Concepts of programming languages

Lecture 3

Wouter Swierstra

Last time

The previous lecture was about introducing **terminology**

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

Today

Wrap up last lecture:

- How do programming languages (de)allocate variables?

Today's lecture will apply these concepts to study a **domain specific language**.

- How can we define the semantics of such languages?
- How can we reason about such semantics?
- How can we define rules for scoping?

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:

```
int foo() {
         /* 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 */
}        /* x is now unallocated */
```

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.

These are known 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;
b.mutate();
```

Question:

Is `a` modified or not?

Assignment

What does the following code do?

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

Question:

Is `a` modified or not?

That depends on how this language treats variables.

Reference vs value semantics

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

- Either the variable `b` points to the same storage cell as `a` – no new memory is allocated (reference semantics)
- Or 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 or value semantics).

The same consideration applies when calling a function with `a` as an argument – is it shared or duplicated?

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.

Memory management

Different languages treat the lifetime of heap variables differently.

- C or C++ require programmers to explicitly allocate and deallocate memory for heap variables.
- In Haskell or Java, heap variables that can no longer be accessed through are *garbage collected* – their storage may be reused for other heap variables.
- Swift counts the number of references to heap allocated variables. Once this drops to zero, it is deallocated.

The compiler construction course teaches more about how these different kinds of variables may be stored on the stack or the heap.

Garbage collection: pros and cons

Manual memory management is tricky to get right:

- refer to unallocated memory;
- try to deallocate the same memory twice;
- or forget to deallocate memory.

These all lead to bugs that are hard to track down.

Automatic garbage collection avoids such bugs, but takes time, which may lead to decreased or uneven performance.

Memory recap

- The simple, abstract storage model presented is useful enough to describe most common memory allocation behaviour.
- Reference semantics and copy semantics correspond are two different ways in which programming languages can handle references passed to a function call or assignment.
- Program variables have a finite *lifetime* and the corresponding memory may be deallocated when they are no longer needed. This deallocation may happen manually or automatically, depending on whether or not the language is garbage collected.

Evaluation order

What happens in the following function call?

```
length [2/0]
```

Does it throw an error or not?

Evaluation order

In a **strict** language, arguments to functions are fully evaluated – that is, you know that all function parameters will be instantiated with **values**.

In a **non-strict** language, arguments are only evaluated on demand.

A **lazy** language is a variation non-strict languages, that guarantees sharing of intermediate results.

Evaluation order – strict example

```
let x = 1 + 2 in x + x
```

Evaluates as:

- `let x = 1 + 2 in x + x`
- `let x = 3 in x + x`
- `x + x` where `x = 3`
- `3 + x` where `x = 3`
- `3 + 3` where `x = 3`
- `6`

Evaluation order – non-strict example

```
let x = 1 + 2 in x + x
```

Evaluates as:

- `let x = 1 + 2 in x + x`
- `x + x` where `x = 1 + 2`
- `(1 + 2) + x` where `x = 1 + 2`
- `3 + x` where `x = 1 + 2`
- `3 + (1 + 2)` where `x = 1 + 2`
- `3 + 3` where `x = 1 + 2`
- `6`

Evaluation order – lazy example

```
let x = 1 + 2 in x + x
```

Evaluates as:

- `let x = 1 + 2 in x + x`
- `x + x` where `x = 1 + 2`
- `(1 + 2) + x` where `x = 1 + 2`
- `3 + x` where `x = 3`
- `3 + 3` where `x = 3`
- `6`

Evaluating the expression associated with `x` is shared!

Evaluation order

We'll need to perform a more careful study of semantics to make the distinction between these different approaches more precise.

But these examples – together with your experience with Haskell – should give you some intuition.

Question:

In this example, the lazy and strict evaluation strategies were more or less equivalent. When will *lazy evaluation* require fewer reduction steps? When will *strict evaluation* require fewer reduction steps?

Evaluation order: trade-offs

Strict languages:

- Have a clear cost model – you can understand the complexity of a function by studying its definition.
- No reason to worry about undefined or bottom values – you know that when defining a function that takes an `int` as argument, it really is a number. **Question:** Why does this property not hold for Haskell?
- But can make other things harder, such as defining control flow operators or infinite/cyclic data structures.

Evaluation order: trade-offs

In a lazy languages it is safe to eliminate common subexpressions:

```
if c then X else 5
```

vs

```
let e = X in
if c then e else 5
```

If the expression `X` throws an error or takes very long to compute, these two programs will behave very differently in a strict language.

Evaluation order: trade-offs

In a lazy languages you can write your own 'control flow operators':

```
unless c t e = if not c then t else e
```

Evaluating a call to `unless` in a strict language will evaluate **all** its arguments – which is typically not what you want.

Similarly, in Haskell you can work with infinite lists as if they are finite (mapping over them, inspecting the first five elements, etc.) – this is **not** the case in a strict language.

Evaluation order

Each evaluation order has its own advantages and disadvantages.

Lazy languages like Haskell often provide primitives to force evaluation;

Strict languages often have some support for deferring computations.

There is no clear winner – I've added links to two blogposts by Bob Harper and Lennart Augustsson on the website.

What is a programming language?

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.

What is a programming language?

HyperText Markup Language (HTML) is the markup language used to describe webpages.

It is completely standardized.

There is a lengthy specification by the World Wide Web Consortium.

Question: Does this make it a programming language?

General purpose specific languages

Many programming languages with which you are already familiar (C#, Haskell, Javascript, …) are *general purpose* programming languages

They can (in principle) be used to write any application from a spreadsheet to computer game.

They may still have a very different syntax, type system, semantics, runtime, etc.

Unlike HTML, they are not tailored to one particular *domain*.

Domain specific language

A *domain specific language* is a programming language designed to solve one particular class of problems.

Examples include:

- Calculations in Excel;
- Websites using HTML;
- Formatting using CSS;
- Markup using LaTeX;
- Build processes using Make;
- Simple games using Game Maker Language;
- Regular expressions;
- ...

DSL example: regular expressions

Regular expressions are used to define a collection of strings – typically used to define a pattern such as:

- files ending in `.txt`
- html tags surrounded by ange brackets `< ... >`
- IP addresses in a certain range
- trailing whitespace in your editor

They are not as expressive as full blow parsers for context free languages, but useful for all kinds of smaller applications.

Regular expression: abstract syntax

```
r :=
    c          // The language with one character
  | 0          // The empty language
  | 1          // The language only the empty string
  | r + r'     // Choice of either r or r'
  | r · r      // Sequencing two expressions
  | r⋆         // Repeat r zero or more times
```

More complex definitions

Using these basics, we can assemble more complex regular expressions easily enough:

```
-- recognize re once or not at all
r? = r + 1

-- recogize re one or more times
r+ = r · r⋆
```

Concrete vs abstract syntax

Many different editors or tools have different concrete syntax for the same concepts:

- Perl and Python write `l | r` for our `l + r`
- Gnu grep and emacs write `l \| r` for `l + r`
- Some regexp implementations provide special support for matching the end of a line, the beginning of a word, etc.

The nice thing about working with the *abstract* syntax is that we can *abstract* from such implementation details.

Regular expression semantics

Question: What *semantics* are we interested in for regular expressions?

Regular expression semantics

Question: What *semantics* are we interested in for regular expressions?

We want to know when a string `s` is matched by a regular expression `re`.

That is, when is `s` in the language generated by `re`?

(Here we use language to refer to the set of strings matched by `re`).

Regexp semantics: take one

We could specify our semantics using natural language:

- the string consisting of the character `c` is in the language generated by the regular expression `c`.
- when the string `s` is in the language generated by the regular expression `re` it is also in the language generated by the regular expression `re + re'` for all regexps `re'`.
- ...

Regexp semantics: take one

We could specify our semantics using natural language:

- the string consisting of the character `c` is in the language generated by the regular expression `c`.
- when the string `s` is in the language generated by the regular expression `re` it is also in the language generated by the regular expression `re + re'` for all regexps `re'`.
- …

This is verbose, easy to get wrong, and potentialy ambiguous.

Regexp semantics: take two

Instead, we will define a **relation** between regular expressions and strings, expressing when a string is in the language generated by a regular expression.

Binary relation: a binary relation on a set A is a collection of ordered pairs, that is, a subset of the set $A \times A$.

Example: the set $\{ (i, i) \mid i \in N\}$ defines the equality relation on natural numbers.

Example: the set $\{ (xs, xs +\!\!+\, ys) \mid xs, ys \in String\}$ defines when one string is a prefix of another.

Relations

The drawback of this definition is that it relies on manipulating sets directly.

Defining any non-trivial relation is really hard – try defining the semantics of regular expressions in this style yourself...

Instead, such relations are usually defined using **inference rules**, just as those for logic that you are already familiar with, such as the *modens ponens* rule:

$$\frac{p \to q \qquad p}{q}$$

'When both $p \to q$ and p holds, we can deduce q.'

Defining relations

To warm up, consider the problem of defining a relation between two strings xs and ys, describing when xs is a prefix of ys.

$$\frac{}{\text{prefixOf}([], xs)}$$

$$\frac{\text{prefixOf}(xs, ys)}{\text{prefixOf}(x : xs, x : ys)}$$

This defines two *rules*, characterizing the relation.

The judgement below the line is the *conclusion*. The judgements above the line are the *hypotheses*.

A rule without hypotheses is an *axiom*.

Derivations

Question: How can we use these rules to prove the following statement holds?

prefixOf$(\texttt{ab}, \texttt{abba})$

Derivations

Question: How can we use these rules to prove the following statement holds?

prefixOf$(\texttt{ab}, \texttt{abba})$

We can use these inductive rules like lego-blocks to piece together a prooftree that a required statement holds:

- each node is an instance of the rules defining our relation;
- each leaf is an axiom.

Such a prooftree is called a **derivation**.

Back to relations as sets

What is the connection between the rules we saw for prefixOf and the notion of relation as sets of pairs?

We can connect the two easily enough: define the relation

$$\{(xs, ys) \mid \text{ there is a derivation of prefixOf}(xs, ys)\}$$

But as we shall shortly, the inductive definition of relations using inference rules has additional advantages…

Back to regular expressions

Let's now define a set of rules $xs \in \mathsf{L}(r)$ that describe when a string xs is in the language generated by r.

I'll go over the rules one by one.

Characters

$$\text{Char}\,\frac{}{\mathsf{c} \in L(\mathsf{c})}$$

The only string in the language generated by the regular expression consisting of a single character c , is the string consisting of the character c.

We have named the rule (Char) in order to distinguish it from other rules, and refer to it in the derivations that we construct.

The empty string

$$\text{One}\ \frac{}{\varepsilon \ \in\ \mathsf{L}(1)}$$

The only string that inhabits the language 1 (the language containing only the empty string) is the empty string (written ε).

Choice

$$\text{Left}\,\frac{xs \ \in \ \mathsf{L}(r)}{xs \ \in \ \mathsf{L}(r + r')} \qquad \text{Right}\,\frac{xs \ \in \ \mathsf{L}(r)}{xs \ \in \ \mathsf{L}(r' + r)}$$

To show that a string xs is in the language generated $r + r'$ it suffices to show:

- xs is in the language generated by r;
- or xs is in the language generated by r'.

In either case, we can use one of the two rules above to construct the complete derivation.

Sequential composition

Question: What should the rule for sequential composition be?

In other words, when is a string in the language generated by $r \cdot r'$?

Sequential composition

Question: What should the rule for sequential composition be?

In other words, when is a string in the language generated by $r \cdot r'$?

$$\text{Seq}\,\frac{xs \;\in\; \mathsf{L}(r) \quad ys \;\in\; \mathsf{L}(r')}{xs +\!\!+\, ys \;\in\; \mathsf{L}(r \cdot r')}$$

Here we assume that we can split the string into two parts xs and ys.

But we don't prescribe *where* to split the string, but rather specify that such a splitting must exist.

Iteration

$$*\text{Stop}\frac{}{\varepsilon \ \in \ \mathsf{L}(r^*)} \qquad *\text{Step}\frac{xs \ \in \ \mathsf{L}(r) \quad ys \ \in \ \mathsf{L}(r^*)}{xs \mathbin{+\!\!+} ys \ \in \ \mathsf{L}(r^*)}$$

- The empty string is in the language generated by r^*;
- If xs is in the language generated by r and ys is in the language generated by r^*, then $xs \mathbin{+\!\!+} ys$ is in the language generated by r^*.

Note: the *Step rule is *recursive* – it assumes a derivation exists for $ys \ \in \ \mathsf{L}(r^*)$ to define when a string is in the language generated by r^*.

Empty language

Empty language

There is no rule describing when a string is in the empty language.

By definition, no strings are in the empty language, hence there is no corresponding rule.

Summary of rules

$$\text{Char}\,\frac{}{\mathsf{c} \in \mathsf{L}(\mathsf{c})} \qquad \text{One}\,\frac{}{\varepsilon \in \mathsf{L}(1)}$$

$$\text{Left}\,\frac{xs \in \mathsf{L}(r)}{xs \in \mathsf{L}(r + r')} \qquad \text{Right}\,\frac{xs \in \mathsf{L}(r)}{xs \in \mathsf{L}(r' + r)}$$

$$\text{Seq}\,\frac{xs \in \mathsf{L}(r) \quad ys \in \mathsf{L}(r')}{xs \mathbin{+\!\!+} ys \in \mathsf{L}(r \cdot r')}$$

$$*\text{Stop}\,\frac{}{\varepsilon \in \mathsf{L}(r^*)} \qquad *\text{Step}\,\frac{xs \in \mathsf{L}(r) \quad ys \in \mathsf{L}(r^*)}{xs \mathbin{+\!\!+} ys \in \mathsf{L}(r^*)}$$

Derivations

Question: Can you use these to show that:

$\mathtt{ab} \in \mathsf{L}((\mathtt{a} + \mathtt{b})^*)$?

Derivations

Question: Can you use these to show that:

$\mathtt{ab} \in \mathsf{L}((\mathtt{a} + \mathtt{b})^*)$?

We can construct a suitable derivation easily enough.

Semantics of regular expressions

We have given an abstract syntax for regular expressions.

We have seen a collection of rules for proving that a string is in the language generated by a regular expression.

For each syntactic construct, we have zero, one or two rules.

Each rule is a ‘building block’ that can be used to write a derivation, showing a string is in the language generated by a regular expression.

This does **not** give an algorithm – only a specification.

Uniqueness

If we have two derivations that $\texttt{xs} \in \mathsf{L}(r)$, are they equal?

Uniqueness

If we have two derivations that $\mathtt{xs} \in \mathsf{L}(r)$, are they equal?

Not necessarily.

In particular, we have a proof that $\mathtt{xs} \in \mathsf{L}(r^*)$ and the empty string is in the language generated by r – we can apply the *Step rule many times.

Proofs!

Question: How can we prove that if $xs \in \mathsf{L}(r)$ then also $xs \in \mathsf{L}(r + 0)$?

Proofs!

Question: How can we prove that if $xs \in \mathsf{L}(r)$ then also $xs \in \mathsf{L}(r + 0)$?

This is relatively easy: we can use our assumption and the rules to construct the desired proof tree.

Proofs!

Question: How can we prove that $xs \in \mathsf{L}(r + 0)$ then $xs \in \mathsf{L}(r)$.

Proofs!

Question: How can we prove that $xs \in \mathsf{L}(r+0)$ then $xs \in \mathsf{L}(r)$.

This is a harder proof. We can't construct a proof directly from our assumptions – there is no rule that we can apply to build the desired proof.

Induction

One first attempt might be to do induction on the string xs...

Induction

One first attempt might be to do induction on the string xs...

But suppose that our induction hypothesis is $xs \in \mathsf{L}(r + 0)$.

How can we prove $(x : xs) \in \mathsf{L}(r + 0)$?

We don't know what r is and cannot use our hypothesis.

Our proof is stuck.

Rule induction

Instead, we need to do induction on something else.

We use **rule induction**, induction on the *derivation* of $xs \ \in \ \mathsf{L}(r + 0)$, to prove this.

Question:

What rules could have been used to construct $xs \ \in \ \mathsf{L}(r + 0)$?

Rule induction

Instead, we need to do induction on something else.

We use **rule induction**, induction on the *derivation* of $xs \in \mathsf{L}(r + 0)$, to prove this.

Question:

What rules could have been used to construct $xs \in \mathsf{L}(r + 0)$?

$$\text{Left}\ \frac{xs \in \mathsf{L}(r)}{xs \in \mathsf{L}(r+0)} \tag{1}$$

$$\text{Right}\ \frac{xs \in \mathsf{L}(0)}{xs \in \mathsf{L}(r+0)} \tag{2}$$

Rule induction

If $xs \in \mathsf{L}(r + 0)$ holds, we know it must be constructed with one of the two rules on the previous slide.

1. In the first case, we know $xs \in \mathsf{L}(r)$ also holds – the bigger proof was built by first proving this and the applying the Left rule.
2. In the second case, we know $xs \in \mathsf{L}(0)$ – but this is impossible! There is no rule to prove a string is an element of the empty language, hence we have reached a contradiction.

Rule induction

Question: Can you use these to show that:

$\mathtt{c} \notin \mathsf{L}((\mathtt{a} + \mathtt{b})^*)$?

Rule induction

Question: Can you use these to show that:

$\texttt{c} \notin \mathsf{L}((\texttt{a} + \texttt{b})^*)$?

To do so, we assume $\texttt{c} \in \mathsf{L}((\texttt{a} + \texttt{b})^*)$ and try to derive a contradiction.

Which rule could have been used to construct this proof?

No rule is applicable, so the proof cannot exist.

Rule induction

These last examples show that we can use rule induction to show that a string is *not* in the language, by showing that no valid derivation can exist.

Similarly, we can use rule induction to establish general properties of algebraic expressions $xs \in \mathsf{L}(r \cdot (s + t))$ holds if and only if $xs \in \mathsf{L}((r \cdot s) + (r \cdot t))$.

Rule induction

Rule induction is a powerful proof technique that forms central to any study of programming languages.

Rules such as those for regular languages are typically used to define:

- when a program is well-scoped;
- when a program is well-typed;
- what the result of evaluating a program is;
- ...

Any proof about these properties uses rule induction.

Extensions

There are plenty of examples of extensions to this language:

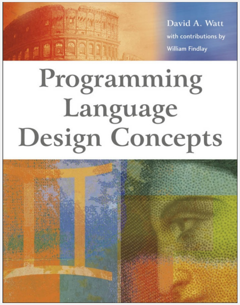
- Match a any single character, or any character in a certain range.
- Match a given expression exactly n times;
- Match a given expression between n and m times;
- Match the end or beginning of a line;
- Do not match a given expression;
- Or introduce *marked expressions* that may be re-used.

Exercise: Try formalizing the rules for any of the above.

Summary

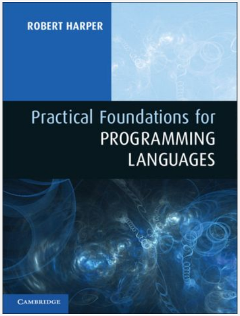
- We can use rules to define the static and dynamic semantics of languages.
- To reason about such rules, we can use rule induction. This allows us to prove formal properties of our languages.
- Getting the rules just right is not at all easy!

Reading – concepts



Programming language design concepts,David A. Watt, Chapters 2-5

Reading – rule induction



Practical Foundations of Programming languages,Robert Harper, Chapters 1-3