Basic Concepts about Programming Languages

Luca Abeni

November 18, 2022

1 Identifiers, Bindings, and Environment

Almost all of the programming languages allow to associate symbolic names to the various "entities" composing the programs (values, memory locations, variabiles, functions, etc...).

More formally, a programming language is composed of *denotable entities* that can be referred through symbolic names (identifiers). This mechanism is implemented by a function named *environment*, having the set of possible identifiers as a domain and the set of denotable entities as a codomain. This function can change during the program runtime, as it is possible to create new bindings between identifier and denotable entities, or to destroy existing bindings. Moreover, some bindings between identifiers and denotable entities only exist in some parts of the code[1]. Hence, the following definitions hold:

Definition 1 (Binding) *A binding between an identifier I and a denotable entity E is a pair (I, E) associating name and entity.*

Definition 2 (Environment) *The environment is the set $\{(I, E)\}$ of bindings existing in a specific moment of the program execution, while the code of a specific program block is being executed.*

this definition is not surprising, since the environment is a function $env : \mathcal{I} \rightarrow \mathcal{E}$ (where $\mathcal{I}$ is the set of possible identifiers, and $\mathcal{E}$ is the set of denotable entities) and from the mathematical point of view a function is a set of (I, E) pairs, so $env \subset \mathcal{I} \times \mathcal{E}$.

The set of denotable entities is clearly language-dependent; for example:

- in the Assembly language, identifiers can be associated only to memory locations (so, the only possible denotable entities are memory addresses)
- in higher-level imperative languages, identifiers can be associated to values, variables, functions, or data types
- in λ-calculus, identifiers can only be associated to functions (so, functions are the only possible denotable entities)
- in higher-level functional programming languages, identifiers can be associated to values (mutable variables do not exist, and functions are values!)

In programming languages having the "code block" concept, it is possible to make a distinction between *local environment* (the environment valid inside a code block), *non-local environment*, and *global environment*.

Definition 3 (Local Environent) *The local environment of a code block is the subset of the environment composed of the bindings that have been created inside the code block (and are hence valid only inside this block)*

Definition 4 (Non-Local Environment) *The non-local environment of a code block is the subset of the environment which is not part of the local environment (and hence contains all the bindings that have been created* outside *of this block and will remain valid when the execution exits the block)*

Definition 5 (Global Environment) *The global environment is the subset of the environment containing bindings that are not created inside any code block*

[1] As an example, the environment is modified by a declaration, or by the first usage of a denotable entity.

2 Mutable Variables

According to the imperative approach, the execution of programs happens by modifying the values contained in some memory locations, which are called *variables* in high-level languages.

Definition 6 (Mutable Variable) *A mutable variable is a denotable entity representing some memory locations that can contain* storable entities

The definition above is based on "*storable entities*", which, again, are a language-dependent concept... But in general a storable entity is some value that can be stored in a variable.

From the conceptual point of view, mutable variables imply the existance of a second function, after the environment, named "*store*", associating each variable with the storable entity contained into it.

Definition 7 (Store) *The store is a set of pairs* (V, E) *(where* V *is a variable and* E *is a storable entity) associating each variable with the value contained into it*

The store is hence a function (representing the memory used by the program to store its data, or the mutable state of the program) which has the set of program's variables as a domain, and the set of storable entities used by the program as a codomain.

When, using an imperative programming language, the value stored in variable "`x`" is used, the (abstract) machine applies the store function to the value returned by the environment function applied to identifier "`x`": "`x`": $store(env(\texttt{x}))$ (where "env" is the environment).

3 Denotable, Storable, and Expressible Entities

As discussed, a program is composed of some "entities" (whose definition depends on the programming languages, but can be data types, values, variables, functions, etc...). Such entities can be *denotable*, *storable*, and *expressible*.

(Note: in literature the definitions of "storable", "expressible", and "denotable" are often associated to values)

Definition 8 (Denotable Entities) *A denotable entity is a language entity that can be associated to a symbolic name / identifier*

Definition 9 (Storable Entities) *A storable entity is a language entity that can be stored in a variable*

Definition 10 (Expressible Entities) *An expressible entity is a program entity that can be generated computing an expression*

A denotable entity is hence a generic entity that can be referred through a name (defined by the user or pre-defined in the language); the set of denotable entities is the codomain of the environment function. An expressible entity is a generic entity that "can be computed/allocated" somehow using the language's constructs. Finally, the storable entities (which only exist in imperative programming languages) form the codomain of the store function.

In a functional programming language, all entities are denotable and expressible, whereas in an imperative programming language there can be entities that are denotable but not expressible or storable (for example, functions in the C programming language).

4 Functions

Almost all the high-level programming languages allow some form of code modularization by decomposing programs into a set of components (subprograms/subroutines/functions/procedures). Each one of these components implements a specific functionality according to a well-specified interface.

Each subroutine is hence an *entity* implementing a self-contained part of the code which can be invoked exchanging some values (parameters and return values) with the caller. A subroutine which can return a value is generally called "function" (clearly, this is a very different thing respect to a mathematical function!!!).

Definition 11 (Function) *A function is a denotable entity composed of a block of code associated to a name that can be used to invoke its execution. When the execution of a function is invoked, the caller can exchange some data with it through its parameters, its return value and some global state of the program.*

```
void wrong_swap(int a, int b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}

void correct_swap(int *a, int *b)
{
  int tmp;

  tmp = *a; *a = *b; *b = tmp;
}
```

Figure 1: Example of C function trying to exchange the values of two variables. Since the C language mandates parameters passing by value, the "`wrong_swap()`" will have no effect (as its name suggests); the "`correct_swap()`" function, instead, receives as parameters some *pointers* to the variables, and can hence work correctly.

```
void reference_swap(int &a, int &b)
{
  int tmp;

  tmp = a; a = b; b = tmp;
}
```

Figure 2: Example of function swapping the contents of two variables, implemented in C++ passing parameters by reference.

Definition 12 (Foramal Parameter) *A formal parameter of a function (specified in the function's definition) identifies a variable that can be used by the caller to pass data to the function when invoking it*

Definition 13 (Actual Parameter) *An actual parameter is an expression (specified when invoking a function) that will be associated to the corresponding formal parameter during the function's execution*

The fact that a function is composed of a block of code makes it clear that the function is characterised by a local environment (containing bindings between names and local variables, names and function parameters, etc...). Moreover, since a function is defined as a denotable entity this block of code (the function body) can be associated to a name (but this is not always necessary: anonymous functions do exist!).

The way actual parameters are associated to formal parameters depends on the mechanism used to invoke the function, and to the parameters passing style that is used. For example, a formal parameter identifies a variable which can be created when the function is invoked (and is hence a local variable of the function), or can exist before the function is invoked. Different parameters passing styles can be used, and the most important are: parameters passing by value, by reference, and by name.

When parameters are passed *by name*, a new local variable for each formal parameter is allocated when the function is invoked (and is deallocated when the function returns). The simplest way to manage these variables is allocating them on the stack. The local environment of the function then binds the formal parameter name to this variable, and the variable is initialized with the value of the actual parameter (hence the name "by value"). If the function modifies the value stored in this variable, the modifications are discarded when the function returns and the variable is deallocated. In other words, passing parameters by value it is possible to pass data from the caller to the called, but not vice-versa. This mechanism is the only one supported by the C programming language; as a result, the "`wrong_swap()`" function in Figure 1 does not work correctly (and pointers are needed to code a working "swap" function).

When parameters are passe *by reference*, instead, no new variable is allocated when the function is

```
int f(int v)
{
  int a = 666;

  return a + v;
}
```

Figure 3: Example of function passing parameters by name.

invoked. Parameters are passed by modifying the local environment of the function so that the formal parameter's name is bound to the actual parameter. Hence, the actual parameter has to be a denotable entity (for example, things like "`x + 1`" are not valid actual parameters when passing parameters by reference). Using the C/C++ jargon, this means that actual parameters have to be L-values. This parameters passing style is not supported by the C language, but is supported by C++; as an example, see the "`reference_swap()`" function in Figure 2 (comparing this code with Figure 1 it is possible to better understand the differences between passing parameters by name and by reference).

Finally, when parameters are passed *by name* invoking a function requires to replace each formal parameter with the corresponding actual parameter (this is just text replacement). This mechanism is typically used to evaluate (reduce) functional programs. Although parameters passing by name might look simple to simplement, there are some subtle issues to be addressed. For example, consider the "`f()`" function from Figure 3: the goal of the function is to sum 666 to the value received as input, and if parameters are passed by name the "`f(n)`" is evaluated as "{ **int** a = 666; **return** a + n;}", which is reduced to "{ **return** 666 + n;}", and the return value is correctly "`666 + n`". But if "`f(a)`" is invoked, things become more complex: a simple replacement of "`v`" with "`a`" would result in "{ **int** a = 666; **return** a + a;}" which reduces to "`666 + 666`", clearly not the expected result... The issue is that when replacing "`v`" with "`a`" it is not possible to make a distinction between two different entities (a local variable and a non local one) which have the same name "`a`".

This issue is generally addressed in functional programming by properly changing the variables' names: if "`f()`" is invoked using as an actual parameter an expression containing "`a`", then the local variable "`int a`" must be renamed (for example to "`a1`") so that the replacement can result in "{ **int** a1 = 666; **return** a1 + a;}" which correctly evaluates to "`666 + a`".

From an implementation point of view, parameter passing by name has been historically implemented by passing "*thunk*s", which are (environment,expression) pairs, as parameters. Hence, a function receiving parameters passed by name can be implemented as a function receiving (environment,expression) pairs as parameters[2].

5 Closures

```
void->int counter(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}
```

Figure 4: Example of a function returning a closure.

In some programming languages, functions are storable entities (there are variables that can store functions) or expressible entities (it is possible to build expressions evaluating to a function). As a result,

[2]the expressions are the actual parameters, and the environments are used to evaluate such expressions; in the example of Figure 3, the expression is "`a`" and the environment binds this "`a`" name to the global "`a`" variable).

they can be used as actual parameters for other functions, or as return values for functions. In these cases, the values to be stored, returned, or passed as parameters are technically "*closures*"

Definition 14 (Closure) *A closure is a pair composed of a function and its non-local environment*

Basically, a closure is needed to find the entities bound to identifiers for which there are no bindings in the local environment of the function.

```
#include <stdio.h>

int (*counter(void))(void)
{
  int n = 0;

  int f(void) {
    return n++;
  }

  return f;
}

int main()
{
  int (*c1)(void) = counter();
  int (*c2)(void) = counter();
  int (*c3)(void) = counter();

  printf("   %d %d\n", c1(), c1());
  printf("%d %d %d\n", c2(), c2(), c2());
  printf("   %d %d\n", c3(), c3());

  return 0;
}
```

Figure 5: Example showing the function pointers in C are not closures (the example also uses a non-standard extension provided by the `gcc` compiler).

As an example, look at the function "`void->int contatore(void)`" from Figure 4, is coded using a pseudo-language with a C-like syntax in which "`void->int`" is the type of the functions withot arguments that return a value of type `int`. The "`counter()`" function receives no arguments and returns a function that generates all the integer numbers starting from 0. It is possible to notice that "`n`;; is a local variable of the "`counter()`" function, and not a variable or an argument of the "`f()`" function. Hence, when "`counter()`" is invoked its local variable contains a binding from "`n`" to a local variable initialized to 0. Such a binding is not in the local environment of "`f()`" (it is in its non-local environment) and when "`counter()`" returns the variable bound to "`n`" is deallocated. So, in order for the program to work 2 things are needed:

1. The variable bound to identifier "`n`" should not be deallocated when "`counter()`" returns. Hence, it cannot be allocated on the stack, but must be allocated on the heap
2. The binding between identifier "`n`" and such a variable has to be somehow associated to the returned function. It is hence copied in a new environment that will be part of a closure returned by "`counter()''`

To better understand the differences between closures and function pointers, look at Figure 5, that uses a `gcc` extension to implement the function of Figure 4 in C. In this case, the "`int n`" variable is deallocated when function "`counter()`" returns, hence the program has an undefined behaviour.

6 Closures and Classes

```
#include <functional>
#include <iostream>

auto counter(void)
{
  int n = 0;

  return [n](void) mutable {
    return n++;
  };
}

int main()
{
  auto c1 = counter();
  auto c2 = counter();
  auto c3 = counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure 6: Example showing how to use C++ lambda functions, which implement closures.

```
#include <iostream>

class Counter {
  private:
    int n;
  public:
    Counter(void) : n(0) {
    }
    int operator() (void) {
      return n++;
    }
};

int main()
{
  auto c1 = Counter();
  auto c2 = Counter();
  auto c3 = Counter();

  std::cout << c1() << " " << c1() << std::endl;
  std::cout << c2() << " " << c2() << " " << c2() << std::endl;
  std::cout << c3() << " " << c3() << std::endl;

  return 0;
}
```

Figure 7: Implementing the "`counter()`" function in C++ using classes.

Closures allow associating a non-local environment to a function, and have been originally introduced to implement high-order functions (functions returning functions as a result, or accepting functions as parameters). But the resulting abstraction is much more powerful than this, and allow associating data (the state contained in the variables bound by the non-local environment) to code (the code implementing the function's body). And this is very similar to what classes do.

To better understand the relationship between closures and classes, look at the C++ implementation of a counter as a closure or as a class, as shown in Figures 6 and 7. As it is possible to notice, the "`int n`" variable in the first case is a local variable of the "`counter()`" function (then embedded in the closure) and in the second case is encapsulated in a class as a private member.