Evaluation of programs

Ziyan Maraikar

July 8, 2014

Table of Contents

Defining a language

A programming language, has three aspects to it:

Syntax rules for structuring language elements to create a valid program.

Type system rules for computing the the type of an expression.

Semantics rules for computing the *result* of a program.

Informal vs. formal definitions

So far we have introduced Ocaml informally. We can formally define each aspect using:

Syntax grammar rules.

Type system typing rules.

Semantics rewrite (substitution) rules.

For now, we will define only semantics formally.

Why formal semantics?

- When learning a language we need a mental model of what happens during execution so we can "run a program in our head."
- A formal model lets us reason *mechanically* and *precisely* on paper.
- Functional programs can be described using a fairly simple set of rules.

Algebraic substitution (rewriting)

You have been using substitution since school, e.g.

- Substitute $x = y^2 + 2y$ in $2x^2 + x - 1$
- The rules to compute $\frac{d}{dx}(x+1)(x^2+x)$

We will define a set of rules like those used for differentiation, for *evaluating programs*.

Evaluating variable bindings

The general form of a *let* expression is,

$$\texttt{let } x = v \texttt{ in } e_x$$

We evaluate this expression by substituting the value v for the variable x in e_x. Formally we write this as,

$$\texttt{let } x = v \texttt{ in } e_x$$
$$\equiv e[v/x]$$

A *local* variable definition makes explicit which variable we are substituting for.

Example

```
let x=3 mod 2 in
let y=3/2 in
    x*x + x*y + y*y
≡ let x=3 mod 2 in
      let y=1 in
        x*x + x*y + y*y
≡ let x=3 mod 2 in
        x*x + x*1 + 1*1
≡ let x=1 in
        x*x + x*1 + 1*1
≡ 1*1 + 1*1 + 1*1
```

Note how we begin evaluation from the innermost scope.

Exercise

```
let a=(not true)||false in
let y=10.0 in
    y > 0. && a
```

```
let x=1 in
let x=2 in
    x * x
```

Remember the scoping rule!

Evaluating functions

The rule for function application is very similar to variable substitution.

We apply a function by substituting *arguments* for the corresponding parameters in its body.

`let f x =` e_x `in f a`

$\equiv e[a/x]$

Again, a *local* function makes explicit which function we are applying.

Example

```
let square  x =
    x * x in
let quad x =
    square (square x) in
quad 2
```

$\equiv$ `square (square 2)`

$\equiv$ `square (2 * 2)`

$\equiv$ `4 * 4`

Exercise

```
let sos x y =
    (square x) + (square y) in
sos 2 3

let circle_area r =
    let pi = 3.142 in
        pi *. r *. r in
circle_area 1.0
```

Table of Contents

The syntax of if

`if` e_b `then` e_T `else` e_F

The result of an *if expression*[1] is the value of e_T when the boolean condition e_b is **`true`** or the value of e_F otherwise.

```
let sign n =
  if n>0 then 1
  else if n=0 then 0
  else -1
```

Remember that equality is written `=`. Don't use the `==` operator, which has a different meaning.

[1] unlike the C if statement

IF SEMANTICS

`if` **`true`** `then` e_T `else` e_F
$\equiv e_T$

`if` **`false`** `then` e_T `else` e_F
$\equiv e_F$

Exercise

- Write a function **abs** that prints the absolute value of a number. Show the evaluation of `abs -10`.
- Write a function that takes a mark and prints the corresponding grade. The grades are A:100–80, B: 60–79, C: 40–59, and F: less than 40.