The Substitution Model

In CS111 and CS230, we used the JAVA Execution Model to explain the execution of JAVA programs. In order to understand OCAML features like function values, recursion, pattern-matching, and `let`-binding, it is helpful to have a model to explain how OCAML expressions are evaluated. Here we introduce the OCAML **substitution model** as a way to understand OCAML evaluation. We shall see that the OCAML substitution model is *much* simpler than the JAVA Execution Model, in large part because it is similar to performing algebraic simplification of mathematical expressions.

1   Values

The goal of the substitution model is find the *value* denoted by an expression. Intuitively, a value is an expression that is so simple that it cannot be simplified any further — it stands for itself. Here are some examples of OCAML values:

- the unit value: `()`;

- boolean values: `true`, `false`;

- integers: e.g., `17`, `0`, `-23`;

- floating point numbers: e.g., `3.14159`, `5.0`, `-1.23`;

- characters: e.g., `'a'`, `'B'`, `'3'`, `'\n'`;

- strings: e.g., `"Hi!"'`, `"foo bar baz"`, `""`;

- functions: e.g., `fun x -> x + 1`, `(+)` (*Note:* The fact that functions are values in OCAML is an extremely important feature — one we shall explore in more detail soon.)

- tuples of values: e.g., `(true, 17)`, `(3.14159, 'a', ("Hi!", fun x -> x + 1))`;

- lists of values: e.g., `[]`, `[3;1;2]`, `[[""]; ["a";"b"]; ["aa";"ab";"ba";"bb"]]`, `[('a', true); ('b', false)]`, `[fun x -> x + 1; fun y -> y * 2; fun z -> z * z]`.

2   Simple Operations

OCAML comes equipped with many built-in operations on values that we shall treat as **primitive black-box functions** in the substitution model. We use the notation $E_1 \Rightarrow E_2$ to indicate that the expression E_1 can be simplified to E_2 in the substitution model. For example:

- `1 + 2` $\Rightarrow$ `3`;

- `1 < 2` $\Rightarrow$ `true`;

- `2.718 +. 3.141` $\Rightarrow$ `5.859`

- `max 2.718 3.141` $\Rightarrow$ `3.141`

- `true && false` $\Rightarrow$ `false`;

- `String.get "abc" 1` $\Rightarrow$ `'b'`;

- `fst (4,'a')` $\Rightarrow$ `4`;

- `snd (4,'a')` $\Rightarrow$ `'a'`;

- `List.hd [4;1;2]` $\Rightarrow$ `4`;

- `List.tl [4;1;2]` $\Rightarrow$ `[1;2]`

- `List.length [4;1;2]` $\Rightarrow$ `3`

- `[4;1;2] @ [3;5]` $\Rightarrow$ `[4;1;2;3;5]`

An expression with subexpressions can be evaluated in several steps. For example:

`(1+4) - (2*3)` $\Rightarrow$ `5 - (2*3)` $\Rightarrow$ `5 - 6` $\Rightarrow$ `-1`

As in algebraic simplification, the order in which OCAML subexpressions are evaluated does not matter. So we could evaluate the `(2*3)` *before* the `(1+4)`

`(1+4) - (2*3)` $\Rightarrow$ `(1+4) - 6` $\Rightarrow$ `5 - 6` $\Rightarrow$ `-1`

or we could evaluate the two subexpressions *in parallel*:

`(1+4) - (2*3)` $\Rightarrow$ `5 - 6` $\Rightarrow$ `-1`

All OCAML expressions evaluate to a value, but some also perform a **side effect** — that is, they change the state of the computational world in some way. For example, the expression `print_string "cs251"` has the side effect of displaying the characters `cs251` on the computer console. It also evaluates to the **unit value**, `()`, which is the only value in the `unit` type. This trivial value is typically used for expressions that are evaluated for their side effect, not for their value. The substitution model can show that `print_string "cs251"` evaluates to the unit value, but it does not explain how to keep track of its side effects; that is outside the scope of the substitution model:

`print_string "cs251"` $\Rightarrow$ `()` *(* Also displays ''cs251'' on the console *)*

We will only consider the evaluation of well-typed OCAML expressions, so we never have to worry about evaluating non-sensical expressions like:

- `1 + true`

- `fst [3;1;2]`

- `List.hd (1, true)`

However, even with well-typed expressions, it is possible to encounter expressions that cannot be evaluated because they contain an error. In the substitution model, we will say that such expressions are **stuck**, and will use the notation $E \not\Rightarrow$ to indicate that the expression E is stuck. For example:

- `5/0` $\not\Rightarrow$

- `List.hd []` $\not\Rightarrow$

- `String.get "abc" 3` $\not\Rightarrow$

- `1 + (5/0)` $\not\Rightarrow$

2

The last example indicates that an expression containing a stuck subexpression can also be stuck. Note that a "real" OCAML interpreter will raise an exception in situations like the above.

```
# 5/0;;
Exception: Division_by_zero.
# List.hd [];;
Exception: Failure "hd".

# String.get "abc" 3;;
Exception: Invalid_argument "String.get".
# 1 + (5/0);;
Exception: Division_by_zero.
```

So our substitution model will handle stuck expressions differently than the OCAML interpreter.

3 Conditionals

A conditional expression if E_{test} then E_{then} else E_{else} is evaluated by first evaluating E_{test} to a boolean value, and then using this to determine the branch taken by the following rules:

1. if true then E_{then} else $E_{else} \Rightarrow E_{then}$

2. if false then E_{then} else $E_{else} \Rightarrow E_{else}$

For example:

```
if (1<2) && (3>4) then 5+6 else 7*8
⇒ if true && false then 5+6 else 7*8
⇒ if false then 5+6 else 7*8
⇒ 7*8
⇒ 56
```

Note that the then or else branch of a conditional is not evaluated until the test expression is fully evaluated. This means that it is possible for the branch not taken to contain a stuck expression that does *not* cause the whole conditional to be stuck. For example:

```
if true then 5+6 else 7/0 ⇒ 5+6 ⇒ 11
```

4 Sequential Evaluation

A sequence expression (E_1 ; E_2) is evaluated by first evaluating E_1 to a unit value, and then rewriting the sequence expression using the following rule:

(() ; E_2) $\Rightarrow E_2$

It is an error if E_1 does not have the unit type. For example:

```
1 + (print_string "cs251"; 2*3)
⇒ 1 + ((); 2*3) (* Also displays ``cs251'' on the console *)
⇒ 1 + (2*3)
⇒ 1 + 6
⇒ 7
```

5 Pattern Matching

A pattern matching construct `match` E_{disc} `with` *clauses* is evaluated by (1) evaluating the discriminant expression E_{disc} to a value V_{disc}; (2) using this value to choose the matching clause P_{pat} `->` E_{body} in *clauses*; and (3) evaluating E_{body} after substituting the values in V_{disc} for the corresponding names in the pattern P_{pat}. For example:

- `match (1,2) with (a,b) -> a+b`
  $\Rightarrow$ `1+2`
  $\Rightarrow$ `3`

- `match ((1,2),(3,4)) with ((a,b),(c,d)) -> (a+c,b+d)`
  $\Rightarrow$ `(1+3,2+4)`
  $\Rightarrow$ `(4,6)`

- `match [] with [] -> [17] | [x] -> [x*2] | (x::xs) -> (x+1)::xs`
  $\Rightarrow$ `[17]`

- `match [3] with [] -> [17] | [x] -> [x*2] | (x::xs) -> (x+1)::xs`
  $\Rightarrow$ `[3*2]`
  $\Rightarrow$ `[6]`

- `match [3;1;2] with [] -> [17] | [x] -> [x*2] | (x::xs) -> (x+1)::xs`
  $\Rightarrow$ `(3+1)::[1;2]`
  $\Rightarrow$ `[4;1;2]`

If there is no clause that matches V_{disc}, the `match` expression is stuck. For example:

> `match [] with (x:xs) -> x` $\not\Rightarrow$

A `let` expression desugars into a `match` expression:

> `let (a,b) = (1,2) in a+b`
> $\Rightarrow$ `match (1,2) with (a,b) -> a+b`
> $\Rightarrow$ `1+2`
> $\Rightarrow$ `3`
>
> `let (a,b) = (1,2)`
> `and (c,d) = (3,4)`
> `  in (a+c,b+d)`
> $\Rightarrow$ `match ((1,2),(3,4)) with ((a,b),(c,d)) -> (a+c,b+d)`
> $\Rightarrow$ `(1+3,2+4)`
> $\Rightarrow$ `(4,6)`

However, we will often evaluate a `let` expression "directly" (i.e., without the desugaring step):

> `let (a,b) = (1,2) in a+b`
> $\Rightarrow$ `1+2`
> $\Rightarrow$ `3`
>
> `let (a,b) = (1,2)`
> `and (c,d) = (3,4)`
> `  in (a+c,b+d)`
> $\Rightarrow$ `(1+3,2+4)`
> $\Rightarrow$ `(4,6)`

In cases where the same name appears multiple times, we will often add subscripts to the names to distinguish them. This models the fact that the same name may refer to different logical variables in different parts of the expression. For example:

```
let a = 2+3 in (let a = a*a in 2*a) + a
⇒ let a₁ = 2+3 in (let a₂ = a₁*a₁ in 2*a₂) + a₁
⇒ let a₁ = 5 in (let a₂ = a₁*a₁ in 2*a₂) + a₁
⇒ (let a₂ = 5*5 in 2*a₂) + 5
⇒ (let a₂ = 25 in 2*a₂) + 5
⇒ (2*25) + 5
⇒ 50 + 5
⇒ 55
```

6   Function Application

As we have seen, a `fun` expression is just an OCAML notation for a function value. For example, the `fun` expression

```
fun x -> x*x
```

is pronounced "a function that takes an integer `x` and muliplies it by itself." Such an expression can be used in the operator position of a function call. In the substitution model, an invocation of a function to an argument value rewrites to a copy of the body of the function in which each occurrence of the formal parameter has been replaced by the argument value. For example:

```
(fun x -> x*x) (2+3)
⇒ (fun x -> x*x) 5
⇒ 5*5
⇒ 25
```

OCAML is a **call-by-value** language, which means that all function arguments must be fully evaluated to values before the function is invoked. For example, the following expression is stuck because the function argument is stuck, even though the function does not "use" the argument:

```
(fun y -> 3) (5/0) ⇏
```

Some other languages use alternative evaluation strategies (known as call-by-name and call-by-need) in which the above function application would evaluate to 3 rather than being stuck. We will study these other strategies later in this semester.

Functions with patterns in the formal parameter position desugar to bodies involving `match`:

```
(fun (a,b) -> (a+b)/2) (3,7)
⇒ (fun p -> match p with (a,b) -> (a+b)/2) (3,7)
⇒ match (3,7) with (a,b) -> (a+b)/2
⇒ (3+7)/2
⇒ 10/2
⇒ 5
```

In practice, we will often do the pattern-matching on formal parameter patterns directly:

```
(fun (a,b) -> (a+b)/2) (3,7)
⇒ (3+7)/2
⇒ 10/2
⇒ 5
```

Functions that take multiple parameters desugar into nested functions of single parameters:

```
(fun x y -> (x+y)/(x-y)) 6 4
⇒ (fun x -> (fun y -> (x+y)/(x-y))) 6 4
⇒ (fun y -> (6+y)/(6-y)) 4   (* first substitute 6 for x *)
⇒ (6+4)/(6-4)                (* then substitute 4 for y *)
⇒ 10/2
⇒ 5
```

In practice, we will often substitute all available argument values simultaneously:

```
(fun x y -> (x+y)/(x-y)) 6 4
⇒ (6+4)/(6-4)
⇒ 10/2
⇒ 5
```

7   Global Names

Global names can be handled in the substitution model by replacing any globally defined name by its associated value. It may be necessary to rename variables (e.g., by subscripting) to make all global names distinct. For example:

```
let a₁ = 2+3
⇒ let a₁ = 5

let add_a x = x+a₁

let a₂ = a₁*a₁
⇒ let a₂ = 5*5
⇒ let a₂ = 25

let mul_a y = y * a₂

let dec a₃ = a₃-1

dec (mul_a (add_a a₂))
⇒ dec (mul_a (add_a 25))
⇒ dec (mul_a ((fun x -> x+a₁) 25))
⇒ dec (mul_a (25+5))
⇒ dec (mul_a 30)
⇒ dec (mul_a 30)
⇒ dec ((fun y -> y*a₂) 30)
⇒ dec (30*25)
⇒ dec 750
⇒ (fun a₃ -> a₃-1) 750
⇒ 750-1
⇒ 749
```

8 Recursion

The meaning of recursion on globally defined functions is explained by the substitution model without any new rules. For example:

```
let rec fact n = if n = 0 then 1 else n*(fact(n-1))

fact 5
⇒ if 5 = 0 then 1 else 5*(fact(5-1))
⇒ if false then 1 else 5*(fact(5-1))
⇒ 5*(fact(5-1))
⇒ 5*(fact(4))
⇒ 5*(4*fact(3)) (* skip the evaluation of if and decrement *)
⇒ 5*(4*(3*(fact(2))))
⇒ 5*(4*(3*(2*(fact(1)))))
⇒ 5*(4*(3*(2*(1*(fact(0))))))
⇒ 5*(4*(3*(2*(1*1))))
⇒ 5*(4*(3*(2*1)))
⇒ 5*(4*(3*2))
⇒ 5*(4*6)
⇒ 5*24
⇒ 120
```

Note how pending multiplications in the factorial example are represented in the substitution model.
   Tail recursion is also explained by the substitution model. For example:

```
let rec factTail (num,ans) = if num=0 then ans else factTail(num-1,num*ans);;

let factIter n = factTail(n,1);;

factIter 5
⇒ factTail(5,1)
⇒ if 5=0 then 1 else factTail(5-1,5*1)
⇒ if false then 1 else factTail(5-1,5*1)
⇒ factTail(5-1,5*1)
⇒ factTail(4,5)
⇒ factTail(4-1,4*5) (* skip evaluation of if *)
⇒ factTail(3,20)
⇒ factTail(3-1,3*20)
⇒ factTail(2,60)
⇒ factTail(2-1,2*60)
⇒ factTail(1,120)
⇒ factTail(1-1,1*120)
⇒ factTail(0,120)
⇒ 120
```