Concepts of programming languages

Lecture 2

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Last time: programming languages

In the first lecture I tried to motivate *why* we might consider programming *languages* themselves as interesting objects of study.

A programming language's definition consists of three parts:

- syntax
- static semantics
- dynamic semantics

Different languages make very different choices in all three of these aspects.

Today

Today's lecture is about introducing **terminology**

- What are the differences between values and expressions?
- What is a type system? How can we classify different type systems?

You will encounter many of these concepts in the languages you study for your project and the languages we will encounter in these lectures.

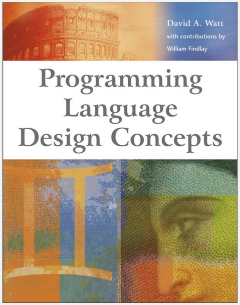

Figure 1: *Programming language design concepts* by David A. Watt

This book gives a fairly comprehensive overview of concepts and terminology that you might encounter during your projects.

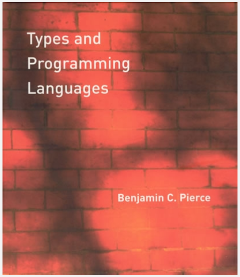

Figure 2: *Types and programming languages* by Benjamin Pierce

This gives a much more precise introduction to the study of programming languages and type systems. The later lectures will be based on this book.

Historical perspective - I

The very first computers were programmed using machine instructions directly.

In the 1950's, hardware became more reliable and people started to recognize that *software development* was a non-trivial problem.

The first programming languages, such as Fortran, were a thin layer of abstraction over instructions for specific machines.

Later languages, such as Algol and C, introduced more *structured programming*.

Historical perspective - II

In the 1990's the concept of *object-oriented programming* took off (Java, C++).

Since then, we've seen the emergence of the web (Javascript) and mobile devices (Swift, Java) as important development platforms.

Functional languages are gaining prominence (Haskell, OCaml, ML, Racket, Erlang) and existing languages are adopting more functional features (C#, Swift).

Historical perspective - III

It is still unclear how languages will continue to evolve:

- Webassembly may offer a more viable compilation target than Javascript.
- Type systems are becoming increasingly advanced. *Dependent types* are gaining traction (Agda, Idris, Coq).
- Large companies control a great deal of the language ecosystems (Microsoft/.NET, Google & Sun/Android, Apple/iOS & OS X). How will these evolve?
- Many applications no longer run on a single desktop machine, but ‘in the cloud’ – how can we program such applications effectively?

We live in interesting times.

Terms, evaluation and values

We will refer to a piece of abstract syntax as a *term* or *expression*.

Every term may be *evaluated*:

```
> if true then 0 else 1
1
```

Evaluating a term produces a *value* – a special kind of term that cannot be reduced any further.

The specification of how evaluation proceeds is given by the language's (dynamic) *semantics*.

Dynamic Semantics – operational semantics

Operational semantics specifies a program's behaviour by defining a transition function between terms.

$$(1 + 2) + 3 \quad \to \quad 3 + 3 \quad \to \quad 6$$

Terms that do not have any transition associated with them are called *normal forms*.

We'll see examples of such semantics later in the course.

Dynamic Semantics – denotational semantics

Denotational semantics specifies a program's behaviour by defining an interpreter as a total function operating on certain types.

$$[\![\text{Add } e_1 \ e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

Giving a denotational semantics for programming languages whose terms may not terminate is not at all trivial. This problem has sparked an area of research known as *domain theory*.

Types

Besides terms, many programming languages have some notion of *type*.

We write $t : \tau$ when the term t has type τ.

Primitive types are those types built into the language definition that cannot be decomposed further. *Primitive values* are values with a primitive type. Examples:

- `'a' : Char` in Haskell;
- `3.14 : float` in C;

Composite types

Besides the primitive types, there are many ways to assemble *composite types* from existing types:

- cartesian products or pairing `(3,'a') : (Int,Char)`
- function space `incr : Int -> Int`
- arrays `int a[12]`
- disjoint union (`Either` in Haskell, enums/unions in C dialects)
- records, dictionaries, sets, …

Some languages allow you to define *recursive types*, such as lists or trees – as you've seen in the course on Functional Programming.

Static vs dynamic typing

If the programming language has a *static semantics* that checks all programs are well-typed at compile type, we say the programming language is *statically typed*.

In a dynamically typed language, these checks are not performed statically, but as a program is run – the type checking is part of the language's *dynamic semantics*.

A language can be both statically and dynamically typed; most languages fall into one of these two categories.

Type inference vs type checking

We can make further distinctions in statically typed languages.

When a programmer provides type signatures for all program terms, the compiler need only perform *type checking*.

When a programmer may leave out type signatures, the compiler needs to perform *type inference* – that is, it needs to *infer* a suitable type for parts of the program.

Most languages – even those that support type inference – still encourage you to write type signatures for (top-level) definitions.

This distinction is sometimes referred to as *manifest* versus *inferred* typing.

Dynamic typing

Different dynamically typed languages take a very different approach to types.

An expression like `1 + "a"` has a different meaning, depending on the language:

- JavaScript coerces the integer to a string and appends them;
- Python will fail dynamically with a type error.

Dynamic languages treat type information very differently.

Static typing and safety

Some people claim that static type systems are necessarily safer than dynamically typed languages.

Yet C is a statically typed language that allows all kinds of unsafe memory access, implicit coercions between data and memory addresses, etc.

It is impossible to say anything sensible about the guarantees that *every* static/dynamic type system provides.

Type soundness

One particularly important theoretical property of type systems is *type soundness*

- **Progress:** every well-typed term is either a value or can take a next evaluation step.
- **Preservation** if a well-typed term takes an evaluation step, the resulting term is well-typed

Together these two properties of a static & dynamic semantics is called **type soundness**.

Question: Can you name any languages that have this property?

Untyped languages

Some people refer to languages without any type system as *untyped*.

Examples may include most assembly dialects or Tcl.

In practice, this distinction is not very important: an ‘untyped’ language just has a trivial static semantics.

Bob Harper (CMU) sometimes refers to such languages as ‘unityped’.

Static vs dynamic typing

There is a great deal of discussion about which is better: static or dynamic typing.

In a statically typed language, a program is only run once it has been type checked. Expressions such as `1 + "a"` are (usually) rejected during compilation.

Some programmers perceive this as being ‘harder’.

Some programs are very hard to assign a static type:

```
var x;
if condition {
  x = "Hello";
} else {
  x = 4;
}
```

Static vs dynamic typing

Some language features – such as metaprogramming – are hard to type statically.

```
var x = 10;
var y = 20;
var a = eval("x * y")
```

Javascripts `eval` function takes a string and evaluates the corresponding program. What is its type?

There are many people who believe strongly in the merits of dynamic typing.

Static vs dynamic typing

Disclaimer: I am not one of these people.

- Programs written in statically typed languages can be *much* easier to refactor and maintain.
- If a program does not type check, more often than not, I am doing something wrong.
- With expressive enough type systems – like those offered by dependently typed languages – we can assign static types to functions like `eval`.
- Compiling static typed languages can generate more efficient code.
- Programming is hard; I need all the help from the machine I can get.

Static vs dynamic typing

A common fallacy is that dynamic languages offer 'more freedom' – that there are sensible programs which static type systems forbid.

The opposite is true!

Consider the following JavaScript example:

```
var x;
if condition {
  x = "Hello";}
else {
  x = 4; }
```

This code will not type check in a statically typed language, such as Haskell.

Static vs dynamic typing

But we can embed the dynamically checked version into Haskell:

```
data Value = IsString String | IsNumeric Integer
  | IsNull | IsObject (Map String Value) | ..

main :: Value
main = if condition then IsString "Hello"
                    else IsValue 5
```

The only difference is that JavaScript provides built-in support for tagging values with their type, taking the union of all possible static types, and converting implicitly between them.

Static vs dynamic typing

Now try to Haskell’s enforce static type safety in JavaScript...

The static types give us *more information* – that can be exploited to write parts of the program and guide program development.

Conor McBride has a great talk on this topic: Is a type a lifebuoy or a lamp?

Types do not only rule out bad behaviour (lifebuoy) or are they inherently useful for program development (lamp).

Static vs dynamic typing

Don’t think of a type system as *ruling out* certain programs.

Instead, in some languages knowing a program is well-typed provides almost no useful information (JavaScript)…

In some languages, it rules out certain classes of problems related to memory bookkeeping (C or C++).

In others, it guarantees that a program will produce a value of a certain type without side-effects (Idealized Haskell).

Or even that a function definition satisfies a certain specification (Idris, Agda, Coq, …)

Types classify data. In some languages this classification is more expressive than others.

Static vs dynamic typing

Programmers have a common experience when *refactoring* or modifying strongly typed languages, such as Haskell.

- Any non-trivial change to the design of the program will effect the types involved.
- Once you have fixed the resulting type errors, your program ‘just works’.

Static types are a downpayment on program maintainability.

Static vs dynamic typing

Meaningful static types serve as a form of *machine-checked* documentation.

Comments can quickly go out of date or ‘bitrot’.

Carefully chosen types can expose a great deal of information about how to use a library and how to fit various pieces together:

Consider a function that exchanges between two currencies. Which type signature would you prefer?

```
exchange :: Double -> Double -> Double
exchange :: Amount c -> ExchangeRate c d -> Amount d
```

Static vs dynamic typing

Question: Where do you stand?

Question: What arguments did I forget to mention?

Mixing statically and dynamically typed languages

There are many different approaches to mixing static and dynamic typing:

- Gradual typing (some variables may be typed, others may not - Siek & Taha ’06)
- Soft typing (insert run time checks to coerce dynamic to static types - Cartwright & Fagan ’91)
- Embedding dynamic values in a typed language (Baars & Swierstra ’02)

Polymorphism

There are numerous other features of type systems that you will have encountered:

Parametric polymorphism (generics) – allow you to define functions that work on *all* types:

```
id :: forall a . a -> a
func map<T,U>(f : T -> U, xs : List<T>) -> List<U>
```

Ad-hoc polymorphism (overloading, protocols, traits) – allow you to define functions that work on *some* types:

```
sort :: Ord a => [a] -> [a]
func map<T : Ord>(xs : List<T>) -> T
```

When are two types equal?

This may seem like an obvious question, but it can be very subtle.

If I define two classes `A` and `B` in that define exactly the same methods and fields, are they equal?

Different languages behave differently – dynamically typed languages like Python will not distinguish between the two; statically typed languages like C# will.

We sometimes make the distinction between *structural equivalence* and *name equivalence*.

When are two types equal?

This question becomes even more subtle in the presence of more advanced type features.

Consider the application of a function `f : a -> c` to an argument `x : b` – when is this type correct?

- if `a` and `b` are a primitive type such as `int` or `bool`, we typically require them to be equal
- if `f` is polymorphic, we require the types `a` and `b` to *unify*
- if our language supports subtyping or inheritance, we require that `b` is a subtype of `a`.
- in Haskell we can define *type-level functions*, which may require further work to answer this question.

Variables

Every non-toy programming language has some notion of **variable**, that allow programmers to associate a name with some piece of data.

- the name of a method;
- the name of a class;
- the name of a function's argument;
- the name of a new type definition;
- …

Defining how to treat variables is one of the key design decisions in any programming language.

Variables

Here are a few of the design questions that show up:

- When is a variable in scope or not?
- How are variables stored in memory?
- Which variables may be mutated?
- What kind of values may be bound to a variable?
- How are arguments passed to a function call?

I'll try to cover some of the design space in the remainder of this class.

Variables and scoping

The **scope** of a declaration is the portion of the program where the declared variable may be used.

Different languages have a very different treatment of scope.

Consider Haskell:

```
foo :: Int -> Int -> Int
foo x y =
  let z = x + y in
  q
    where
    q = 2 * z
```

Variables and scoping

Or C:

```
public int foo(x:int, y: int) {
  int z = 4;
  for (int i = 0; i < 5; i++) {
    z = z + x;
  }
  return y * z;
}
```

Bind vs use

Looking at a program text, we can see many variables.

We need to distinguish between **binding occurrences** – that introduce a variable – and **applied occurrences** – that refer to previously bound variables.

```
\x -> let y = 1 in x + y
```

The lambda expression and let declaration are binding occurrences of `x` and `y`.

The body has two applied occurrences of x and y.

Variables and scoping

The rules for **scoping** describe to which binding occurrence an applied occurrence of a variable refers.

Typically the rules for variable scoping are reasonably straightforward.

A **block** is a program construct that delimits the scope of any declarations within it:

For example C, function bodies, source files, or explicit blocks (`{..}`) all start a new block.

Variables and scoping

In Haskell, function bodies, lambdas, let/where expressions, source files (and probably several others) introduce new blocks:

```haskell
let b = let c = 3 in c + 1
in b + b  -- c is no longer in scope
-- after this point, b is no longer in scope
```

Block-structured scoping

Most languages use some form of **block-structure** to determine scoping.

- if an applied occurrence of a variable is bound within the same block, it refers to that binding;
- otherwise, proceed to the enclosing block and search for binding occurrences of the variable there.

In this way we can determine to which `x` is being referred in expressions such as:

```
\x -> \y -> let x = y + x in x
```

Dynamic vs static scoping

A language is **statically scoped** if the body of a procedure is executed using the environment of the procedure's *definition*.

A language is **dynamically scoped** if the body of a procedure is executed using the environment of the procedure *call*.

Dynamic scoped languages, such as SmallTalk and early versions of Lisp, can make code much harder to understand – you can no longer study a function in isolation – and are no longer very popular.

Exceptions

Not all languages follow such block structure:

- Python is not statically scoped. The interpreter will not check if all applied occurrences of variables can be resolved before a program is executed.
- Overloading (using the same name for different values) complicates matters...

First-class value

Typically any value that can be associated with a variable or passed to a function is called a **first-class value**.

For example in Haskell:

- integers and lists are first class variables,
- patterns or data types are not.

Storage

To discuss how variables are stored in memory, we'll introduce a simple **storage model**:

- A store is a collection of **storage cells**, each of which has a unique **address**.
- Each storage cell is either **allocated** or **unallocated**.
- Every allocated storage cell has a contents, which may be a **value** or **undefined**.

This model is simplistic in many ways, but adequate for approximating how most programming languages store variables in memory.

Example: storage

To illustrate how storage changes, consider the following C code:

```
       /* no variables allocated */
int x; /* x is allocated, but undefined */
x = 5; /* x now stores the value 5 */
x++; /* x now stores the value 6 */
```

Composite variables

Some variables only take up a single storage cell – such as integers or booleans.

Others may take up many storage cells – such as objects or structs.

We will refer to the latter as **composite variables**.

Storing values

Different languages have different restrictions on what may be stored:

In Java and C#, you may only store primitive values or (pointers to) objects in variables.

But you cannot store functions or objects directly.

Assignment

What does the following code do?

```
MyObject a = new MyObject();
MyObject b = a;
```

There are two distinct possibilities:

- the variable `b` points to the same storage cell as `a` – no new memory is allocated (reference semantics)
- the contents of the object referred to by `a` is duplicated. The new storage cells may now be referred to using the variable `b` (copy semantics).

Copy semantics vs reference semantics

- Reference semantics save time and memory: there is no work necessary to copy values or ensure later writes do not cause interference between `a` and `b`
- Copy semantics ensure later changes to `a` do not effect `b` – the two values exist as separate and independent entities.

Copy semantics vs reference semantics

Most languages have some mix of copy semantics and value semantics:

- C++ uses copy semantics for primitive values and structs, but reference semantics for objects;
- Java uses reference semantics for (almost) everything; programmers can explicitly duplicate objects using the `clone` method.
- Swift uses copy semantics for everything (data types, structs, booleans, integers, etc.) – classes are the only entities with reference semantics.

Lifetime

All such variables are **created** (or allocated) and **destroyed** (or deallocated):

Creation typically happens when a variable is first declared.

Destruction may happen at different times, depending on the variable.

A variable's **lifetime** is the time when it may be accessed safely, after creation but before destruction.

- A *global variable* is destroyed when the program finishes;
- A *local variable* is destroyed when execution leaves the enclosing *block*.
- A *heap variable* is destroyed when the program finishes or earlier.

Summary

The aim of today's lecture was to revisit and define the core concepts that you should be familiar with when comparing programming languages.

Next time we'll start applying these concepts in the study of **domain specific languages**.