Environmental Model

COMP 320

ENVIRONMENTAL MODEL

Substitution Model

So far, evaluation of expressions has been done using the substitution model

This model provides a high level abstract view of how programs are evaluated

It is simple

It is useful in proving properties of programs

The following rules are used in evaluating functions:

```
(fn x => e) v -> [v/x]e

let x = v in e -> [v/x]e
```

Drawbacks - Inefficiency

The value of v is substituted for each occurrence of x in the expression

- This would be inefficient to implement
- It would be nice if we could just remember the binding between x and the value of v in the appropriate environment
- We could just look up this binding when we need it

Drawbacks - Incorrectness

The substitution model is not easy to extend to support references and assignment

```
let
  val x = ref 0
in
  (x := 3); (!x)
end
```

With substitution model, we get result `(!(ref 0))` which gives 0, not 3

Substitutions do not keep track of state/updates to memory cells

Extending the substitution model can be done but we look at a different approach

The Environmental Model

Introduce an environment to keep track of the binding between a variable name and value

A lower level view of operational semantics

Closer to a practical implementation model

Explains references

Definitions

Binding: the association between a name and value

- The name can be the name of a variable, function or memory cell
- Examples:

```
let val x = 10 in x + 3 end
val square = (fn x => x * x)
let val x = ref 2 in !x + 3 end
```

Frame: a collection of zero or more bindings with a pointer to another frame called the enclosing environment.

Environment: A structured collection of frames starting from a particular frame and following the links to the enclosing environments until the global environment is reached

Evaluation

An expression is evaluated in the context of a specific environment

The environment associates a value with a name used in the expression

How?

The first frame in the environment is searched for a binding and the associated value is used

If there is no binding there, the frame of the enclosing environment is searched and so on, up to the global environment

If the name is never found, an error is reported

Example 1

```
val x = 10;

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

First bind the name x to the value 10

Bind the name “fact” to a structure with the definition

- The definition has the body, and the argument
- Link this body to the binding for fact
- Link fact to the binding for x

Example 2

```
val x = 10;

fun foo r = x * r;

val x = 5;
```

First bind x to 10

Then bind foo to an environment containing the body, the parameter r and a pointer to the binding of foo

Then bind x to 5 and link this to the binding of foo

Example 2 Evaluation

```
let
  val x = 10
  fun foo r = x * r
  val x = 5
in
  foo x
end
```

Create a new binding between r and 5 which points to the binding of foo

When we evaluate the body of foo, we find the x bound to 10

Example 3

```
val x = 10

fun square n = n * n

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

We have bindings for x, square and fact

Evaluate `square x`:

- Search environment for square
- Bind the parameter to the value of the argument (new frame)
- Evaluate the body of square in this environment
- Remove the binding that was added

Evaluating Fact

Evaluate `fact (x - 8)`

- Create a new frame linked to fact that binds n to (x-8)
- Execute the body of fact
- The recursive call creates a new binding for n (1) which points to the previous binding
- This process continues until we reach the base case

Note that this model only keeps track of bindings

The computation is performed using a runtime stack which is not modelled here

Modelling References

```
val x = ref 10;
fun foo r = (!x) * r;
x := 5;
foo (!x)
val x = ref 6;
```

References represented by a name and a pointer to a cell

Example 3

```
let x = 1 in
  let y = (let u = 3 in u + x end) in
    let x = 2 in
      x + y
    end
  end
end
```

We bind x with value 1, then y with no value yet.

We bind u, use it to compute a value for y and then remove the binding of u

Then we bind the new binding of x and compute the body

Example 4

```
let x = 1 in
  let f = (let u = 3 in (fn y => u + x + y) end) in
    let x = 2 in
      f(x)
    end
  end
end
```

We can draw the environments created in evaluating this expression