Concepts of programming languages

Lecture 3

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

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

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?

Defining programming languages

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.

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.

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`

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

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 = 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.

Evaluation order: trade-offs

Strict languages

- Have a clear cost model;
- No reason to worry about undefined or bottom values;
- But can make it harder to define control flow operators or infinite/cyclic data structures.

Lazy languages:

- safe to eliminate common subexpressions;
- short-circuiting computations without cost;
- reasoning about (space) complexity is very subtle.

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 from both sides 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.

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

```
data RE where
  -- the empty language
  Zero : RE
  -- the language containing the empty string
  One : RE
  -- the language containing a single character
  C : Char -> RE
  -- choice of two regular expressions
  Or : RE -> RE -> RE
  -- sequencing two regular expressions
  Seq : RE -> RE -> RE
  -- repeating a regexp zero or more times
  Star : RE -> RE
```

More complex definitions

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

```
-- recognize re once or not at all
maybe : RE -> RE
maybe re = Or re One

-- recogize re one or more times
plus : RE -> RE
plus re = Seq re (Star r)

...
```

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Metalanguage vs object language

Note: I'm using Haskell here as a **metalanguage** to study the **object language** of regular expressions.

- The **object language** is the language we are currently studying.
- The **metalanguage** is the language we are using to study the object language.

We could many different metalanguages to study the same object language; or use the same metalangauge to study different object languages.

Haskell notation is precise enough for our purposes, and you're already familiar with it. All the above definitions can be made mathematically precise.

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 `Or l r`
- Gnu grep and emacs write `l \| r` for `Or l r`
- Some regexp flavours allow the use of `r?` and `r+` for our `maybe` and `plus` operations respectively; others don't.
- 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

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

Regular expression semantics

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 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 `Or 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 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 `Or 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 (according to wikipedia) 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 $\{(xs, xs \mathbin{+\!\!+} ys) \mid xs, ys \in \mathrm{String}\}$ 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.

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

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

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*.

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Derivations

Question: How can we use these rules to prove that prefixOf(“`ab`”, “`abba`”)?

Derivations

Question: How can we use these rules to prove that prefixOf(“`ab`”, “`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

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

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.

We will write:

- ε for the empty string;
- $xs \mathbin{+\!\!+} ys$ for the concatenation of two strings xs and ys;
- $r \cdot r'$ for the sequential composition of regular expressions (`And`);
- $r + r'$ for the choice between two regular expressions (`Or`);

Semantics of regular expressions

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

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

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

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

Derivations

Question: Can you use these to show that:

$\texttt{aba} \in \mathsf{L}((\texttt{a} + \texttt{b})^*)$?

Derivations

Question: Can you use these to show that:

$\texttt{aba} \in \mathsf{L}((\texttt{a} + \texttt{b})^*)$?

We can construct a suitable derivation easily enough.

Proofs!

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

Proofs!

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

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 $xs \in \mathsf{L}(r \cdot (s + t))$ holds if and only if $xs \in \mathsf{L}((r \cdot s) + (r \cdot t))$.

Proofs!

Question: How can we prove that $xs \in \mathsf{L}(r \cdot (s + t))$ holds if and only if $xs \in \mathsf{L}((r \cdot s) + (r \cdot t))$.

This is a harder proof. We can't construct a proof directly from our assumptions.

A challenge

Let's start with a simpler problem. Suppose $xs \in \mathsf{L}(r + 0)$ – where 0 corresponds to the empty regular expression.

How can we show that $xs \in \mathsf{L}(r)$?

Induction

One first attempt might be to do induction on the string $xs\ldots$

Induction

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

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

This doesn't tell us anything useful about $(x : xs) \in \mathsf{L}(r)$ – in other words, the induction hypothesis cannot be used and 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.

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

$$\frac{xs \in \mathsf{L}(r)}{xs \in \mathsf{L}(r+0)} \quad (1)$$

$$\frac{xs \in \mathsf{L}(0)}{xs \in \mathsf{L}(r+0)} \quad (2)$$

In the first case, we have established our goal and we're done. What about the second?

Rule induction

We claim that if $xs \in \mathsf{L}(0)$ we can derive a contradiction and our proof is done.

Once again, using rule induction, which rule could have been used to construct $xs \in \mathsf{L}(0)$?

Rule induction

We claim that if $xs \in \mathsf{L}(0)$ we can derive a contradiction and our proof is done.

Once again, using rule induction, which rule could have been used to construct $xs \in \mathsf{L}(0)$?

No rule fits! Hence the proof cannot exist and we're done.

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:

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

To do so, we assume $\mathtt{c} \in \mathsf{L}((\mathtt{a} + \mathtt{b})^*)$ 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.

Let's look at the last one in a bit more detail.

Marked expressions

Putting regular expressions in parentheses `\(r\)` creates a **marked expression**. Such an expression is matched whenever `r` matches.

Writing `\1` refers to the string matched by previous marked expression; `\2` refers to the one before that; etc. Most regular expressions allow for at most 9 marked expressions.

Example: What does the expression `\((a|b|c|d)*\)\1` match?

Marked expressions

Putting regular expressions in parentheses `\(r\)` creates a **marked expression**. Such an expression is matched whenever `r` matches.

Writing `\1` refers to the string matched by previous marked expression; `\2` refers to the one before that; etc. Most regular expressions allow for at most 9 marked expressions.

Example: What does the expression `\((a|b|c|d)*\)\1` match?

Palindromes over the alphabet `a`, `b`,`c`, `d`.

Marked expressions

To avoid too many parentheses, we will sometimes write $\mathsf{mark}(r)$ rather than `\(r\)`; similarly, we will use $\mathsf{ref}(i)$ to refer to the backreference to the i-th previous marked expression.

How can we add new rules for marked expressions?

$$\frac{xs \;\in\; \mathsf{L}(r)}{xs \;\in\; \mathsf{L}(\mathsf{mark}(r))}$$

Marked expressions

To avoid too many parentheses, we will sometimes write $\mathsf{mark}(r)$ rather than `\(r\)`; similarly, we will use $\mathsf{ref}(i)$ to refer to the backreference to the i-th previous marked expression.

How can we add new rules for marked expressions?

$$\frac{xs \ \in \ \mathsf{L}(r)}{xs \ \in \ \mathsf{L}(\mathsf{mark}(r))}$$

$$\frac{???}{xs \ \in \ \mathsf{L}(\mathsf{ref}(i))}$$

We have no idea what the string is that has been matched by the i-th marked expression.

Environments

To solve this, we need to remember which strings matched with which marked expressions we have seen.

We record this information in an **environment**, a list of the strings matched by the marked expressions we have encountered so far.

We now rephrase our relation to pass around an environment of marked expressions. As a first attempt, we will write $\Gamma \vdash xs \ \in \ \mathsf{L}(r)$ when we can show that r is in the language generated by the regular expression r in the environment Γ.

Using the environment

If we assume that all marked expressions are recorded in the environment, we can use this to assign meaning to backreferences:

$$\frac{}{\Gamma \vdash \Gamma_i \ \in \ \mathsf{L}(\mathsf{ref}(i))}$$

Here Γ_i refers to the i-th elment of the environment – remember this is a *string*.

Adapting rules

Most of the rules we saw previously do not use the environment at all.

We can adapt some of the rules easily enough:

$$\frac{\Gamma \vdash xs \ \in \ \mathsf{L}(r)}{\Gamma \vdash xs \ \in \ \mathsf{L}(r + r')} \qquad \frac{\Gamma \vdash xs \ \in \ \mathsf{L}(r)}{\Gamma \vdash xs \ \in \ \mathsf{L}(r' + r)}$$

Extending the environment

Whenever we encounter a marked expression, we can optimistically remember it in the environment:

$$\frac{\Gamma; xs \vdash xs \ \in \ \mathsf{L}(r)}{\Gamma \vdash xs \ \in \ \mathsf{L}(\mathsf{mark}(r))}$$

Extending the environment

Whenever we encounter a marked expression, we can optimistically remember it in the environment:

$$\frac{\Gamma; xs \vdash xs \ \in \ \mathsf{L}(r)}{\Gamma \vdash xs \ \in \ \mathsf{L}(\mathsf{mark}(r))}$$

(This is wrong. We'll see why shortly)

One environment is not enough..

Yet if we only have a single environment, how should we handle the rule for sequential composition of regular expressions?

$$\frac{\Gamma \vdash xs \ \in \ \mathsf{L}(r) \quad \Gamma \vdash ys \ \in \ \mathsf{L}(r')}{\Gamma \vdash xs +\!\!+ ys \ \in \ \mathsf{L}(r \cdot r')}$$

All the marked expressions in r should be **in scope** in r'.

Yet both branches of the rule share the same environment…

Something is wrong.

Something is very wrong

If we look more closely at our original rule for marked expressions:

$$\frac{\Gamma; xs \vdash xs \ \in \ \mathsf{L}(r)}{\Gamma \vdash xs \ \in \ \mathsf{L}(\mathsf{mark}(r))}$$

We see that the only place where we may refer to the string xs that is matched with the marked expression r is in the derivation of $xs \ \in \ \mathsf{L}(r)$ itself!

We really need to rethink our rules.

Two environments

We rephrase our relation one more time to pass around two environments of strings. We will write $\Gamma, \Delta \vdash xs \ \in \ \mathsf{L}(r)$ when we can show that r is in the language generated by the regular expression r in the environment Γ. Any new strings matched by r will be recorded in Δ.

Marking: revisited

$$\frac{\Gamma, \Delta \vdash xs \;\in\; \mathsf{L}(r)}{\Gamma, (\Delta; xs) \vdash xs \;\in\; \mathsf{L}(\mathsf{mark}(r))}$$

Here we use Γ to match r; after doing so, we extend the environment Δ so any later matches may refer to r.

Sequential composition

At least our rules for sequential composition looks reasonable:

$$\frac{\Gamma, \Delta \vdash xs \ \in \ \mathsf{L}(r) \quad \Delta, \Phi \vdash ys \ \in \ \mathsf{L}(r')}{\Gamma, \Phi \vdash xs +\!\!+ ys \ \in \ \mathsf{L}(r \cdot r')}$$

Backreferences

The obvious rule looks up the i-the variable in the context of visible bindings Γ:

$$\frac{}{\Gamma, \Gamma \vdash \Gamma_i \;\in\; \mathsf{L}(\mathsf{ref}(i))}$$

What about

- marked expressions and repetition, `r*`?
- nested marked expressions?
- references within marked expressions?
- references to a marked expression within itself?
- …

What about

- marked expressions and repetition, `r*`?
- nested marked expressions?
- references within marked expressions?
- references to a marked expression within itself?
- ...

At least writing down the semantics gives us an answer to these questions.

Lessons learned: scoping is hard

It seems like such an innocent extension: being able to refer to previously marked regular expressions.

But **scoping rules can be surprisingly hard** – specifying exactly what is meant by this extension is not at all trivial.

But doing so forces us to think about all the corner cases.

And this was just adding marked expressions to regular languages, now think how hard it is to add types to JavaScript/lambda expressions to Java/dependent types to Haskell/. . .

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!