The substitution model

This reading is a summary of the rules of the substitution model, discussed starting in lecture 2.

Overview

The substitution model is a way of manually evaluating OCaml expressions. It's similar to, but not identical to, what the computer does when it evaluates OCaml code. The point of learning the model is so that you have a good mental framework for how OCaml code evaluates.

> **Note**
>
> At some point, this model will not be sufficient to explain all the features of OCaml that we will use (for instance, updating variables in imperative programming), so we will define a new model: the *environment model*. That model will share many features with the substitution model, but its handling of name lookup will be much more precise.
>
> The substitution model is adequate to understand the purely functional subset of OCaml we will use for the first half of the course.

Desugaring and precedence

Before evaluating any OCaml code using the substitution model, the code should first be *desugared*, and then operator *precedence* should be made explicit using parentheses.

Desugaring

The main kind of desugaring we need is to replace expressions of this form:

```
let f x y = ...
```

with this:

```
let f = fun x y -> ...
```

After desugaring, all `let` expressions are "simple" *i.e.* they bind a single *name* (here, `f`) to a *value* (the result of evaluating the expression on the right-hand side of the `=`; here, `fun x y -> ...`).

> **Note**
>
> If we are really being picky, we could desugar this further to:
>
> ```
> let f = fun x -> fun y -> ...
> ```
>
> but we don't require that. This is because of function *currying*, which is the way that OCaml interprets functions with multiple arguments.
>
> We recommend that you don't do this in your written evaluations, unless you have to for some reason.

Precedence

Operator precedence should be made explicit by wrapping parentheses around nested operator expressions. For instance, this code:

```
1 + 2 * 7
```

would become:

```
1 + (2 * 7)
```

The full precedence table of OCaml operators is here. In most cases, it will be what you would expect.

This step is done so that there is never any question about what the operands of an operator are.

The basic rule

To evaluate an OCaml expression:

1. evaluate the *operands* of the expression
2. evaluate the *operator* or *function* of the expression
3. *apply* the operator/function to the evaluated operands

> **Note**
>
> OCaml actually evaluates operands from right-to-left, in contrast to most languages, which evaluate operands from left-to-right. Evaluation order is *not* part of the substitution model, and shouldn't make any difference.
>
> Similarly, you can evaluate the operator/function before or after evaluating the operands, and it won't make any difference in this model.

The basic rule is used when evaluating a function call or an operator expression. Other expressions (such as `let`, `fun` and `if` expressions) do not use the basic rule; they have their own evaluation rules. Such expressions are called *special forms*.

Specific cases of the basic rule

- Numbers evaluate to themselves: `10` → `10` (In fact, any literal data value evaluates to itself; that's why it's called "literal". For now, we are mostly working with numbers.)
- Primitive (built-in) functions evaluate to the corresponding internal procedure. We can write this as either: `+` → `[primitive function +]` or just `+` → `+`
- Variables that have been previously defined are "looked up" (in some unspecified way) and evaluate to the value that they were previously bound to. (If they weren't bound to a value previously, it's an error.)

`let`-bound names

There are two kinds of `let`-bound names: top-level and local. They evaluate differently.

Top-level `let` expressions

A top-level `let` expression[1] looks like this:

```
let <var> = <expr>
```

for some variable name `<var>` and some expression `<expr>`. For instance:

```
let x = 2 + 3
```

To evaluate this, you

- evaluate the expression to the right of the `=` sign (here, `2 + 3`, which evaluates to `5`);
- "make an association" or "bind" the name (`x` here) to the value of the evaluated expression (`5`). The details of how to make this association aren't important for now, though we'll revisit this later.

In this case, we make an association between the name `x` and the value `5`. We can also say that we bind `x` to `5`.

After this expression is evaluated, the name `x` will be bound to the value `5` for the rest of the evaluated code, unless `x` is given a new binding with a new top-level `let` expression.

> **Note**
>
> A new binding for `x` would be something like:
>
> ```
> let x = 42
> ```
>
> From then on, any reference to `x` would get the new value. This is *not* the same as assignment in an imperative language, because it doesn't overwrite the old binding. Instead, this code creates a *new* binding to `x` which "shadows" the old binding, but the old binding still exists, and in some cases can still have effects.

Local `let` expressions

A "local" `let` expression is a `let` expression where a name is bound to a value and then immediately used in another expression, such as this:

```
let x = 2 + 3 in x * x
```

(Note the use of the `in` keyword; that's how you know it's a local `let` expression.)

In this case, the name `x` has a meaning inside the body of the expression (the `x * x` part) but not outside. To evaluate this, you:

- evaluate the *binding expression* (here, `2 + 3`, which evaluates to `5`),
- bind the name to the value (here, bind `x` to `5`),

- and evaluate the *body expression* (here, `x * x`).

In the body expression, of course, you can use the name `x`, which has a value (`5`). Outside of this expression, the name `x` either has no value (if it wasn't bound before) or has the value it previously had (if it was). Remember: you aren't changing a previous binding (to `x`), you're creating a new one that is used only in a single expression.

`if` expressions

`if` expressions consist of three subexpressions:

- the *test* subexpression (between the `if` and the `then` keywords)
- the `then` subexpression (between the `then` and the `else` keywords)
- the `else` subexpression (everything following the `else` keyword)

They have the following evaluation rule.

1. Evaluate the test subexpression.
2. If the test subexpression evaluates to `true`, evaluate the `then` subexpression.
3. If the test subexpression evaluates to `false`, evaluate the `else` subexpression.

One consequence of this is that you never evaluate both the `then` and `else` subexpressions. This is why `if` can't be a function (it's not just the syntax!)[2]

`if` without `else`

When writing purely functional code in OCaml, `if` expressions always have both a `then` subexpression and an `else` subexpression. (These are sometimes called the "then clause" and the "else clause".) With respect to the substitution model, we will always have `else` subexpressions in `if` expressions.

To learn more about how `if` expressions without `else` subexpressions work, see the OCaml cheat sheet.

`fun` expressions

A `fun` expression (like `fun x y -> x + y`) represents an anonymous function. In the substitution model, a `fun` expression is usually the result of desugaring a regular function definition, like

```
let f x y = x + y
```

which desugars to:

```
let f = fun x y -> x + y
```

`fun` expressions consist of two parts:

- the *formal parameters* of the function (here, `x` and `y`),
- the *body* of the function (here, `x + y`).

`fun` expressions are trivial to evaluate: you basically just leave them alone. For this reason, in written evaluations you can write:

```
fun x y -> x + y  -->  fun x y -> x + y
```

or just:

```
fun x y -> x + y  -->  itself
```

Function application

Function application is at the heart of the substitution model. There are two cases.

Applying built-in functions or operators

Applying a built-in function or operator is simple: you just do it. For instance:

```
Evaluate: 2 + 3
  2 --> 2
  3 --> 3
  + --> [primitive function +]
  apply + to 2, 3 --> 5

Evaluate: abs (-10)
  -10 --> -10
  abs --> [primitive function abs]
  apply abs to -10 --> 10
```

In some cases we'll allow you to shorten lines like

```
abs --> [primitive function abs]
```

to just:

```
abs --> abs
```

Don't do this unless we explicitly say it's OK.

Applying user-defined functions: substitution

Now we come to the "substitution" part of the substitution model. It happens when you are applying a user-defined function (which means a `fun` expression, or a function which gets desugared to a `fun` expression) to its arguments (which have already been evaluated to values, so they aren't expressions anymore).

The rules for applying user-defined functions are:

1. *Substitute* the function argument variables (formal parameters) with the values given in the call everywhere they occur in the function body.
2. Evaluate the resulting expression.

For the most part, the substitution process is straightforward. For each function parameter, you substitute the argument value for the function parameter name in the function body to get the substituted expression, which you then evaluate. For instance, this expression:

```
(fun x y -> x + y) 2 3
```

evaluates as follows:

```
Evaluate: (fun x y -> x + y) 2 3
  2 --> 2
  3 --> 3
  (fun x y -> x + y) --> itself
  apply (fun x y -> x + y) to 2, 3
    substitute 2 for x, 3 for y in x + y --> 2 + 3
    evaluate: 2 + 3
      2 --> 2
      3 --> 3
      + --> [primitive function +]
      apply + to 2, 3 --> 5
```

You can even evaluate inside a complex expression (for instance, an `if` expression):

```
Evaluate (fun x y -> if x > y then x + y else x - y) 2 3
  2 --> 2
```

```
  3 --> 3
  (fun x y -> ...) --> itself
  apply (fun x y -> ...) to 2, 3
    substitute 2 for x, 3 for y in if x > y then x + y else x - y
    --> if 2 > 3 then 2 + 3 else 2 - 3
    evaluate: if 2 > 3 then 2 + 3 else 2 - 3
      if is a special form; evaluate 2 > 3
        2 --> 2
        3 --> 3
        > --> [primitive function >]
        apply > to 2, 3 --> false
      For false case, evaluate else clause: 2 - 3
        2 --> 2
        3 --> 3
        - --> [primitive function -]
        apply - to 2, 3 --> -1
```

Notice that the substitution goes right inside the `if` expression, regardless of which branch of the `if` will eventually end up being evaluated. (Here we also see that for long `fun` expression bodies, we can abbreviate them with `...`.)

> **Note**
>
> On the other hand, we can't substitute inside a nested `fun` expression in all cases. See below for more details.

Recursion

Recursion doesn't require any special treatment in the substitution model, except that you have to replace a function name with the correct definition of that function when looking up the name. This is usually obvious, and probably won't cause you any problems. It can get weird in pathological cases where you redefine a name. Here's a silly example:

```
let factorial n = 0   (* obviously wrong *)

(* Redefine factorial. *)
let factorial n =     (* oops, forgot the "rec" *)
  if n = 0 then
    1
  else
    n * factorial (n - 1)
```

When evaluating this function, the "recursive" call to `factorial` will actually pick up the previous definition of `factorial`, and `factorial` will return `0` for any input but `0`. The solution is to

include the `rec` in the second definition:

```
let factorial n = 0   (* obviously wrong *)

(* Redefine factorial. *)
let rec factorial n =
  if n = 0 then
    1
  else
    n * factorial (n - 1)
```

Now everything will work correctly because the `rec` tells OCaml that any references to `factorial` inside the body of `factorial` represent the function being defined.[3]

Of course, this normally doesn't come up because you normally don't redefine functions in a file, but something like this could happen when defining functions interactively inside the OCaml interpreter. This can lead to very peculiar bugs.

Nested `fun` expressions and shielding

The substitution model has to be adjusted to deal with nested `fun` expressions. Here's a simple example:

```
let f x =
  fun x -> x + x
```

How would you evaluate `f 3`? Here's a first (wrong) attempt:

```
Evaluate (f 3)
  3 --> 3
  f --> fun x -> fun x -> x + x
  apply (fun x -> fun x -> x + x) to 3
    substitute 3 for x in (fun x -> x + x)
    --> fun 3 -> 3 + 3  (* ??? *)
```

at which point it seems clear that something has gone wrong.

You might think that `fun 3 -> 3 + 3` is a syntax error, but that isn't actually the case in OCaml! The `fun` form uses pattern-matching to decide what to do with the arguments, so `fun 3 -> 3 + 3` is a function that can only take as its input the integer `3`, and will always return 6. Clearly, this isn't going to be useful, and is not what was intended.

> **Note**
>
> If you type in `fun 3 -> 3 + 3` into the OCaml interpreter, you'll get a warning about non-exhaustive pattern matches, as you'd expect.

OK, so it seems that naively doing substitution in cases like this (where you have nested `fun` expressions that use the same argument name) doesn't work properly. So how can we fix our model so it can handle situations like this?

The key is to realize that the problems started with this line:

```
substitute 3 for x in (fun x -> x + x)
```

Normally, we are substituting a value for a *free variable*, which is one that isn't being bound by a `fun` expression. However, in the expression `fun x -> x + x`, there are two different `x`s:

- the first `x`, which is in the *binding position* of the `fun` expression
- the second two `x`s, which are both *bound variables*, which means that their value is determined by the (eventual) value passed to the `fun` expression as its `x` argument.

Neither case corresponds to a free variable. And, as it turns out, we can't substitute into either of them.

We've already seen that when we allow substituting into the binding position of a `fun` expression, strange things like `fun 3 -> 3 + 3` result. So it's perfectly reasonable to add an extra rule to the substitution model which says "don't allow substitutions into the binding position of `fun` expressions". So let's do that:

> **Extra rule for the substitution model**
>
> Don't allow substituting into the binding position of a `fun` expression.

If we go back to the substitution that gave us grief:

```
substitute 3 for x in (fun x -> x + x)
```

then we could do the substitution with the extra rule to get:

```
substitute 3 for x in (fun x -> x + x)
--> fun x -> 3 + 3
```

This isn't as obviously broken as the previous attempt, but something is still funny here. It doesn't seem reasonable that we would want to generate a function that, given an arbitrary argument `x`, would always return `6`.

In fact, the expression `fun x -> x + x` is completely self-contained (other than the `+` symbol). It only has one reasonable interpretation: a function which doubles its input. So we should disallow substitution into the `x`s that come after the arrow (`->`) in the `fun` expression too. Our rationale for this is that the `x` in the binding position "shields" the other `x`s (which we refer to as being *bound* by the `x` in the binding position) from substitution. So we have yet another rule for the substitution model:

Second extra rule for the substitution model

Don't allow substituting into a variable which is shielded by the same variable in the binding position of a `fun` expression.

We can combine both rules into a single rule:

Shielding rule for the substitution model

Don't allow substituting into a variable when it is either in the binding position of a `fun` expression or when it is a bound variable of a `fun` expression. We say that the variable in the binding position *shields* the bound variables from substitution.

That's basically it for the substitution model. We will find eventually that this model has limitations that can only be overcome by switching to a new mode (the *environment model*), which we will discuss later in the course.

---

1. Technically speaking, this isn't a true *expression*, since it doesn't return a value. It would be more accurate to call this a top-level `let` *binding*. ↩
2. In a lazy functional language like Haskell, `if` *could* be a function, except for the syntax. ↩
3. This would make an awesome assignment problem, but I'm not quite that mean. ↩