S1.4: Expressions, Values, Evaluation

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

Before introducing new values (and types) we need to understand the relationship between

- values,
- expressions,
- and the process of expression evaluation.

Expressions and values

Some examples, as we have seen before.

Expressions		Values
`1 + 2 * 3`	⟶	`7`
`[1+2; 4*5]`	⟶	`[3; 20]`
`List.tail [1;2;3]`	⟶	`[2;3]`
`add 3`	⟶	`(fun x -> x + 3)`[1]
`4`	⟶	`4`
`[1;2]`	⟶	`[1;2]`
`"Hel" @ "lo"`	⟶	`"Hello"`

- All values are expressions. These are values.
  `4, [1;2;3], true, "Hello"`
- Not all expressions are values. These are not values.
  `1+3, [1,2] @ [3], 3 > 2, "He"@ "llo"`
- The expressions above evaluate to the values above.

[1] (There was a typo in original slides: was `(fun x -> x + 1)`)

Evaluation

- In OCaml and other functional languages, computation is done by expression evaluation.
- This process transforms expressions into values.

  The resulting values are the answers are programs are intended to provide.
- Repeated replacement of "reducible expressions" or (redexs) by their values until there are no more reducible expressions to evaluate.
- An expression with no more reducible expressions is called a value.

Sample simple evaluations

```
   1 + 2 * (3 + 4)
-> 1 + (2 * 7)
-> 1 + 14
-> 15

   let x = 3 + 4 in x * 4
-> let x = 7 in x * 4
-> let x = 7 in 7 * 4
-> 7 * 4
-> 28
```


Referential transparency

- This is a viable mechanism for evaluation because of the concept of **referential transparency**.
- An expression is referentially transparent if it can be replaced with its value without changing the program's behavior.
- Replacing `(3 * 4)` by `12`
  in `2 + (3 * 4)`
  yields `2 + 12`.
  This has the same value as the initial expression.
- The C expression `++x` is not referentially transparent, it is **referentially opaque**.

  This expression has a side effect on the variable `x`.

- Because of referential transparency, we have some flexibility in deciding which redex to reduce to take a step.
- In fact, for referentially transparent expressions the order doesn't matter.
- Meaning, any order will yield the same value.
- Though some orders may not yield a value.
  e.g. `if 5 > 2 then 4 else 2 / 0`
- We'll be guided by our intuition for now.

  Let's consider a few examples that are more interesting than those above.

A few examples

```
   let add x y = x + y in 2 * add 3 (5 + 5)
-> let add x y = x + y in 2 * add 3 10
-> let add x y = x + y in 2 * (3 + 10)
-> 2 * (3 + 10)
-> 2 * 13
-> 26
```

(Note how function bodies (*e.g.*`(3 + 10)`) are wrapped in parenthesis. We must maintain the correct structure of the expression during evaluation.)

Note that `ocaml` (and the `utop` front-end to `ocaml`) will display values.

So we can use `utop` to check our work.

A example of incorrect evaluation

If we ignore the note above and do not write function bodies (when inlined into an expression) in parenthesis then we may get an incorrect result.

For example:

```
   let add x y = x + y in 2 * add 3 (5 + 5)
-> let add x y = x + y in 2 * add 3 10
-> let add x y = x + y in 2 * 3 + 10 ← WRONG MOVE, parenthesis missing
-> 2 * 3 + 10
-> 6 * 10 ← Ack! here we see the problem
-> 16
```

By not keeping `3 + 10` grouped as an expression (in parenthesis) we incorrectly multiply 2 * 3.

A more detailed version

```
   let add = fun x -> (fun y -> x + y) in 2 * (add 3) (5 + 5)
-> let add = fun x -> (fun y -> x + y) in 2 * (add 3) 10
-> let add = fun x -> (fun y -> x + y)
   in 2 * (((fun x -> (fun y -> x + y)) 3) 10)
-> 2 * (((fun x -> (fun y -> x + y)) 3) 10)
-> 2 * ((fun y -> 3 + y)) 10)
-> 2 * (3 + 10)
-> 2 * 13
-> 26
```

This may be "more accurate" but our aim here is

- not to be overly precise about evaluation order or details
- but to see expression evaluation as the process by which computation is carried out.

Some guidelines

We have a few guidelines that we'll follow for now.

Later, we'll see how the decisions we make lead to the notion of eager evaluation (what is done in OCaml and other languages) and lazy evaluation (what is done in Haskell).

Some guidelines

For now, we'll be guided by our intuition about how languages like OCaml work.

- Only replace names in the body of a let-expression with values.
  Thus, the binding expression must be reduced to a value first.
- We don't need to replace names with functional values (as we did in the more detailed version above), we'll treat these function calls a bit differently.

  We replace a function call with the function body, instantiated with values replacing function arguments.

  Thus, function arguments get reduced to values before the function is applied.
- We replace a let expression with its body when the bound name no longer appears in the body - that is, after the value has been instantiated.
- We don't perform reductions in the body of a lambda expression.
  (This one is really a hard-and-fast rule, not a flexible guideline.)

Declarations

- The evaluation of
  `let add x y = x + y in 2 * add 3 (5 + 5)`
  was a bit tedious above.
- So we may consider the declarations of functions as we've seen in our OCaml sessions.
  `let add x y = x + y ;;`
- then

```
   2 * add 3 (5 + 5)
-> 2 * add 3 10
-> 2 * (3 + 10)
-> 2 * 13
-> 26
```

- This will make things a bit easier.

Exercise S1.4: #1: Evaluation

Write the step-by-step transformation of the following expression into a value.

- `let x = 1 + 2 in let y = x + 3 in x + y`

If you're online: put this in a file with a `.md` extension and put your evaluation trace in a code block, as indicated by 3 back-tick characters (on the same key as ~):

```
let x = 1 + 2 in let y = x + 3 in x + y
...
```


Recursive functions

Recall what we did with `sum_to` and `adder` in `getting_started.ml`.

Now consider

```
let rec even_length' (xs: 'a list) : bool =
  match xs with
  | [] -> true
  | n::ns -> not (even_length' ns)
```

and the expression

```
even_length (1 :: 2 :: 3 :: [])
```

What does this evaluation look like?

Let's show that in `lists_tuples.ml` where this function is defined.

Exercise S1.4: #2: Evaluation

Write the step-by-step transformation of the following expression into a value.

- Assuming this declaration

```
let rec sum lst = match lst with
  | [] -> 0
  | x::xs -> x + sum xs
```

evaluate `sum (1::2::3::[])`

λ-expressions

We can still handle lambda expressions

```
   1 + (fun x -> x + 1) 5
-> 1 + (5 + 1)
-> 1 + 6
-> 7
```

```
   1 + (fun x -> x + x) (4 * 5)
-> 1 + (fun x -> x + x) 20
-> 1 + (20 + 20)
-> 1 + 40
-> 41
```


Patterns

- In a `match...with` expression, the patterns are essentially values with "holes."
- These holes are either
  - names or
  - underscores (`_`)
- Consider again the application of `sum` above.

Reconsider `sumr` and `suml`

```
let rec foldr (f: 'a -> 'b -> 'b) (lst: 'a list) (base: 'b) : 'b =
  match lst with
  | [] -> base
  | x::xs -> f x (foldr f xs base)

let sumr xs = foldr (+) xs 0
```

```
sumr (1::2::3::[])
-> foldr (+) 1::2::3::[] 0
-> (+) 1 (foldr + 2::3::[] 0)
-> (+) 1 ((+) 2 (foldr + 3::[] 0))
-> (+) 1 ((+) 2 ((+) 3 (foldr + [] 0)))
-> (+) 1 ((+) 2 ((+) 3 (0)))
-> (+) 1 ((+) 2 3)
-> (+) 1 5
-> 6
```


Reconsider `sumr` and `suml`

Or with `+` as an infix operator

```
let rec foldr (f: 'a -> 'b -> 'b) (lst: 'a list) (base: 'b) : 'b =
  match lst with
  | [] -> base
  | x::xs -> f x (foldr f xs base)

let sumr xs = foldr (+) xs 0
```

```
sumr (1::2::3::[])
-> foldr (+) 1::2::3::[] 0
-> 1 + (foldr + 2::3::[] 0)
-> 1 + (2 + (foldr + 3::[] 0))
-> 1 + (2 + (3 + (foldr + [] 0)))
-> 1 + (2 + (3 + 0))
-> 1 + (2 + 3)
-> 1 + 5
-> 6
```


Exercise S1.4: #3: Reconsider `sumr` and `suml`

Now evaluate `suml (1::2::3::[])` using the following definitions.

```
let rec foldl (f: 'b -> 'a -> 'b) (accum: 'b) (lst: 'a list) : 'b =
  match lst with
  | [] -> accum
  | x::xs -> foldl f (f accum x) xs

let suml xs = foldl (+) 0 xs
```

Computation by expression evaluation

- Programs don't run by taking actions that produce observable results
- They are expressions that are evaluated to a value.
- The choice of `=` in `let` expressions and declarations is intentional.

  Above, we replaced expressions with other expressions based on the equalities specified in the declarations.
- (We will later see side effects, like `print`, in OCaml, but our focus on expression evaluation.)

With these notions of expressions and values clarified, we can consider inductive values and types in the S1.5 slides.