * 2^{(n \text{ div } 2)} * a$ by two recursive calls to *iterative_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:

$T(0) = 1$
$T(2n) = O(1) + 2T(n)$
$T(2n+1) = O(1) + T(2n) = O(1) + 2T(n)$

Here again we have a standard recurrence whose solution is

$T(n) = O(n \lg n)$

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 $n>=0$, then generalized_iterative_fast_exp (b, n, a) evaluates to $b^n * a$

Here's the code:

```
fun generalized_iterative_fast_exp (b, 0, a) = a
  | generalized_iterative_fast_exp (b, n, a) =
    if n mod 2 = 0 then
       generalized_iterative_fast_exp (b*b, n div 2, a)
    else
       generalized_iterative_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 $(b*b)^{(n \text{ div } 2)} * a = b^n * a$, and if n is odd, we obtain inductively the result $b^{(n-1)} * b * a = b^n * a$. 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 generalized_iterative_fast_exp (b, 0, a) =
```

```
    | generalized_iterative_fast_exp (b, n, a) = ... as
above ...
in
  fun exp n = generalized_iterative_fast_exp (2, n, 1)
end
```

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.

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, m|n,* iff *n* is a multiple of *m*, which is to say that there is some *k>=0* such that *n=km*. 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>=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:

```
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 *mn*. If *mn* 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<=n*. Suppose that *m>n*. Observe that *m mod n = m - (m div n)n*, so that *(m mod n)n = mn - (m div n)n*^{2} *< mn*, so that by induction we return the g.c.d. of *m mod n* and *n*. It remains to show that this is the g.c.d. of *m* and *n*. If *d* divides both *m mod n* and *n*, then *kd = (m mod n) = (m - (m div n)n)* and *ld = n* for some non-negative *k* and *l*. Consequently, *kd = m - (m div n)ld*, so *m = (k+(m div n)l)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 *(m mod n)* and *n*, so *d>d'*. That is, *d* is the g.c.d. of *m* and *n*. The other case, *m<=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 "bulletproofing" --- 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 d mod m = 0 andalso d mod n = 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 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=kd* for some *k*>=0, *n=ld* for some *l*>=0, and *d=am+bn* 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=am+bn*. Here's the specification:

> *if m,n>=0, then ggcd (m, n) evaluates to (d, a, b) such that d divides m, d divides n, and d=am+bn; 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 *mn*. If *mn=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 *mn>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 *m mod n* and *n*, and hence is the g.c.d. of *m*and *n*, and *d=a(m mod n) + bn*. Since *m mod n = m - (m div n)n*, it follows that *d = am + (b-a(m div n))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.

Sample Code for This Chapter

[Home][Up][Next]

Structural Induction

[Back][Home][Up][Next]

Last edit: *Tuesday, May 05, 1998 12:24 PM*

Sample Code for this Chapter

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. 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>=0* that the value of *f* is uniquely determined on all values *m<=n*. If *n=0*, this is obvious since *f(0)* is defined by clause (1). If *n=m+1*, then by induction the value of *f* is determined for all values *k<=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.)

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 fpr *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*`]`.

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 two_three_tree =
  Empty |
  Binary of 'a * 'a two_three_tree * 'a two_three_tree |
  Ternary of 'a * 'a two_three_tree * 'a two_three_tree *
'a two_three_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 (Binary (_, t1, t2)) = ???
  | size (Ternary (_, 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 (Binary (_, t1, t2)) = 1 + size t1 + size t2
  | size (Ternary (_, 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 two_three_tree`.

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.

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_recursion base step =
    let
        fun loop Zero = base
          | loop (Succ n) = step (loop n)
    in
        loop
    end
```

This function has the following type

```
'a -> ('a -> 'a) -> nat -> 'a
```

Given the first two arguments, `nat_recursion` 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_recursion` to define the exponential function:

```
val double = nat_recursion Zero (fn result => Succ (Succ
result))
val exp = nat_recursion (Succ Zero) double
```

Note well the pattern! The arguments to `nat_recursion` 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 two_three_tree` is given as follows:

```
fun two_three_recursion base binary_step ternary_step =
    let
        fun loop Empty = base
          | loop (Binary (v, t1, t2)) =
            binary_step (v, loop t1, loop t2)
          | loop (Ternary (v, t1, t2, t3)) =
            ternary_step (v, loop t1, loop t2, loop t3)
```

Notice that we have two inductive steps, one for each form of node. The type of `two_three_recursion` is

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

We may instantiate it to define the function size as follows:

```
val size =
    two_three_recursion 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.

Sample Code for this Chapter

[Back][Home][Up][Next]

Proof-Directed Debugging

[Back][Home][Up][Next]

Last edit: *Thursday, June 25, 1998 02:57 PM*

Sample Code for this Chapter

It is difficult to write a program that works well. A significant part of the problem is to state precisely what it means for a program to work correctly. What assumptions do we make about the way in which it is invoked? What guarantees does it make about its results? How much time and space does it require? Answers to these questions are called *specifications* --- descriptions of the expected behavior of a program. Checking that a particular program satisfies a given specification is called *verification*. There are many forms of specification and many techniques for verification of programs. One form of specification with which you are by now very familiar is a *type specification*; verification of a type specification is called *type checking*. We've seen that type specification and type checking are useful tools for helping us to get programs right. Another form of specification is an asymptotic time and space bound on a procedure, expressed as a function of the argument to the procedure. For example, we may state that the function `sort : int list -> int list` takes time $T(n) = O(n \lg n)$ and space $S(n) = O(n)$ for an input of size n. Verification of a complexity bound is often a tricky business. Typically we define a recurrence relation governing the time or space complexity of the program, then solve the recurrence using asymptotic methods to obtain the result.

Type specifications and complexity specifications are useful tools, but it is important to keep in mind that neither says very much about whether the code works properly. We might define an incorrect sort routine (say, one that always returns its input untouched), verify that its type is `int list -> int list`, and check that it runs in time $O(n \lg n)$, yet the code doesn't sort its input, despite its name! Clearly more refined forms of specification are needed to state precisely the expected behavior of a program, and some methods are needed to verify that a program satisfies a given specification. We've explored such forms of specification and verification earlier in these notes, for example when we checked the correctness of various forms of the integer exponential function. In this chapter we'll put these ideas to work to help us to devise a correct version of the regular expression matcher sketched in the Overview, correcting a subtle error that may not be immediately apparent from inspecting or even testing the code. The goal of this chapter is to demonstrate the use of specification and verification to discover and correct an error in a program through a technique that we call *proof-directed debugging*. We first devise a precise specification of the regular expression matcher, a difficult problem in itself, then attempt to verify that the matching program satisfies this specification. The attempt to carry out a proof breaks down, and suggests a counterexample to the specification. We then consider various methods of handling the problem, ultimately settling on a *change of specification* rather than a *change of implementation*.

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 function `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).*

Recall that the *language* of a regular expresson *r* is a set of string *L(r)* defined as follows:

$$L(0) = 0$$

$$L(1) = 1$$

$$L(a) = a$$

$$L(r_1 r_2) = L(r_1)\ L(r_2)$$

$$L(r_1 + r_2) = L(r_1) + L(r_2)$$

$$L(r^*) = L(0) + L(r) + L(rr) + L(rrr) + \ldots$$

where $0 = \{\}$, $1 = \{""\}$, $a = \{"a"\}$, $L_1\ L_2 = \{\ s_1 s_2 \mid s_1 \text{ in } L_1 \text{ and } s_2 \text{ in } L_2\ \}$, and $L_1 + L_2 = \{\ s \mid s \text{ in } L_1 \text{ or } s \text{ in } L_2\ \}$. The language $L(r^*)$ can be characterized as the smallest language L such that $L = 1 + L(r)\ L$. For if $s \text{ in } L(r^*)$ as defined above, then $s \text{ in } L(r^i)$ for some $i >= 0$. We may show by induction on i that $s \text{ in } 1 + L(r)L$. If $i=0$, then $s=""$ *in 1*, and if $i>0$, then $s=tu$ with $t \text{ in } L(r)$ and $u \text{ in } L(r^{i-1})$. By induction u *in* L, and hence $s \text{ in } 1 + L(r)L$ and hence s *in* L. Conversely, if s *in* L, then either s *in 1*, in which case $s \text{ in } L(r^*)$, or $s=tu$ with $t \text{ in } L(r)$ and u *in* L. Inductively $u \text{ in } L(r^*)$ and hence $s \text{ in } L(r)L(r^*)$ and hence s *in* L.

We saw in the Overview 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'' evaluating 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 implementation of `match_is`! 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. 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. Formally, we can reduce the *halting problem* to the matching problem so defined, which suffices to show that no such `match_is` procedure exists. Instead, we

must restrict attention to *total* continuations, those that terminate for all inputs. 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'' evaluating to true, then* `match_is` *r cs k evaluates to true; otherwise,* `match_is` *r cs k evaluates to false.*

Observe that the condition *"If cs=cs'@cs' with ..., then ..."* contains an implicit existential quantification. Written out in full, we might say *"If there exists cs' and cs'' such that cs = cs'@cs'' with ..., then ..."*. This is an important observation because it 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 consider *any* partitioning whose initial segment matches *r*; we can consider only those that also induce *k* to accept the corresponding final segment.

Suppose for the moment that `match_is` satisfies this specification. Does it follow that match satisfies the original specification? Recall that 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, if `match_is` satisfies its specification, then `match` satisfies the following property obtained by plugging in the initial continuation:

> *For every regular expression r and character list cs, if cs in L(r), then* `match` *r cs evaluates to true, and otherwise* `match` *r cs 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 (satisfies its specification).

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) (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 *0*. Then no string is in *L(r)*, so `match_is` must return *false*, which indeed it does. Suppose that *r* is *1*. Since the null string is an initial segment of every string, and the null string is in *L(1)*, we must yield *true* iff *k cs* yields *true*, and false otherwise. Again, 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 iff either some initial segment matches r_1 and drives *k* to accept or some initial segment matches r_2 and drives *k* to accept. 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 $r=r_1r_2$, 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_1r_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 either no initial segment of *cs* matches r_1, in which case the outer recursive call yields *false*, as required, or 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 else *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. (Write out the argument to see where you get into trouble.) What to do? A moment's thought suggests that we proceed by an inner induction on the length of the string, based on the theory 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. But this time we can use the failure of the proof to obtain a counterexample to the specification! For if $r=1^*$, for example, then `match_is` *r cs*

k does not terminate! In general if $r=r_1^*$ with *"" 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 derived for it! Our conjecture is false!

We have used the failure of an attempt to prove that `match_is` satisfies a reasonable specification of its behavior to discover a bug in the code --- the matcher does not always terminate. What to do? One approach is to explicitly check for failure to make progress when matching against an iterated regular expression. 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 of correctness sketched above goes through, provided that the regular expression satisfies the condition that no iterated regular expression matches the null string. That is, r^* is admitted as a valid regular expression only if *""* is not in *L(r)*. Call a regular expression satisfying this condition *standard*. As an exercise check that the proof sketched above goes through under the additional assumption that *r* is a standard regular expression.

Thus the matcher behaves correctly for all standard regular expressions. But what about those non-standard ones? A simple observation is that *every regular expression is equivalent to one in standard form*. That is, we never really 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. This pre-processing is based on the following definitions. First, we define *null(r)* to be the regular expression *1* if *r* accepts the null string, and the regular expression *0* if not. Then we define *nonnull(r)* to be a regular expression *r'* in standard form such that *L(r') = L(r) \ {""}*--- that is, *r'* accepts the same strings as *r*, except for the null string. Thus for every regular expression *r*, we have

L(r) = L(null(r)+nonnull(r)).

Moreover, the regular expression *null(r)+nonnull(r)* is in standard form.

Here is the definition of *null*:

null(0) = 0
null(1) = 1
null(a) = 0
null(r_1*+*r_2*) = null*(r_1*) ++ null*(r_2*)*
null($r_1 r_2$*) = null*(r_1*) ** null*(r_2*)*
null(r^**) = 1*

where we define *0++1=1++0=1++1=1* and *0++0=0* and *0**1=1**0=0**0=0* and *1**1=1*.

Here is the definition of *nonnull*:

nonnull(0) = 0
nonnull(1) = 0
nonnull(a) = a
nonnull(r_1*+*r_2*) = nonnull*(r_1*)+nonnull*(r_2*)*

$nonnull(r_1 r_2) = null(r_1)nonnull(r_2) + nonnull(r_1)nonnull(r_2)$

$nonnull(r^*) = null(r) + nonnull(r)^*$

Check that the stated properties of these regular expressions indeed hold true, and use these equations to define a pre-processor to put every regular expression into standard form.

This chapter is based on the paper entitled *Proof-Directed Debugging*, which is scheduled to appear as a Functional Pearl article in the *Journal of Functional Programming*.

Sample Code for this Chapter

[Back][Home][Up][Next]

Infinite Sequences

[Back][Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:28 PM*

Sample Code for this Chapter

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 *i*th 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!

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

  val constantly : 'a -> 'a seq               (* constant
sequence *)
  val alternately : 'a * 'a -> 'a seq         (*
alternating values *)
  val insert : 'a * 'a seq -> 'a seq          (* insert an
element at the front *)

  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
  val merge : ('a * 'a) seq -> 'a seq          (* fair
```

```
merge *)

  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:

```
fun loopback loop = loop (loopback loop)     (* bad
definition *)
```

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*:

f 0 = 1
f 1 = 1
f (n+2) = f (n+1) + f (n)

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 *s n = s n + 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.

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".

```
infixr **;
fun (f ** g) (x, y) = (f x, g y)        (* apply two
functions in parallel *)

fun logical_and (Low, _) = Low          (* hardware logical
and *)
  | 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

val logical_nor =
    logical_and o (logical_not ** logical_and)  (* a nor b
= not a and not b *)

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, following 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`.

Sample Code for this Chapter

[Back][Home][Up][Next]

Representation Invariants and Data Abstraction

[Back][Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:29 PM*

Sample Code for This Chapter

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.

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 dict
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.

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.

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 $2(2^h-1)+1 = 2^{h+1}-1$ 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 $2(2^{h-1}-1)+1 = 2^h-1$ nodes. Now, observe that a red-black tree of height *h* with *n* nodes has black height at most $h/2$, and hence has at least $2^{h/2}-1$ nodes. Consequently, $lg(n+1)>=h/2$, so $h <= 2lg(n+1)$. 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 violation, 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

  (* Representation invariant: 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 violate color invariant at
root *)
          (* ins (Black _) or ins (Empty) will be red/black
tree *)
          (* 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.

Sample Code for This Chapter

[Back][Home][Up][Next]

Persistent and Ephemeral Data Structures

[Back][Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:29 PM*

Sample Code for This Chapter

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:

```
val l = [1,2,3]              (* original list *)
val m1 = hd l                (* first future of l *)
val n1 = rev m1
val m2 = l @ [4,5,6]        (* second future of l *)
```

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.

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 chapter 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 invariant 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.*

This invariant is 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.

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 3*n* 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)<=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)>=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 $C(n)<=2n$ for any sequence of n inserts and removes. Thus $C(n)=O(n)$. After any sequence of $n>=0$ operations have been performed, the queue contains $0<=b<=n$ elements on the back "half" and $0<=f<=n$ elements on the front "half". We claim that for every $n>=0$, $R(n)=b$. We prove this by induction on $n>=0$. 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 $R(n)=b>=0$, and hence $C(n)>=T(n)$.

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*.

Sample Code for This Chapter

[Back][Home][Up][Next]

Options, Exceptions, and Failure Continuations

[Back][Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:29 PM*

Code for This Chapter

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.

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 *nxn* 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 *nxn* 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 $n>=0$. 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

  (* representation: size, next free column, number placed,
placements *)
  (* rep'n invariant: 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>=0*, then `queens` *n* evaluates either to `NONE` if there is no safe placement of *n* queens on an *nxn* 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.

Here's a solution based on option types:

```
(* addqueen bd evaluates to SOME bd', where bd' is 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.

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 evaluates to 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.

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 evaluates to 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.

Code for This Chapter

[Back][Home][Up][Next]

Memoization and Laziness

[Back][Home][Up][Next]

Last edit: *Monday, May 04, 1998 03:28 PM*

Code for this Chapter

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 implementation provided by the SML/NJ compiler.

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` *f n* computes the sum of values of a function *f (k)* with *k* running from *1* to *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!

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\n"            (* 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\n")
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.

The constructs for manipulating lazy data structures provided by the SML/NJ compiler may be explained in terms of suspensions. For the sake of specificity we'll consider the implementation of streams, but the same ideas apply to any lazy datatype.

The type declaration

```
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

```
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.

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.

Code for this Chapter

[Back][Home][Up][Next]

Modularity and Reuse

[Back][Home][Up]

Last edit: *Monday, May 04, 1998 03:29 PM*

In this chapter we illustrate the use of the ML module system to build a program from re-usable components. The main example is a generic game-tree search algorithm.

[Back][Home][Up]

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/recind.sml] Page 31

```
fun exp 0 = 1
  | exp n = 2 * exp (n-1) ;

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)) ;

fun iterative_fast_exp (0, a) = a
  | iterative_fast_exp (n, a) =
    if n mod 2 = 0 then
       iterative_fast_exp (n div 2, iterative_fast_exp (n div 2, a))
    else
       iterative_fast_exp (n-1, 2*a) ;

fun generalized_iterative_fast_exp (b, 0, a) = a
  | generalized_iterative_fast_exp (b, n, a) =
    if n mod 2 = 0 then
       generalized_iterative_fast_exp (b*b, n div 2, a)
    else
       generalized_iterative_fast_exp (b, n-1, b*a) ;

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) ;

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

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
```

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/structur.sml] Page 32

```
(* Natural numbers in unary *)

datatype nat = Zero | Succ of nat

fun add (m, Zero) = m
  | add (m, Succ n) = Succ (add (m, n))

fun mul (m, Zero) = Zero
  | mul (m, Succ n) = add (mul (m, n), m)

fun double Zero = Zero
  | double (Succ m) = Succ (Succ (double m))

fun exp Zero = Succ Zero
  | exp (Succ m) = double (exp m)


(* Lists *)

(* datatype 'a list = nil | :: of 'a * 'a list *)

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

(* Two-three trees *)

datatype 'a two_three_tree =
    Empty
  | Binary of 'a * 'a two_three_tree * 'a two_three_tree
  | Ternary of 'a * 'a two_three_tree * 'a two_three_tree * 'a two_three_tree

fun size Empty = 0
  | size (Binary (_, t1, t2)) = 1 + size t1 + size t2
  | size (Ternary (_, t1, t2, t3)) = 1 + size t1 + size t2 + size t3

(* Recursion patterns *)

fun nat_recursion base step =
    let
        fun loop Zero = base
          | loop (Succ m) = step (m, loop m)
    in
        loop
    end

val double = nat_recursion (Zero) (fn (_, result) => Succ (Succ result))
val exp = nat_recursion (Succ Zero) (fn (_, result) => double result)

fun list_recursion base step =
    let
        fun loop nil = base
          | loop (h::t) = step (h, loop t)
    in
        loop
    end

fun reverse l = list_recursion nil (fn (h, t) => t @ [h]) l

fun two_three_recursion base step2 step3 =
    let
        fun loop Empty = base
```

```
          | loop (Binary (v, t1, t2)) =
            step2 (v, loop t1, loop t2)
          | loop (Ternary (v, t1, t2, t3)) =
            step3 (v, loop t1, loop t2, loop t3)
    in
        loop
    end

fun size t =
    two_three_recursion
    0
    (fn (_, s1, s2) => 1+s1+s2)
    (fn (_, s1, s2, s3) => 1+s1+s2+s3)
    t
```

```
(* Lists with mutable tails. *)

datatype 'a mutable_list = Nil | Cons of 'a * 'a mutable_list ref

local
   fun ipr (Nil, a) = a
     | ipr (this as (Cons (_, r as ref next)), a) =
       ipr (next, (r := a; this))
in
   (* destroys argument, yields its reversal *)
   fun inplace_reverse l = ipr (l, Nil)
end

(* Queues *)

(* Signature of queues as an abstract type. *)
signature QUEUE = sig

  type 'a queue

  exception Empty

  val new : unit -> 'a queue

  val insert : 'a * 'a queue -> 'a queue

  val remove : 'a queue -> 'a * 'a queue

end

(* Inefficient implementation of a persistent queue as a list.  A sequence
   of n operations takes O(n^2) time in the worst case. *)
structure NaiveQueue :> QUEUE = struct

    type 'a queue = 'a list

    fun new () = nil

    fun insert (x, q) = x::q

    exception Empty

    fun remove [x] = (x, nil)
      | remove (x::xs) =
        let
            val (y, q) = remove xs
        in
            (y, x::q)
        end

end

(* Persistent queues with amortized constant-time behavior for
   single-threaded executions of queue operations.  Rep invariant:
   1. front is empty only if the back is empty
   2. list of elements (in order of departure) of the queue (bs, fs)
      is fs @ rev bs *)
structure AmortizedSingleThreadedQueue :> QUEUE = struct

    type 'a queue = 'a list * 'a list
```

```
    (* smart constructor to enforce rep inv *)
    fun make_queue (q as (nil, nil)) = q
      | make_queue (q as (bs, nil)) = (nil, rev bs)
      | make_queue q = q

    (* queue operations *)
    fun new () = make_queue (nil, nil)

    fun insert (b, (bs, fs)) = make_queue (b::bs, fs)

    exception Empty

    fun remove (_, nil) = raise Empty
      | remove (bs, f::fs) = (f, make_queue (bs, fs))

end;

(* Amortized constant-time single-threaded queues, variant representation
   in which a queue has the form (bs, sb, fs, sf) satisfying the rep inv:
   1. sb = length bs, sf = length fs
   2. sf >= sb
*)
structure AmortizedSingleThreadedQueue2 :> QUEUE = struct

    type 'a queue = 'a list * int * 'a list * int

    fun make_queue (q as (bs, sb, fs, sf)) =
        if sf >= sb then
            q
        else
            (nil, 0, fs @ rev bs, sf+sb)

    fun new () = make_queue (nil, 0, nil, 0)

    fun insert (b, (bs, sb, fs, sf)) = make_queue (b::bs, sb+1, fs, sf)

    exception Empty

    fun remove (_, _, _, 0) = raise Empty
      | remove (bs, sb, f::fs, sf) = (f, make_queue (bs, sb, fs, sf-1))

end

(* Naive attempt to handle the multi-threaded case by memoization.  Fails
   to achieve an amortized constant-time bound in general. (Consider a
   sequence of n inserts, followed by an n-way split consisting of one more
   insert and one remove.  Each remove takes O(n) time, for a total time of
   O(n^2) for O(n) operations.) *)
structure NaiveMemoizedQueue :> QUEUE = struct

    type 'a queue = ('a list * 'a list) ref

    fun make_queue (qv as (nil, nil)) = ref qv
      | make_queue (qv as (bs, nil)) = ref (nil, rev bs)
      | make_queue qv = ref qv

    fun new () = make_queue (nil, nil)

    fun insert (b, ref (bs, fs)) = make_queue (b::bs, fs)

    exception Empty

    fun remove (ref (_, nil)) = raise Empty
      | remove (ref (bs, f::fs)) = (f, make_queue (bs, fs))
```

```
end ;

(* Amortized constant-time multi-threaded queues.  Combines specialized
   representation with memoization to achieve amortized constant-time
   behavior, even in the multi-threaded case. *)
structure AmortizedMultiThreadedQueue :> QUEUE = struct

   (* Specialized list representations, with memoization. *)
    datatype 'a special_list_value =
        Nil
      | Cons of 'a * 'a special_list
      | Append of 'a special_list * 'a special_list
      | Reverse of 'a list
    withtype 'a special_list = 'a special_list_value ref

    (* Reverse a list, forming a special_list. *)
    fun revltosl ([], s) = s
      | revltosl (x::xs, s) = revltosl (xs, Cons (x, ref s))

    (* Force a special_list r into Nil/Cons form. *)
    fun inspect (r as ref (Append (xs, ys))) =
        (case inspect xs
           of Nil =>
               let
                   val s = inspect ys
               in
                   r := s; s
               end
            | Cons (x, xs') =>
               let
                   val s = Cons (x, ref (Append (xs', ys)))
               in
                   r := s; s
               end)
      | inspect (r as ref (Reverse xs)) =
        let
            val s = revltosl (xs, Nil)
        in
            r := s; s
        end
      | inspect (r as ref (nil_or_cons)) = nil_or_cons

    type 'a queue = 'a list * int * 'a special_list * int

    fun make_queue (q as (bs, sb, fs, sf)) =
        if sf >= sb then
            q
        else
            (nil, 0, ref (Append (fs, ref (Reverse bs))), sf+sb)

    fun new () = make_queue (nil, 0, ref Nil, 0)

    fun insert (b, (bs, sb, fs, sf)) =
        make_queue (b::bs, sb+1, fs, sf)

    exception Empty

    fun remove (bs, sb, fs, sf) =
        case inspect fs
          of Nil => raise Empty
           | Cons (f, fs') =>
             (f, make_queue (bs, sb, fs', sf-1))
```

```
end;
```

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/regexp.sml] Page 34

```
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

structure RegExp :> REGEXP = struct

  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

  datatype regexp =
    Zero | One | Char of char |
    Plus of regexp * regexp | Times of regexp * regexp |
    Star of regexp

  exception SyntaxError of string

  fun parse_exp ts =
      let
          val (r, ts') = parse_term ts
      in
          case ts'
            of (PlusSign::ts'') =>
               let
                   val (r', ts''') = parse_exp ts''
               in
                   (Plus (r, r'), ts''')
               end
```

```
            | _ => (r, ts')
    end

and parse_term ts =
    let
        val (r, ts') = parse_factor ts
    in
        case ts'
          of (TimesSign::ts'') =>
             let
                 val (r', ts''') = parse_term ts''
             in
                 (Times (r, r'), ts''')
             end
           | _ => (r, ts')
    end

and parse_factor ts =
    let
        val (r, ts') = parse_atom ts
    in
        case ts'
          of (Asterisk :: ts'') => (Star r, ts'')
           | _ => (r, ts')
    end

and parse_atom nil = raise SyntaxError ("Factor expected\n")
  | parse_atom (AtSign :: ts) = (Zero, ts)
  | parse_atom (Percent :: ts) = (One, ts)
  | parse_atom ((Literal c) :: ts) = (Char c, ts)
  | parse_atom (LParen :: ts) =
    let
        val (r, ts') = parse_exp ts
    in
        case ts'
          of nil => raise SyntaxError ("Right-parenthesis expected\n")
           | (RParen :: ts'') => (r, ts'')
           | _ => raise SyntaxError ("Right-parenthesis expected\n")
    end

fun parse s =
    let
        val (r, ts) = parse_exp (tokenize (String.explode s))
    in
        case ts
          of nil => r
           | _ => raise SyntaxError "Unexpected input.\n"
    end
    handle LexicalError => raise SyntaxError "Illegal input.\n"

fun format_exp Zero = [#"@"]
  | format_exp One = [#"%"]
  | format_exp (Char c) = [c]
  | format_exp (Plus (r1, r2)) =
    let
        val s1 = format_exp r1
        val s2 = format_exp r2
    in
        [#"("] @ s1 @ [#"+"] @ s2 @ [#")"]
    end
  | format_exp (Times (r1, r2)) =
    let
        val s1 = format_exp r1
        val s2 = format_exp r2
```

```
      in
          s1 @ [#"*"] @ s2
      end
    | format_exp (Star r) =
      let
          val s = format_exp r
      in
          [#"("] @ s @ [#")"] @ [#"*"]
      end

  fun format r = String.implode (format_exp r)

end

functor Matcher (structure RegExp : REGEXP) :> MATCHER = struct

    structure RegExp = RegExp

    open RegExp

    fun match_is Zero cs k = false
      | match_is One cs k = k cs
      | match_is (Char c) nil _ = false
      | match_is (Char c) (c'::cs) k = (c=c') 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 (r as Star r1) cs k =
        (k cs) orelse match_is r1 cs (fn cs' => match_is r cs' k)

    fun match regexp string =
        match_is regexp (String.explode string)
        (fn nil => true | _ => false)

end

structure Matcher = Matcher (structure RegExp = RegExp)
```

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/memo.sml] Page 35

```
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)

local

    val limit = 100
    val memopad : int option Array.array =
        Array.array (limit, 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 (memopad, n) of
                 SOME r => r
               | NONE =>
                 let
                     val r = p' n
                 in
                     Array.update (memopad, n, SOME r);
                     r
                 end
        else
            p' n

end

signature SUSP = sig
  type 'a susp
  val force : 'a susp -> 'a
  val delay : (unit -> 'a) -> 'a susp
end

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

val t = Susp.delay (fn () => print "hello\n")
val _ = Susp.force t;
val _ = Susp.force t;

signature SUSP = sig
    type 'a susp
    val force : 'a susp -> 'a
```

```
    val delay : (unit -> 'a) -> 'a susp
    val loopback : ('a susp -> 'a susp) -> 'a susp
end

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
  fun loopback f =
      let
          exception Circular
          val r = ref (fn () => raise Circular)
          fun t () = force (!r)
      in
          r := f t ; t
      end
end

datatype 'a stream_ = Cons_ of 'a * 'a stream
withtype 'a stream = 'a stream_ Susp.susp

fun ones_loop s = Susp.delay (fn () => Cons_ (1, s))
val ones = Susp.loopback ones_loop

fun bad_loop s = let val r = Susp.force s in Susp.delay (fn () => r) end
(* val bad = Susp.loopback bad_loop   (* raises Circular *) *)
```

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/seq.sml] Page 36

```
signature SEQUENCE = sig

  type 'a seq = int -> 'a

  val constantly : 'a -> 'a seq               (* constant sequence *)
  val alternately : 'a * 'a -> 'a seq         (* alternating values *)
  val insert : 'a * 'a seq -> 'a seq

  val map : ('a -> 'b) -> 'a seq -> 'b seq
  val filter : ('a -> bool) -> 'a seq -> 'a seq

  val zip : 'a seq * 'b seq -> ('a * 'b) seq
  val unzip : ('a * 'b) seq -> 'a seq * 'b seq
  val merge : 'a seq * '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

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 filter p s n =
        let
            val x = s n
        in
            if p x then x else filter p s (n+1)
        end

    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

open Sequence
```

```
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)

fun fibs_loop s = insert (1, insert (1, map (op +) (zip (drop 1 s, s))))
val fibs = loopback fibs_loop

fun bad_loop s n = s n + 1
val bad = loopback bad_loop
(* val _ = bad 0 *)

(* wires *)

datatype level = High | Low | Undef
type wire = level seq
type pair = (level * level) seq

val Z : wire = constantly Low
val O : wire = constantly High

(* clock pulse with given duration of each pulse *)
fun clock (freq:int):wire = stretch freq (alternately (Low, High))

(* combinational logic *)

infixr **
fun (f ** g) (x, y) = (f x, g y)

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

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
  | gate f w i = f (w (i-1))

val delay : unary_gate = gate logical_nop
val inverter : unary_gate = gate logical_not
val nor_gate : binary_gate = gate logical_nor

(* Flip-flops *)

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
```

```
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 Z)
val R = pulse Low 6 (pulse High 2 Z)
val Q = RS_ff (S, R)
val _ = take 20 Q
val X = RS_ff (S, S)                    (* unstable! *)
val _ = take 20 X

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
```

[http://www.cs.cmu.edu/People/rwh/introsml/samplecode/streams.sml] Page 37

```
Compiler.Control.Lazy.enabled := true;
open Lazy;

datatype lazy 'a stream = Cons of 'a * 'a stream;

val rec lazy ones = Cons (1, ones);

fun shd (Cons (x, _)) = x;
fun stl (Cons (_, s)) = s;
fun lstl (Cons (_, s)) = s;

val rec lazy s = (print "."; Cons (1, s));
val s' = stl s;                              (* prints "." *)
val Cons _ = s';                             (* silent *)

val rec lazy s = (print "."; Cons (1, s));
val s'' = lstl s;                            (* silent *)
val Cons _ = s'';                            (* prints "." *)

fun take 0 s = nil
  | take n (Cons (x, s)) = x :: take (n-1) s;

fun smap f =
    let
        fun lazy loop (Cons (x, s)) = Cons (f x, loop s)
    in
        loop
    end;

fun succ n = n+1;
val one_plus = smap succ;
val rec lazy nats = Cons (0, one_plus nats);

fun sfilter pred =
    let
        fun lazy loop (Cons (x, s)) =
            if pred x then
               Cons (x, loop s)
            else
               loop s
    in
        loop
    end;

fun m mod n = m - n * (m div n);
fun divides m n = n mod m = 0;

fun lazy sieve (Cons (m, s)) = Cons (m, sieve (sfilter (not o (divides m)) s));
val nats2 = stl (stl nats);
val primes = sieve nats2;

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 *)
```

Sample Programs

[Back][Home][Next]

Last edit: *Monday, May 04, 1998 10:53 AM*

A number of example programs illustrating the concepts discussed in the preceding chapters are available in the Sample Code directory.

[Back][Home][Next]