Bindings and scope

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 Notable references

- Robert W. Sebesta, “Concepts of Programming Languages” (10th edition)
  - Chapter 5, Names, Bindings, and Scopes

1.2 **TODO** Table of contents

2 Introduction

This section introduces several concepts relating to names and bindings of those names.

Of particular note is *scope*, the “visibility” of names throughout a program.

2.1 What’s a binding?

A *binding* is a general term that refers to any association between an entity and an attribute.

- This is a very vague definition, which allows us to apply it to a great many instances.

The *entities* are “syntactic units” of the language.

- That is, some piece of the language syntax.

The *attributes* may happen to also an entity of the language, or may be a more “external” concept.

For instance,

- a variable (entity) will usually have a name (attribute)
  - and often a type (attribute),
- a type (entity) should have also have a name (attribute.)

In these notes, we primarily discuss bindings of *types*, *lifetimes* and *scopes*. Other bindings may come up later in the course.

2.2 Uses of names

Many entities may be have names bound to them;

- variables, constants, parameters/arguments,
- functions, methods, procedures, subroutines,
- modules, packages, contexts, typeclasses, records, and
- types,

to name a few.

2.3 Persistance of names

Take note!

- Some namings are “global”; they persist for “all time and space”.
  - Such as the namings of builtin language methods,
    * (assuming the language does not allow their names to be reused or shadowed.)
- But many namings are temporary!
  - And one instance of a name may shadow another.
    * (Such as when you declare a local variable/argument with the same name as one already in scope.)

2.4 A note about time and space

Throughout these notes, we will discuss *when* and *where* bindings are in effect.

These discussions of "time" and "space" are relative to the program.

- Our notion of "*time*" is in terms of the program's runtime.
  - Or occasionally, its design time.
- Our notion of "*space*" is in terms of the program's *text*.

2.5 Static and dynamic bindings (or "checking of bindings")

Bindings which are decided (or known) *before* runtime are called *static*.

Conversely, bindings which are not decided (or known) until *during* runtime are called *dynamic*.

We say "or known" because some bindings, such as the types of values, are not so much "decided" as "discovered".

- All values have a type, whether or not that type is checked statically or dynamically.

3 Type checking and polymorphism

Type checking is the process of determining the binding between a name of an entity and a type.

- Usually, only variables, arguments, functions and similar entities have types.
  - There is at least one exception to this; sometimes, types may have a type!
- We will discuss types in more detail in the next section.

3.1 "Dynamically checked" instead of "dynamically typed"

As with many bindings, languages can be categorised as being either *statically* or *dynamically* typed.

In "Types and Programming Languages", though, Pierce argues that the term "dynamically typed language" is a misnomer.

- It would be better to say "dynamically (type) checked".
- All values have a type, even before type checking.

3.2 Static and dynamic type checking

It's somewhat natural for interpreted languages to be dynamically typed, and compiled languages to be statically typed.

- The process of type checking is an upfront cost.
  - Compilation is a natural time to carry out that costly process.
- Consider the (primarily) interpreted "scripting" languages; Python, Ruby, Javascript, Lua, Perl, PHP, etc.
- But it's far from universally true that interpreters use dynamic typing!
  - Haskell has an interpreter, and is definitely statically typed.

3.3 Polymorphism

Assigning each value in a program a single type, and then enforcing type correctness (preventint type clashes) introduces a (solveable) problem; it prohibits code reuse!

- Subroutines can only be used on a particular type of arguments.

Subtyping and polymorphism provide solutions to this.

- With *subtyping*, there is a sub/super relation between types.
  - One classic examples involves a type of "shapes", which has a subtype "polygon", which in turn has subtypes "triangle" and "rectangle", etc.
  - Subtypes closely resemble the notion of *subsets*.
  - Values of a subtype can be used anywhere a value of their supertype is expected.
  - Sub-*classing* is one instance of subtyping.
    - And is usually just called subtyping.
  - Subranges and, sometimes, enumeration types are other examples.
- With *polymorphism*, a subroutine can have several types.

3.4 Ad hoc polymorphism (overloading)

One notion of polymorphism is *ad hoc* polymorphism, also called *overloading*.

- With overloading, subroutine names can be reused as long as the types of arguments differ (in some specified way.)
- The downside is the programmer must define the "same" subroutine many times; once for each type.

Here is an example of ad hoc polymorphism in Scala.

```
def sum(a: Int, b: Int): Int = a + b
def sum(a: Boolean, b: Boolean): Boolean = a || b
```

3.5 Parametric polymorphism

With parametric polymorphism, subroutines have a *most general* type, based on the *shape* or *form* of their arguments.

- This involves the use of *type variables*.
- The subroutines behaviour can only be based on the form, not the specific types.
- This is commonly used in functional languages.
  - Several object-oriented languages have *generics*, which are essentially the same concept.

Here is an example of parametric polymorphism in Scala.

```
def swap[A](p : Tuple2[A,A]): Tuple2[A,A] = p match {
  case (p1, p2) => (p2, p1)
}
```

3.6 Duck typing

More formally called *row polymorphism*.

- Duck typing does not actually check the types of values.
- It is used when calling a method on a value; the only check is that that method is defined for that value.

- “If it walks like a duck, and quacks like a duck, it’s a duck!”

Here is an example of duck typing in Ruby.

```
class Thing1
  def initialize(v,w); @v = v; @w = w end
  def do_the_thing() "I'll implement this later." end
end

class Thing2
  def initialize(v,w); @v = v; @w = w end
  def do_the_thing() @v + @w end
end

def doit(x) x.do_the_thing end

x = Thing1.new(1,2)
y = Thing2.new(1,2)

puts doit(x) # Outputs a string
puts doit(y) # Outputs an integer
```

4 Lifetime

The *lifetime* of an entity is the portion of the runtime (not the code) during which that entity has allocated memory.

To discuss lifetimes, we first need to discuss the layout of memory and the process of memory allocation.

4.1 The data segment, stack, and heap

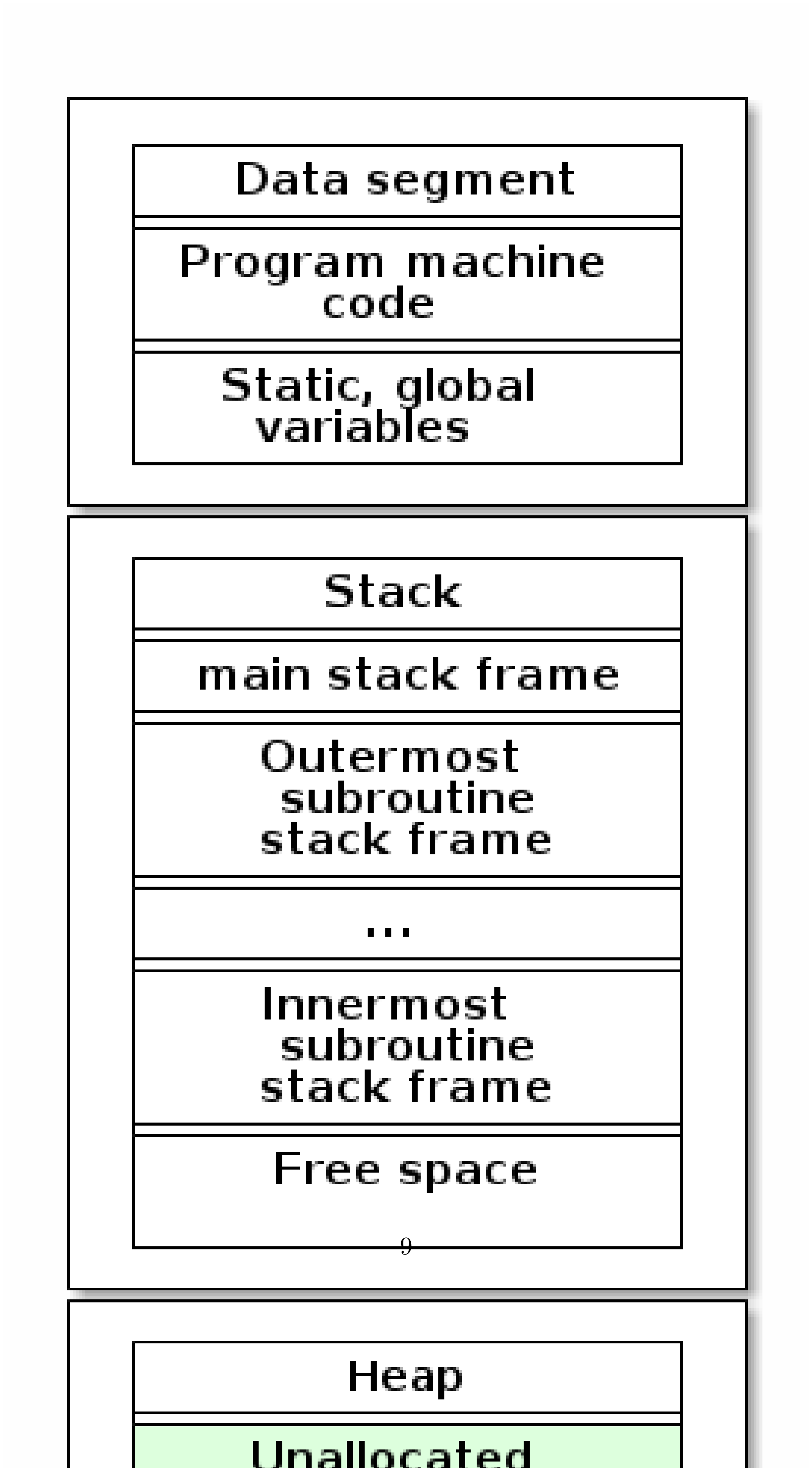
In the context of running a program (or a thread of a program):

- The *data segment* is a portion of memory used to store the program itself and static/global variables.
- The *stack* is an organised portion of memory allocated for the program.
  - Blocks at the “top” of the stack are reserved when entering a scope.
    - These blocks are called *stack frames*.

  - Stack frames are *popped off* the stack when leaving the unit of code.

- The *heap* is an unorganised portion of memory allocated for the program.
  - Allocation/deallocation may be independent of units of code.
  - Disorganisation can lead to inefficiency and other problems.
    - ∗ Cache misses, page faults, etc.
    - ∗ Running out of memory or appropriately sized continguous blocks.
    - ∗ Garbage, dangling references, etc.

4.2 Picturing memory



4.3 Kinds of memory allocation

We can distinguish four kinds of memory allocation;

- static,
- stack dynamic,
- implicit heap dynamic, and
- explicit heap dynamic.

4.4 Static and stack dynamic allocation

- Static
  - Allocation is done *before runtime* (static), in the data segment.
    - At load time, when the program is loaded into memory. (Not compile time.)
- Stack dynamic
  - Allocation is dynamic, automatic, and on the stack.
  - Amount of memory needed must be known when entering their scope.
  - This is "the usual method" in most imperative languages.

4.5 Implicit and explicit heap dynamic allocation

- Implicit heap dynamic
  - Allocation is dynamic, automatic, and on the heap.
  - Memory is simply allocated when the variable is assigned; the programmer does not need to "do anything".
- Explicit heap dynamic
  - Allocation is dynamic, *manual*, and on the heap.
  - The programmer must use a command to allocate memory *before* assignments can be made.
    - E.g., `malloc` or `new`.
  - In fact these blocks of memory can be considered *nameless variables*; we need other, *reference* variables to *point* to their address.

4.6 Exercise: Memory allocation methods

Investigate:

- Which of the five kinds of memory allocation are available in Scala and in Ruby.
- How each kind of memory allocation is denoted.

4.7 The lifetime

To repeat, the *lifetime* of an entity is the portion of the runtime (not the code) during which that entity has allocated memory.

- There may be several *instances* of a single variable in the code, each with its own lifetime.
- The lifetime of a variable depends upon the memory allocation strategy used for it.
  - **Static**: lifetime is the whole of runtime.
  - **Stack dynamic**: lifetime is the portion of runtime between when the scope containing the variable is entered and when control returns to the invoker of that scope.
    - In the case of functions/procedures, local variables end their lifetime when the function/procedure returns.
    - Note that a variable may go out of scope, but still be alive.
  - **Implicit heap-dynamic**: lifetime begins when assigned. The end of lifetime depends upon garbage collection.
  - **Explicit heap-dynamic**: lifetime begins when the allocation command is used. The end of lifetime may be
    - when the programmer issues a deallocation command, or
    - depend upon garbage collection.

4.8 Garbage collection

Garbage is allocated memory which is no longer accessible.

Garbage collection is the process of deallocating garbage.

- A complex activity.

  - We will discuss the basics here, and then likely not touch on it again.
  - Has a large influence on the efficiency of compiled/interpreted code.

- There are two main categories of garbage collection algorithms;
  - tracing,
    - which includes the "mark & sweep" algorithm, and
  - reference counting.

- Additionally, language implementations may include compile time "escape analysis".
  - Memory may be allocated off the stack instead of the heap *if* no references to the variable are available outside its scope (if it doesn't "escape".)

4.8.1 Tracing garbage collection (mark & sweep)

- Usually triggered when a certain memory threshold is reached.
- Starting from some set of base references (usually those in the stack and data segment), "trace" references, somehow marking those that are reachable.
  - Starting from the set of base references, "determine the transitive closure of reachability".
- Once done, free any memory that is not marked.
- "Naive" mark & sweep:
  - Each object in memory has a bitflag for this "marking", kept clear except during garbage collection.
  - When activated, traverse all reachable memory, "marking" what's reached, then "sweep" away all unmarked memory (and reset the flags of marked memory.)
    - With this naive approach, we have to pause the program to do this;
    - better methods do not require pausing.

4.8.2 Reference counting garbage collection

- An "eager" approach.
- As the program runs, keep a count of the number of references to every object in memory.
  - So we must perform some tallying on every assignment!
- When the number of references reaches zero, we can free the memory for that object.
- This approach needs to account for cycles, though!
  - Every object in a cyclic (portion of a) data structure will always have at least one pointer to it, even if the structure as a whole is unreachable.
  - This necessity increases the overhead.

4.9 Dangling/wild references

Garbage is not the only problem that can occur with references to memory.

A *dangling* or *wild* reference is a reference to memory which has already been freed.

- Presumably, by a programmer's action.
- After being freed, memory may be left as is until it is allocated again, or may be wiped in some way.
  - Accessing memory that is no longer reserved usually leads to undefined behaviour.

We will discuss methods of preventing use of wild/dangling references when we discuss reference types later in the course.

5 Scope

A *scope* (singular) is a portion of the program to which the visibility of entities may be limited.

- This is a design decision: what constructs introduce scopes?
  - Almost certainly subroutines do.

  - Do conditional branches? Loop bodies?

- As a general term, within this section we say *block* for a construct which introduces a scope.
  - Outside of this section, we have used and will continue to use the singular *scope* for this, but here that will conflict with the adjective scope we are about to define.

5.1 Static scoping

The *scope* (adjective) of an entity is the portion of the program in which it is "visible".

- Usually scope is statically determined.
  - It is usually the block in which it is defined, and all subblocks of that block,
    - unless it is shadowed by another entity of the same name.
- Static scoping is so pervasive, you have likely never come across a language where scope works differently, outside of differences in what constructs introduce scope.

5.2 Dynamic scoping

With dynamic scoping, the scope of entities depends upon the "execution trace", i.e., the "path" through the program.

- Dynamic scoping is rarely used.
  - The detriments strongly outweigh the benefits in most cases.
  - The one major benefit is relief from having to pass arguments;

5.3 Exercise: On dynamic scoping

- Historically, Lisps (languages directly descended from Lisp) were dynamically scoped.
- Presently, the only widely used Lisp which uses dynamic scoping by default is Emacs Lisp (eLisp.)

Try out this snippet of Emacs lisp code, either in Emacs (if you have it) or using an online REPL.

```
(let ((x 2) (y 5)) ; "global" variables x and y
  (progn
    (defun mult-x-y ()
      (* x y)) ; returns x * y

    (defun A ()
      (let ((x 3)) ; local variable x
        (mult-x-y)))

    (defun B ()
      (let ((y 10)) ; local variable y
        (A)))

    (message "mult-x-y returns %d, A returns %d and B returns
     ↪  %d"
      (mult-x-y) (A) (B))))
```

1. Ensure you understand the results.
2. Using this understanding, formulate some advantages and disadvantages of dynamic scoping.

5.4 Exercise: Entities which introduce scope

In a few different programming languages, investigate whether condition branches and loop bodies introduce scopes.

I.e., test out code such as

```
if B then
  int x = 0
endif

y = x   % Is x still in scope?
```

Specifically, investigate languages which have *iterating loops*, (usually called `for` loops.) What is the scope of an iterator for such loops?