

Handout C: Referencing Environments

Christoph Reichenbach

November 12, 2021

In the definition of the language LETADD in Handout A, we encountered natural (operational) semantics evaluation rules of the form

$$E \vdash p \Downarrow n$$

that evaluate a program p to a value n . Here, E was the (*environment* that helped us interpret the meaning of variables. For example, in the program

```
let x = 1 in x + x
```

we would map the variable x to a value 1 to form the environment $\{x \mapsto 1\}$ and then evaluate the expression $x + x$ with

$$\{x \mapsto 1\} \vdash x + x$$

These environments are closely connected to the concept of *scopes* that we will introduce later. We here highlight what they like in realistic programming languages.

1 Referencing Environments

Consider the following Java code snippet:

```
1 public static boolean isEqual(int a, int b) {  
2     return a == b;  
3 }
```

In this function, line 2 checks whether variables a and b are equal. As we discussed in Handout B, the exact comparison that Java performs can depend on the types of a and b . However, in our environments we have so far only mapped variable *names* to variable *values*, meaning that we would not be able to model this behaviour.

For a general-purpose language, environments thus map variable names not just to values, but also to variable type and address information (and to any other bindings used by the language, e.g., whether the variable is final in Java).

We call these types of environments **referencing environments**. While we can model them as a mapping from variable names to a complete description of all bindings, it is usually easier to think of them as a map from variable names to variable declaration sites (cf. EDAN65), as we will see in the following section.

1.1 Shadowing

In larger programs, we often encounter the same variable name with multiple different meanings. For example, consider the following C code:

```
1 int f(int z) {  
2     int x = 0;  
3     int y = 0;
```

```

4      if (z > 0) {
5          int x = 1;
6          y = x;
7      } else {
8          int z = 2;
9          x = z;
10     }
11     return x + y;
12 }

```

This code defines multiple variables with the names `x` and `z`, so that at different lines in the program, the referencing environment maps `x` and `z` to different variables. The above code is equivalent to the code below, in which all variable names are indexed by the line that contains the variable's declaration; e.g., `x5` means “the variable declared at line 5 that has the name `x`”:

```

1  int f(int z1) {
    // Environment E2 = {x ↦ x2, y ↦ y3, z ↦ z1}
2  int x2 = 10;
3  int y3 = 0;
4  if (z1 > 0) {
    // Environment E5 = {x ↦ x5, y ↦ y3, z ↦ z1}
5  int x5 = 1;
6  y3 = x5;
7  } else {
    // Environment E8 = {x ↦ x2, y ↦ y3, z ↦ z8}
8  int z8 = 2;
9  x2 = z8;
10 }
    // Environment E2 = {x ↦ x2, y ↦ y3, z ↦ z1}
11 return x2 + y3;
12 }

```

The figure also shows the referencing environment at four different points in the program. Within a C function like our function `f`, the environment only changes when we enter or exit a *block* (delimited by curly braces, `{ ... }`). Thus, we have three referencing environments: E_2 , E_5 , and E_8 , one per block. In each environment we represent the bindings by the variable declaration site, e.g., `x5`.

We use the term “binding” not only to refer to the assignment of properties of a variable (storage location, type etc.), but also to refer to the mapping of a name to a variable in an environment, like `x ↦ x5`. This justifies Sebesta’s use of the term in Section 5.3.1 in the textbook. Note that some variables have no names, and other variables may have multiple names, depending on the semantics of the programming language.

As we see, the binding of e.g. `x` to `x5` is temporary and only extends from line 5 to line 6; in those lines, this binding *shadows* the binding `x ↦ x2`. Thus, the assignment in line 6 will assign the value 1 to `y`.

1.2 Implicit Environment Bindings

Most languages pre-initialise the referencing environment with some existing definitions. For example, the `awk` language pre-defines the variable `ARGV` as an array that holds all parameters passed to the current program.

As we will see later, referencing environments also usually contain bindings for functions and other language constructs (types, classes etc.). For example, the initial environment in Python includes several predefined functions, such as `len`, which computes the length of a list.