S1.3: Expressions, Values, Evaluation

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2018

Before introducing new values (and types) we need to understand the relationship between values, expressions, and the process of expression evaluation.

Expressions

Some examples:

- `1 + 2 * 3`
- `[1+2; 4*5]`
- `List.tail [1;2;3]`
- `add 3`
- `4`
- `[1;2]`
- `"Hello"`

Values

Some examples:

- `1`
- `25`
- `[1;2;3]`
- `"Hello"`
- `(fun x -> x + 1)`

Values and expressions

- 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 do the values (further) above.

Evaluation

- This process transforms expressions into values.
- Repeated replacement of "reducible expressions" or (redexs) by their values until there are no more redexs.
- An expression with no more redexs 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
-> let x = 7 in 28
-> 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**.

- 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. 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)
-> let add x y = x + y in 2 * 13
-> let add x y = x + y in 26
-> 26
```

(Note how function bodies 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 `ocaml` to check our work.

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).

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 won't replace names with functional values, 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 a value once its body is a value.
- We can also replace a let-expression with its body if the declared name no longer appears in the body.
- We don't perform reductions in the body of a lambda expression.

Declarations

- The evaluation of
  `let add x y = x + y in 2 * add 3 (5 + 5)`
  was a bit tedious above.
- So let's consider the declarations that we've seen in our OCaml sessions.
  `let add x y = x + y ;;`
- ```
    2 * add 3 (5 + 5)
  -> 2 * add 3 10
  -> 2 * (3 + 10)
  -> 2 * 13
  -> 26
  ```
- This will make things a bit easier.

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 `rev` above.
```

Inductive values and types

With these notions of expressions and values clarified, we can consider inductive values and types.