Lecture 2 - Patterns and Functions

David M Kahn *

Summer 2022

1 Patterns

Patterns are ways of referring to the structure of some value. There are (so far in the course) 4 different kinds of patterns to be aware of:

- constants[1] – these patterns match only to the constant that they are
- variables – these match to any value, and bind that value to the variable in the environment
- the wildcard (underscore) – this matches to any value, but does nothing with it
- tuples of other patterns – this matches tuples of the same length where each element matches the pattern in the same position

For example, `(_, 5, x)` is a pattern matching 3-tuples with a middle element of 5, and the use of this pattern binds the third element of the tuple to `x`. This happens to use all the pattern components we have gone over so far, but later in the course we will learn some other patterns.

Multiple patterns are often used to consider different cases of what some value looks like. For such uses, we will find that the patterns are separated with the vertical line symbol `|`. This symbol is not itself a pattern.

One thing to keep in mind is SML patterns do not allow one to reuse variables. Thus, the "pattern" `(x, x)` is simply not allowed – it is *not* used match pairs where both elements are the same. To do that, one would need to write code like the following:

*Adapted from Michael Erdmann's Spring 2022 notes, in turn adapted from a draft of Frank Pfenning

[1]The constants cannot be functions or reals though. This is because there is no easy way to check equality between them - there are infinitely many digits of precision between real numbers, and infinitely many possible inputs between functions (which might not even terminate). This is similar to the restrictions on the use of `=` and `<>`; you can only use constants in a pattern that can be compared with `=` or `<>`.

```
case p of (a, b) =>
  if a=b
  then ...
  else ...
```

1.1 Case Expressions

One place patterns can be used is in case expressions. These can be used to break up data into its constituent components, and also to branch based upon how those components look. It is very general, can actually replace both if-then-else and let-in-end expressions. If you are trying to *use* some compound type, and the type is not a function, you probably will use the type through breaking it up with a case expression.

Expression `case` e_0 `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n where each p_i is a pattern

Typing `case` e_0 `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n `:` t
if e_0`:`t' and, for each i,

- the pattern p_i could apply to values of the type t'
- e_i`:`t given the types of any new variables bound in p_i

Reduction

- `case` e_0 `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n
  $\stackrel{1}{\Longrightarrow}$ `case` e_0' `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` e_n
  if $e_0 \stackrel{1}{\Longrightarrow} e_0'$
- `case` v `of` p_1 `=>` e_1 `|` $\ldots$ `|` p_n `=>` $e_n \stackrel{1}{\Longrightarrow} [env]e_i$ where v is a value, p_i is the first pattern to match v, and env contains any new bindings to the appropriate components of v that p_i induces

Examples

- `case (a, b) of` e_1 `=>` e_2 is a way to break up pairs, since we can now use the variables `a` and `b` to refer to its elements, and is far preferable to projections (e.g. `#1`) stylistically
- `case` e_1 `of x =>` e_2 is extensionally equivalent to (i.e, behaves exactly like) the expression `let val x =` e_1 `in` e_2 `end`
- `case` e_1 `of true =>` e_2 `| false =>` e_3 is extensionally equivalent to (i.e., behaves exactly like) the expression `if` e_1 `then` e_2 `else` e_3

Special Notes It is highly recommended to always put parentheses around case statements. This ensures they are parsed correctly, which otherwise is a major source of errors, especially when case statements are nested.

Case statement patterns do not need to be exhaustive, but they will throw an exception at runtime if no pattern matches the data being cased on. Also, an error will be thrown if further clauses are written past when the clauses' patterns become exhaustive.

1.2 Val Declarations

Val declarations can bind value components directly just like case statements, as long as the pattern uses variables. Otherwise, if the pattern is a constant, the effect of the let binding is a test that the value to be bound equals that constant. Or if the pattern is a wildcard, the value to be bound is evaluated and then discarded.

For example, consider using patterns in the declarations of our let statements:

Expressions `let` p `=` e_1 `in` e_2 `end` where p is a pattern.

Typing The typing is the same as previous let expressions, except the variables typings assumed are those determined by the pattern p and the type of e_1.

Reduction The reducton is the same as previous let expressions, except the bindings made are those determined by the pattern p and the value of e_1. Also, if constants are used and the value of e_1 does not match those constants, then an exception is thrown instead of reduction continuing.

Examples Here are some examples of val declarations using patterns.

- `val x = 5` is our "normal" declaration, but it is already using a pattern – the variable
- `val (a, b) = (1, true)` is another way to break up tuples – the first element is bound to `a` and the second to `b`
- `val _ =` e evaluates e and throws the value away – this has no effect unless e has side effects
- `val (_, false) =` e tests whether the second element of e evaluates to `false` – if not, an error would be thrown at runtime

1.3 Functions

Patterns provide us with a special syntactic sugar (i.e., code shorthand) for automatically casing on the input to a function. For functions like `fn x =>` e, to use this feature, the syntax is the following:

`fn` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n

This is extensionally equivalent to:

`fn x => case x of` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n

2 Functions

2.1 Anonymous Functions

The functions we've been using so far, which look like `fn x =>` e, are called *anonymous*. This is because they do not come with a name. Lacking a name is normal for expressions – we are not bothered by the fact that `(true, 5 + 7)` has no special name, and likewise we should not be bothered that our functions can lack names.

If we want to give an anonymous function a name, we can name it using a declaration like so:

```
1  (*
2   * square : int -> int
3   * REQUIRES: true
4   * ENSURES: square x evaluates to x * x
5   *)
6  val square = fn (x:int) => x * x
```

It is also good practice to include a header for top-level functions, and to annotated the function, as shown. (Note that anonymous functions do not allow you to annotate the return type directly.) It is also good practice to include tests, which have not been shown.

In the expression `fn x =>` e, we call `x` the *formal parameter* of the function, and e the *body*.

2.2 Recursive Functions

While there is no direct expression for recursive functions like there is for anonymous ones, there is a declaration:

```
1  (*
2   * fac : int -> int
3   * REQUIRES: x >= 0
4   * ENSURES: fac x evaluates to x!
5   *)
6  fun fac (x:int) : int =
7    if x = 0
8    then 1
9    else x * fac (x - 1)
```

Again, it is good practice to include a header like that shown. (This time the return type can be annotated directly.)

We can reason about these recursive functions similarly to anonymous functions. They still have formal parameters and bodies, still result in closure values, and still can be applied, but recursive functions also come with the new feature that their name (which is `fac` in the above case) is recursively bound in their body. We can examine this formally using let expressions.

Expressions `let fun f x =` e_1 `in` e_2 `end`

Values The values bound by recursive function declarations are closures just like those of anonymous functions[2].

Typing `let fun f x =` e_1 `in` e_2 `end:`t
if

- $e_1 : t_2$ assuming that $f : t_1 \to t_2$ and $x : t_1$
- $e_2 : t$ assuming that $f : t_1 \to t_2$

This is a lot like how we typed let expressions before, except notice that the declaration of f causes f to be typed in both the usual scope e_2, and also the function body e_1.

Reduction These functions reduce the same way as anonymous functions [3].

Special Notes Recursive functions also allow the use of patterns similarly to anonymous functions, except they require their name to be repeated in each clause. The exact syntax for this feature is the following:
`fun f` p_1 `=` e_1 `|` ... `| f` p_n `=` e_n
This is just syntactic sugar for the declaration:
`fun f x = case x of` p_1 `=>` e_1 `|` ... `|` p_n `=>` e_n

3 Fixity

Fixity is the categorization of the order of functions and arguments in our syntax. It is nice to know, but not necessary for the course.

[2]However, it is useful to extend our notion of closure to include the function name now. That way we can write out the application reduction finitely like so:
$\langle env,$ `fn x => e`$\rangle_f\ v \xrightarrow{1} [env, v/x, \langle env,$ `fn x => e`$\rangle_f/f]\ e$
Otherwise our environment would contain a closure containing the same environment containing the same closure and so on infinitely, since the function recursively refers to itself. This is fine to reason about, but hard to write down, hence the use of now including the function name in recursive closures. However, we usually elide environments, so this detail does not usually matter.

[3]... with the caveat from the previous footnote

3.1 Prefix

By default, the fixity of functions that we define in SML is prefix, meaning the function appears before its arguments. For example, we write the application of the function `square` to the argument 5 as `square 5`, with the function first.

3.2 Postfix

SML has no way of writing postfix functions, which is where the function comes after the argument. However, it is not uncommon to see in mathematics. The common notation for the factorial uses the exclamation point as a postfix function, so that, e.g., 5! means the application of the factorial function to 5.

3.3 Infix

Infix means that the function appears between its arguments. This is commonly used for functions of 2 arguments, like addition, which we write like $5 + 7$. Indeed, you might have considered SML features like addition to be "primitive" operations, but they are in fact functions with special syntactic sugar. If you ever want to refer to the function and not the operation for these special functions, simply put `op` before the desired symbol, like `op+`. Note the type of `op+` is `int * int -> int`; these infix functions always take a pair as an argument. Also note that writing `op` only works for functions whose name does not start with a letter, like `+`.

To make an infix function `f` in SML, you simply write `infix f` as a declaration, and elsewhere define `f` to take a pair as an argument. After declaring `infix f`, the function `f` will be treated as infix.

If you declare `f` before the declaration `infix f`, then `f` should be defined as usual for a function taking a pair as an argument. Additionally, if the function name `f` begins with `op` followed by some non-letter symbol, then `op` will be dropped in later infix use.

```
1  fun plus (x, y) = x + y
2  infix plus
3  val 12 = 5 plus 7
4
5  fun op& (x, y) = x andalso y
6  infix &
7  val false = true & false
```

If you declare `f` to be infix prior to defining f, then you must either define f with an *infix* declaration like `fun x f y =` e or by using our standard declarations while adding `op` to the function name. The latter is only available for function names that can use `op`, meaning those that do not start with a letter.

```
1  infix plus
2  fun x plus y = x + y
3  val 12 = 5 plus 7
4
```

```
5 | infix &
6 | fun op& (x, y) = x andalso y
7 | val false = true & false
```

4 Extensional Equivalence and Reduction Tricks

Extensional equivalence ($\cong$) and reduction-in-any-number-of-steps ($\Longrightarrow$) are closely related, but not identical. Here are some tricks to help reason about them.

- $\cong$ is an equivalence relation, so it satisfies:
  - reflexivity: $e \cong e$
  - symmetry: if $e_1 \cong e_2$ then $e_2 \cong e_1$
  - transitivity: if $e_1 \cong e_2$ and $e_2 \cong e_3$ then $e_1 \cong e_3$
- $\Longrightarrow$ is a partial order relation, so it satisfies:
  - reflexivity: $e \Longrightarrow e$
  - antisymmetry: if $e_1 \Longrightarrow e_2$ and $e_2 \Longrightarrow e_1$, then $e_1 = e_2$
  - transitivity: if $e_1 \Longrightarrow e_2$ and $e_2 \Longrightarrow e_3$ then $e_1 \Longrightarrow e_3$
- $\cong$ is closed under (side-effect free) reduction, so if $e \Longrightarrow e'$ then $e \cong e'$
- $e_1 \cong e_2$ does *not* mean that $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$ – a counter example is `fn x => x + 1` and `fn x => 1 + x`, which are extensionally equivalent but do not reduce to each other
- we can combine the previous properties to learn that if $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, i.e., if two expressions reduce to the same expression, then those two expressions are extensionally equivalent $e_1 \cong e_2$
- similarly, if $e_1 \Longrightarrow e_1'$ and $e_2 \Longrightarrow e_2'$ where $e_1' \cong e_2'$, then $e_1 \cong e_2$
- thanks to referential transparency, we can replace extensionally equivalent subexpressions without affecting extensional equivalence, so that $[e_1/x]\ e \cong [e_2/x]\ e$ if $e_1 \cong e_2$; for example, we know the extensional equivalence `f 3 + (2 * 3)`$\cong$`f 3 + (5 + 1)` since $2 * 3 \cong 5 + 1$, and we do not need to know anything about the function application `f 3` (except that `f:int -> int`)
- for the *values* of most types, extensional equivalence ($\cong$) coincides with normal equality ($=$), but this is not true for the values of types that cannot be compared with `=` like functions and reals
- if $e \hookrightarrow v$ then $e \cong v$, since $e \hookrightarrow v$ entails that $e \Longrightarrow v$

- there exist many examples of an expression $e : t$ that is not extensionally equivalent to any values type t – this happens when e is not valuable, i.e., when e does not evaluate to a value, and instead does something like loop forever or throw an exception