15-150: Principles of Functional Programming

Functions, Patterns, and Substitution

Part II of Values, Expressions, Types, and Declarations*

1 Functions

The previous notes, *Values, Expressions, Types, and Declarations*, developed the basic vocabulary of the course but restricted attention to the primitive types and to tuples built from them.

These notes take up, in order,

- *functions* as values in their own right, with their own typing and evaluation rules;
- *patterns* and *pattern matching*, which give us a convenient way to take apart tuples (and, later, richer data).

1.1 Function Types and the `fn` Expression

For each pair of types t_1 and t_2 there is a function type $t_1 \to t_2$, which is the type of functions that take an argument of type t_1 and (if they terminate at all) produce a result of type t_2. The basic expression that introduces a function is the *lambda expression*, written in SML as

```
fn x : t1 => e
```

Here `x` is a variable, called the *formal parameter* or *bound variable* of the function, and `e` is an expression, called the *body*, in which `x` may occur. Informally, this expression denotes the function that, given an argument v of type t_1, computes its result by evaluating e with v *substituted* for every occurrence of x.

A typing statement for `e` must now be allowed to depend on an assumption about `x`, since `x` may occur in `e`. We already did something like this informally in earlier notes, when we said things like "assuming `x:int`, the expression `x+x` has type `int`." We continue in that spirit: a typing statement may come with a list of assumptions about the types of whichever variables are in scope, and we simply say "$e : t$, assuming $x_1 : t_1, \ldots, x_n : t_n$" (or leave the assumptions out entirely when `e` is closed, as with a whole program).

Typing rule. `fn` $x : t_1 \Rightarrow e$ has type $t_1 \to t_2$, provided $e : t_2$, assuming $x : t_1$ (in addition to whatever other assumptions are already in scope). That is, to check that a lambda expression has an arrow type, we add the parameter's assumed type to the pile of assumptions and check that the body has the result type.

*Compiled and adapted by Dilsun Kaynar from materials developed by several former instructors of 15-150, most notably Stephen Brookes, Michael Erdmann, Robert Harper, and Frank Pfenning, whose lecture notes and textbook chapters form the basis of much of what follows.

1.2 Function Values

A `fn` expression is already a value: it evaluates to itself, no matter how complicated its body is, since nothing can be done with a function until it is actually applied to an argument. Contrast this with `3+4`, which is a well-typed expression but not a value; the expression `fn x:int => 3+4`, on the other hand, *is* a value, even though its body mentions the un-evaluated expression `3+4`. The expression `3+4` is simply never touched until, and unless, the function is applied.

1.3 Naming Functions: the `fun` Keyword

Just as with any other value, we can bind a function to a name using an ordinary `val` declaration:

```
val circ : real -> real = fn r : real => 2.0 * pi * r;
```

This is exactly what the more familiar

```
fun circ (r : real) : real = 2.0 * pi * r;
```

means, for a *non*-recursive function. It is simply more convenient surface syntax.

In general,

```
fun f (x : t1) : t2 = e
```

abbreviates

```
val f : t1 -> t2 = fn x : t1 => e
```

provided `f` does not occur free in `e`. Functions that call themselves need something more (i.e. `val rec` declaration instead of `val`).

2 Application and Call-by-Value Evaluation

Typing rule. $e_1\ e_2 : t_2$, provided $e_1 : t_1 \to t_2$ and $e_2 : t_1$ (under whatever assumptions are in scope for both). Applying a function-typed expression e_1 to an argument e_2 of the expected type t_1 yields the result type t_2, exactly as one would expect.

Evaluation. SML is a *call-by-value* language: to evaluate an application $e_1\ e_2$, we first reduce e_1 to a value, then reduce e_2 to a value, and only then do we perform the call. It is important to note that the argument passed to the function is always a fully-evaluated value, never an unevaluated expression.

$$
\begin{array}{ll}
e_1\ e_2 \overset{1}{\Longrightarrow} e_1'\ e_2 & \text{if } e_1 \overset{1}{\Longrightarrow} e_1', \\
v_1\ e_2 \overset{1}{\Longrightarrow} v_1\ e_2' & \text{if } e_2 \overset{1}{\Longrightarrow} e_2' \text{ (with } v_1 \text{ already a value)}, \\
(\texttt{fn}\ x : t_1 \Rightarrow e)\ v_2 \overset{1}{\Longrightarrow} [v_2/x]e. &
\end{array}
$$

The first two rules are entirely analogous to the evaluation rules given for + and for tuples in the previous notes: reduce the pieces, left to right, until both are values. The third rule is new, and is the heart of this set of notes. It says that calling a function amounts to substituting the argument value v_2 for the parameter x throughout the body e. We call this the *value-substitution rule.*

Notice the restriction to a *value* v_2 on the left of the arrow: substitution is defined for the value-substitution rule to be applicable only once the argument itself is a value, never for an arbitrary unevaluated expression e_2.

In general, for a value v, expression e, and a variable x, we write $[v/x]e$ for the expression obtained from $e$by replacing every *free* occurrence of `x` with `v`.[1] For most of the expression forms we have seen, this is exactly what you would expect: substituting into $e1 + e2$ just means substituting into $e1$ and into $e2$ separately and adding the pieces back together, and similarly for `if`, tuples, and application. For instance,

$$[7/x](x+9) \implies 7+9, \qquad [7/x]x \implies 7, \qquad [7/x]y \implies y \quad (y \text{ is not the same as } x).$$

Note that we wrote $[7/x]x \implies 7$ above using the notation $\implies$ to denote a zero-step reduction. That is, for our purposes, $[7/x]x$ and 7 are two different ways of writing the same expression.

The one case that needs a moment's thought is `fn`. If the expression e we are substituting into already rebinds `x` as its own parameter, such as if e is `fn` $x : t \Rightarrow e_1$, then substitution has nothing to do: every occurrence of `x` inside e_1 refers to this closer, inner binding, not to the outer `x` we are substituting for, so $[v/x]$(`fn` $x : t \Rightarrow e_1$) is `fn` $x : t \Rightarrow e_1$, unchanged. This is the same *shadowing* phenomenon discussed for declarations in the previous notes: a new binding for a name eclipses an old one for the rest of its scope. If instead e is `fn` $y : t \Rightarrow e_1$ for some other variable `y`, we substitute into the body as usual, obtaining `fn` $y : t \Rightarrow [v/x]e_1$.

Example 2.1 (Substitution at work)**.** *Let e be* (*`fn` x : `int` $\Rightarrow x+9$*)*. Then*

$$e\ (3+4) \overset{1}{\implies} e\ 7 \overset{1}{\implies} [7/x](x+9) \overset{0}{\implies} 7+9 \overset{1}{\implies} 16,$$

using, in order: the argument-reduction rule (reducing `3+4` to `7`), the value-substitution rule, the definition of substitution, and finally ordinary integer addition.

2.1 Declarations, Revisited

The previous notes described the meaning of a `let` expression informally. Substitution lets us say exactly what that means. For a single, non-recursive binding,

$$\texttt{let val } x : t = e_1 \texttt{ in } e_2 \texttt{ end} \overset{1}{\implies} \texttt{let val } x : t = e_1' \texttt{ in } e_2 \texttt{ end}, \quad \text{if } e_1 \overset{1}{\implies} e_1',$$

$$\texttt{let val } x : t = v_1 \texttt{ in } e_2 \texttt{ end} \overset{1}{\implies} [v_1/x]e_2.$$

That is, a `val` declaration inside a `let` is, from the point of view of evaluation, nothing more than the value-substitution rule again: evaluate the bound expression to a value, then substitute that value for the bound name throughout the body, exactly as with a function's parameter. A top-level `val` declaration behaves the same way with respect to "the rest of the program," which plays the role of e_2.

This also finally explains the static scoping example from the previous notes. There, we had

[1] A variable x is said to occur free in an expression e, if e does not contain a binder for x such as `fn` $x \Rightarrow \cdots$ or a pattern involving x. For example, the variable x occurs free in the expression $x+1$, but it is bound in the expression `fn` x `=>` $x+1$.

```
val pi : real = 3.14;
fun circ (r : real) : real = 2.0 * pi * r;
val pi : real = 3.14159; (* does not affect circ *)
```

and observed that `circ` keeps using the *old* value of `pi`. Substitution tells us why: due to the substitution step for the *outer* `val pi` declaration, applied to everything that follows it, including the declaration of `circ`, the value bound to `circ` is, in effect, `fn r:real => 2.0 * 3.14 * r`, where the value `3.14` is already "baked into" the function value, before the second declaration of `pi` is ever reached. A later substitution of `pi` by `3.14159` has nothing left to act on inside `circ`'s already-formed body, precisely because that substitution only reaches the expressions that come *after* it, and `circ` was already fully formed before that point.

2.2 Recursive Functions

The desugaring of `fun` given in Section 2.3,

```
fun f (x : t1) : t2 = e
```

to

```
val f : t1 -> t2 = fn x : t1 => e
```

only makes sense when `f` does not occur free in `e`. A `val` declaration evaluates its right-hand side *before* any binding for the declared name exists, so if `e` mentions `f`, that occurrence of `f` would be unbound. SML resolves this with a different declaration form, *recursive value declaration*:

```
val rec f : t1 -> t2 = fn x : t1 => e
```

in which `f` *is* understood to be in scope already, inside `e` itself. What value should be bound to `f`? Call it F. Since `f` is meant to refer to F everywhere inside the body, F should be the function value obtained by first substituting F itself for every free occurrence of `f` in `e`, and only then abstracting over the parameter:

$$F \;=\; \texttt{fn}\ x : t_1 \Rightarrow [F/f]e.$$

This is a *fixed-point equation*: F is defined in terms of itself. We will not attempt, in this course, a construction of F from first principles. We simply take it as given that such an F exists for every well-typed recursive function declaration. What matters for our purposes is the resulting *evaluation rule* for calling a recursive function, obtained by combining this equation with the ordinary value-substitution rule for application:

$$F\ v \;\overset{1}{\Longrightarrow}\; [v/x]\big([F/f]e\big).$$

For the purposes of actually carrying out evaluations by hand, we can drop the name F entirely and work one level more concretely: rather than naming "the value bound to `f`" and reasoning about it abstractly, we simply substitute the function's own defining code, the whole `fn`-expression for `f` wherever it occurs, every time a recursive call is unfolded.

Recall

```
fun fact (n : int) : int = if n = 0 then 1 else n * fact (n - 1)
```

Evaluating `fact 2` *using the rule above means substituting the entire definition of* `fact` *for every occurrence of* `fact` *in its own body and substituting* `2` *for* `n`*:*

$$\texttt{fact 2} \Longrightarrow \texttt{if 2 = 0 then 1 else 2 * (fact 1)} \Longrightarrow \texttt{2 * (fact 1)},$$

where `fact 1` *still denotes a call to the very same recursive function. Unfolding it in turn, the same way, gives*

$$\texttt{fact 1} \Longrightarrow \texttt{if 1 = 0 then 1 else 1 * (fact 0)} \Longrightarrow \texttt{1 * (fact 0)},$$

and unfolding `fact 0` *once more gives* `1` *directly. Substituting back up through the pending multiplications,*

$$\texttt{fact 2} \Longrightarrow \texttt{2 * (1 * 1)} \Longrightarrow \texttt{2}.$$

3 Patterns and Pattern Matching

A *pattern* is a syntactic template that describes the shape of a value we expect to see, while simultaneously naming the pieces we want to extract from it. Restricted (for now) to the primitive types and tuples, patterns are given by

$$p \ ::= \ x \ \mid \ \_ \ \mid \ c \ \mid \ (p_1, \ldots, p_n)$$

where x ranges over variables, _ is the special *wildcard* pattern, and c ranges over constants (integer, boolean, and similar literals). [2]

Evaluation matches a pattern against an actual *value* v. This either fails, or succeeds and produces bindings of values for the pattern's variables, which are exactly the bindings that substitution needs in order to evaluate whichever branch the pattern guards.

- A variable pattern x always matches, binding x to v itself; in substitution terms, this contributes $[v/x]$.
- The wildcard _ always matches too, but contributes no binding at all.
- A constant pattern c matches only if v is that same constant, and otherwise fails; either way it contributes no binding.
- A tuple pattern $(p_1, \ldots, p_n)$ matches a tuple value $(v_1, \ldots, v_n)$ by matching each p_i against the corresponding v_i in turn; the match as a whole succeeds, with all the individual bindings combined, exactly when every component match succeeds, and fails as soon as any one of them does.

3.1 The `case` Expression

Patterns are put to use in a `case` expression,

```
case e of
    p1 => e1
  | p2 => e2
  | ...
  | pn => en
```

[2] We require that a variable occur at most once in a given pattern (so (x, x) is not a legal pattern), which will let us treat all the variable bindings a pattern produces as independent of one another.

Typing rule. The whole `case` expression has type t', provided `e` has some type t that every pattern p_i can be checked against, and every branch e_i has that same type t', assuming whatever additional bindings its pattern p_i introduces, on top of whatever assumptions were already in scope. That is, just as in the typing rule for `if`, every branch must agree on a single result type.

Evaluation. The expression `e` is evaluated to a value first; the patterns are then tried *in order*, top to bottom, and the first one that matches determines the result:

$$(\texttt{case}\ e\ \texttt{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \xrightarrow{1} (\texttt{case}\ e'\ \texttt{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \quad \text{if } e \xrightarrow{1} e',$$

$$(\texttt{case}\ v\ \texttt{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) \xrightarrow{1} \sigma(e_i)$$

where i is the *least* index such that p_i matches v, yielding substitution σ, and $\sigma(e_i)$ denotes the result of applying every substitution in σ to e_i. Expressions in other branches are left unevaluated. If no p_i matches v at all, evaluation raises the built-in exception `Match` rather than producing a value.

Example 3.1 (Tuple patterns)**.** *Given a pair* `p : int * int`*,*

```
fun swap (p : int * int) : int * int =
  case p of
      (x, y) => (y, x)
```

Evaluating `swap (3, 5)` *proceeds by evaluating the argument to the value* `(3,5)`*, matching it against the pattern* `(x,y)` *to obtain the substitution* $[3/x, 5/y]$*, and then evaluating* $[3/x, 5/y](y, x) = (5, 3)$.

3.2 Clausal Function Definitions

The multi-clause style of function definition such as

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

is syntactic sugar for a single function whose body is a `case` expression over its argument, one clause per pattern:

```
fun fact (n : int) : int =
  case n of
    0 => 1
  | n => n * fact (n - 1)
```

```
fun f p1 = e1
  | f p2 = e2
  | ...
  | f pn = en
```

abbreviates

```
fun f x = case x of p1 => e1 | p2 => e2 | ... | pn => en
```

for a fresh variable `x` not occurring elsewhere.

A remark on exhaustiveness. A set of clauses is called *exhaustive* if every value of the argument's type matches at least one pattern in the list; SML will warn about a clausal definition (or a `case` expression) that is not exhaustive, since such a function can raise `Match` on some inputs. It is good practice to arrange clauses so that exhaustiveness is easy to see by inspection.

4 Extensional Equivalence, Revisited

Two functions $e, e' : t_1 \to t_2$ are extensionally equivalent when, informally, applying them to equivalent arguments of type t_1 produces equivalent results of type t_2. Here, "equivalent results" means: both computations diverge, or both raise the same exception, or both terminate and produce the same (equivalent) value.

Referential transparency continues to hold at function types: one extensionally equivalent function may always be substituted for another without changing the observable behavior of a program, since doing so cannot be detected by any sequence of applications. This is precisely what justified the claim, in the previous notes, that `circ` and `circ'` (one defined directly, the other via a `local` declaration factoring out `2.0 * pi`) are extensionally equivalent: both send every real number `r` to the same value, even though they are built from syntactically different expressions.

5 Summary

- Functions are values. The expression `fn` $x : t_1 \Rightarrow e$ has type $t_1 \to t_2$ whenever $e : t_2$ under the added assumption $x : t_1$, and it evaluates to itself; `fun` is convenient sugar for naming such a value with a `val` declaration.
- Application is call-by-value: both e_1 and e_2 are reduced to values before the call is made, and the call itself is performed by the *value-substitution rule*, (`fn` $x : t_1 \Rightarrow e$) $v \xrightarrow{1} [v/x]e$.
- Substitution, $[v/x]e$, replaces free occurrences of x in e with v and is applied to `let`/`val` as well as to function application.
- Patterns (x, _, constants, and tuples, for now) are matched against values by the partial operation match(p, v), which either fails or produces a substitution; the `case` expression tries its patterns in order and substitutes the resulting bindings into whichever branch matches first, and clausal function definitions are sugar for a single `fn` together with a `case`.
- Extensional equivalence at function types refines the earlier definition: two functions are extensionally equivalent if applying them to equivalent arguments always produces equivalent results.