Concepts of programming languages

Lecture 4

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Last time

In the last lecture we studied an example 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?

This time

- Study an example domain specific language (SVG).
- What different approaches are their to implementing domain specific languages?

Back to rule induction…

In your first logic course, you learn:

- natural numbers are defined as a the least set closed under:
  - a constant zero and
  - a unary operation successor;
- induction on natural numbers requires:
  - a base case for zero;
  - an inductive case for successor.

Back to rule induction…

In the Functional Programming course, you learn:

- lists are defined as a data type with two constructors:
  - nil, corresponding to the empty list;
  - cons, prepending a new element to an existing list.
- proofs by induction over lists require two cases:
  - a base case for the empty list;
  - an inductive case for non-empty lists.

Back to rule induction

In this course, we saw:

- a relation defined inductively with several rules (constructors):

$$c \in \mathsf{L}(c) \qquad \frac{xs \in \mathsf{L}(r)}{xs \in \mathsf{L}(r + r')} \qquad \ldots$$

- an induction principle on such derivations. Given any proof, we can 'pattern match' to determine which rule was used to construct it.

What do I need to know for the exam?

I expect you to be able to:

- Given a set of rules, construct a derivation or proof;
- Proof simple algebraic properties – if $xs \in \mathsf{L}(r)$ then also $xs \in \mathsf{L}(r^*)$? (This is easy – it just requires constructing a 'bigger' proof tree).
- Use rule induction to prove simple properties, given a set of inference rules.
- Define a set of rules for a given problem. Typically this will be some variation of something that we study in class.

I'll try to set up a lab session next week with some examples.

What do I need to know for the exam?

Don’t both memorizing the rules for regular expressions.

What do I need to know for the exam?

Don’t both memorizing the rules for regular expressions.

But you should be able to read and understand any given set of rules.

What do I need to know for the exam?

Don’t both memorizing the rules for regular expressions.

But you should be able to read and understand any given set of rules.

Don’t worry too much about about understanding the last section with marked expressions and references.

What do I need to know for the exam?

Don’t both memorizing the rules for regular expressions.

But you should be able to read and understand any given set of rules.

Don’t worry too much about about understanding the last section with marked expressions and references.

We’ll see other examples later on in the course.

Theory or practice?

How much maths will this course cover?

The Goldilocks amount: not too little, not too much.

I want you to study several different modern programming languages…

Theory or practice?

How much maths will this course cover?

The Goldilocks amount: not too little, not too much.

I want you to study several different modern programming languages…

… but be familiar with the foundations of program language design.

I'll try to alternate a bit between more practical and more theoretically minded lectures.

What is a DSL?

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

Examples include:

- Regular expressions;
- Spreadsheet calculations in Excel;
- Websites using HTML;
- Formatting using CSS;
- Markup using LaTeX or Markdown;
- Build processes using Make;
- …

What makes a language domain specific?

There isn't a precise definition.

A language like Matlab, R, or Excel are all designed to help with certain calculations (matrices, statistics, or spreadsheets).

But you can write all kinds of programs in them.

Question: Where do you draw the line between a domain specific and general purpose programming language?

Characteristics of a DSL

- A formal language (as opposed to natural language)
- Limited expressiveness, often witnessed by lack of complex control structure (loops, conditionals) and abstractions (functions, objects, modules).
- Tailored to solve problems in a particular domain.

Example: Scalable vector graphics

Scalable Vector Graphics (SVG) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation. The SVG specification is an open standard developed by the World Wide Web Consortium (W3C) since 1999.

From Wikipedia.org

SVG example

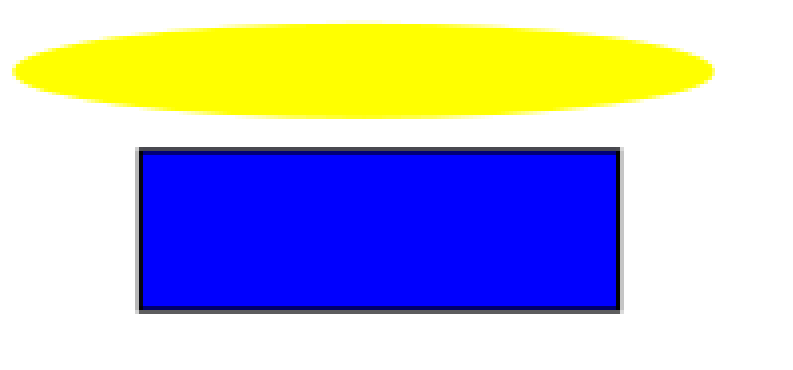

Figure 1: SVG Example

```
<svg height="500" width="500">
  <ellipse cx="240" cy="50" rx="220" ry="30"
           style="fill:yellow" />
  <rect width="300" height="100"
        style="fill:rgb(0,0,255);stroke:rgb(0,0,...
        x="100" y="100">
</svg>
```

SVG in practice

This is a lot more work than Microsoft Paint, but…

- Like their name suggests, SVG graphics are *scalable*
- SVG is supported by many browsers;
- SVG can be manipulated by JavaScript, allowing the graphics to change with user interaction.
- SVG is textual. It is more compact than some other image formats and can be kept under version control.

There are lots of reasons that you may want to use SVG over other image formats. For many *icons*, for instance, smooth scaling is extremely important.

SVG as a programming language

Let's try to abstract some of the syntactic pecularities of SVG and focus instead on the structure of SVG documents:

- Each SVG document contains a sequence of XML nodes.
- Most of these nodes correspond to simple shapes (ellipse, rectangle, polygon,…);
- These nodes may be grouped `<g>...</g>` and referenced `<use>`
- There are further features for applying image filters, animations, etc.

SVG Basics

SVG supports a handful primitive objects (rectangles, ellipses, lines, polygons, polylines, text, etc.)

```
<circle cx="50" cy="50" r="40" stroke="black"
        stroke-width="3" fill="red" />
```

Each of these objects can be customized in various ways by providing different attributes:

- x and y positions;
- width and height;
- stroke color, width, etc.

Note that everything is a string; essentially all SVG images are untyped.

SVG Transformations

We can also specify how to transform certain elements:

```
<ellipse transform="scale(2,5),translate(100,-100)"
      cx="100" cy="50" rx="40" ry="20" fill="grey" />
```

Here we can scale and translate across both the x- and y-axes.

This is particularly useful when modifying an existing picture in some way.

SVG Scoping

We can group several such shapes together using the `g` tags:

```
<g id="MyGroup">
...
</g>
```

Referencing

We can refer to any group or SVG element using the `use` tag.

```
<use xlink:href="MyGroup" transform="translate(120,0)"
```

In this way we can capture certain recurring patterns.

The element being referenced (here `MyGroup`) must be defined elswhere – for instance by the declaration `<g id="MyGroup">...`

`defs` vs `g`

There are two tags to group SVG images:

- `defs` allows you to name a composite image without rendering it;
- `g` renders and names an image.

Fancier features

Besides these basics, there are plenty of more advanced features:

- applying image filters;
- animations;
- adding shadows;
- filling and gradients;
- …

These are quite complex from some perspective – but I would argue that do not make the language more expressive.

What are the first class objects?

- We can associated `ids` with any SVG node;
- And refer to these nodes using the `use` tag;
- But we cannot (easily) refer to strings, integers, styles, attributes, etc. using SVG.

SVG repetition & conditional

Suppose I want to duplicate `n` ellipses:

1. Draw a single ellipse and name it;
2. Refer to this ellipse and translate it;
3. Copy and paste, repeat.

Clearly, this is not a good idea.

Similarly, there is no conditional statement, such as an if-then-else.

There is almost no computation that you can do.

What about…

What if I want to:

- draw a grid?
- abstract over the size of my image? Or its alignment?
- draw a complicated fractal, snowflake, or spiral?

In each of these examples, there is some common pattern or computation.

SVG does not offer you the ability to express this.

Stand-alone DSL: limitations

By their very nature, domain specific languages are not suitable for arbitrary computations.

They offer limited abstractions and limited opportunities for re-use. SVG has no control flow operators.

You could add iteration to SVG, but that would complicate the language – you may well want to perform various computations, add arithmetic, loops, types,…

And then you end up implementing a general purpose programming language.

SVG limitations

Using only SVG we cannot interact with users or compute new graphics on the fly:

- Create a new object wherever the user clicks the mouse;
- Build objects with random values for their attributes;
- Allow objects to have their attributes modified (nontrivially) by users;
- Allow moving objects to have their directions or velocities adjusted (nontrivially) by the user;
- Detect the distance between moving objects on the screen ;
- …

Alternatives

Write a program in Javascript (or any other language) that generates 7 circles, a Fibonacci spiral, etc.

This is fine for small examples, but…

As your program becomes more complex, you end up having to use more and more SVG features;

At some point, it might be worth investigating how to generate SVG in a more principled fashion.

Generating SVG

Let's try to write a Haskell library for generating SVG files.

We'll start with the simplest design possible, and refine it as we go along:

```
type SVG = String
type Attrs = String

circle :: Attrs -> SVG
circle attrs = "<circle " ++ attrs ++ " />"

attr :: Show a => String -> a -> String
attr a val = a ++ "=\"" ++ show val ++ "\" "
```

Generating circles

Even with these simple definitions, we can already write image generators:

```
-- Draw a circle with a given size, translated by dx
circleSizeXYDx :: Int -> Int -> Int -> Int -> SVG
circleSizeXYDx sz x y dx = circle attrs
  where
  attrs = attr "r" sz
          ++ attr "cx" (x + dx)
          ++ attr "cy" (y)

-- Draw n circles of size sz next to one another
myCircles :: Int -> Int -> SVG
myCircles nr sz = unlines $ take nr $
  map (circleSizeXYDx sz 100 100) [0,2*sz..]
```

Generating circles

We can now generate however many circles we want:

```
> myCircles 20 10
```

Yields the following image:

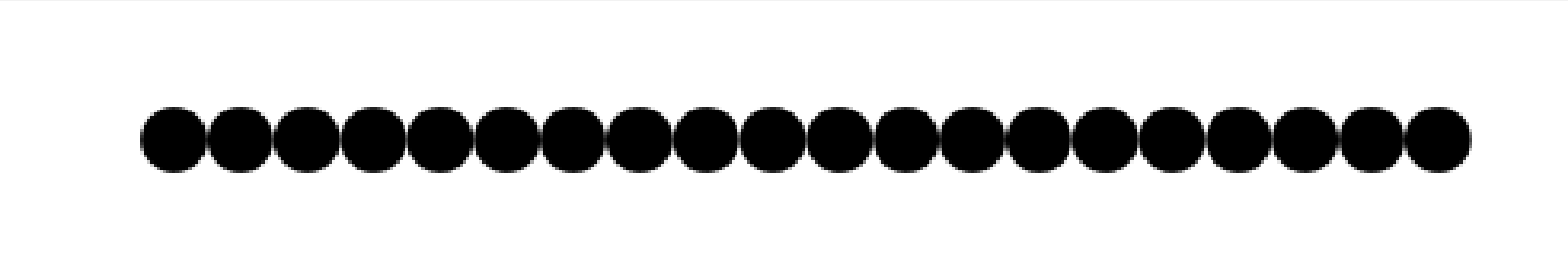

Figure 2: Circles

Embedded domain specific language

This is an example of an **embedded domain specific language** – here we have taken an existing language (SVG) and shown how it can be *embedded* in a general purpose language (Haskell).

We will sometimes refer to the **object language** as the DSL being defined; the **host language** is the language in which we are writing the embedding.

Adding types

Note that we needed to compute new positions for our circles.

To do this computation, we needed to work with *integers* not *strings*.

As a result, we should maybe revisit our design:

```
data Attr =
  CX Int
  | CY Int
  | R Int
  | ...

instance Show Attr where
  show (CX x) = attr "cx" x
  ...
```

More types…

Suppose we want to modify or transform an existing SVG image:

```
translate :: (Int,Int) -> SVG -> SVG
```

Although we can add such attributes when the SVG image is first *defined*, there is 'nothing' we can do to modify an existing SVG image.

We would need to parse the string to reverse engineer the shapes.

Or wrap the current image in a `def` tag, refer to it through `use` and translate it there…

Both of these solutions are a bit unsatisfactory.

Other semantics

As it stands we have a single value representing SVG images: their string representation.

But what if we want to *inspect* existing SVG images?

- Is this image black-and-white?
- How big is this image?
- Does it contain circles?
- Is there a blue square at position (10,100)?
- …

All of these are hard to answer without a better representation.

A deep embedding

As we did for attributes, we can introduce a new data type:

```
newtype SVG = SVG [Element]
data Element =
  Circle [Attr]
  | Rectangle [Attr]
  | Group SVG [Attr]
  | ...
```

We refer to this as a **deep embedding** – we have defined a separate data type representing the AST of our object language in our host language.

Semantics

Given such a data type, we can define arbitrary computations over it.

For example, we can still compute a `String` corresponding to the SVG image:

```
showSVG : SVG -> String
showSVG (SVG elts) = unlines $ map showElt

showElt : Element -> String
showElt (Circle attrs) =
   "<circle " ++ showAttrs attrs ++ ">"
...
```

We refer to such functions as an *interpretation* or a *semantics* of SVG.

Alternative semantics

But we can also define other semantics, such as a function that counts the number of circles in an SVG image:

```
countCircles :: SVG -> Int
countCircles (SVG elts) = sum (map count elts)
  where
  count :: Element -> Int
  count (Circle _) = 1
  count (Rectangle _) = 0
  count (Group g attrs) = countCircles g
  ...
```

We could only do this previously if we have specific information about the way our SVG data is represented as a string.

Shallow vs deep embedding

In a shallow embedding:

- we work directly with the semantics;
- it may be difficult to define alternative semantics;
- you easily define new (primitive) functions.

In a deep embedding:

- we write functions to construct abstract syntax trees;
- we can define alternative semantics by recursing over these trees;
- adding new primitives requires extending our data type and existing semantics.

Beyond SVG

What we have done so far is try to write a Haskell library modelling SVG.

But if we're in Haskell anyhow, we are free to choose our own combinators or language.

And *compile* this into SVG. This gives us more freedom in the design of the language that we want.

I'll sketch one possible way to do this.

Diagrams – example

One thing we could do is avoid absolute positioning altogether:

```
data Diagram =
  Primitive Shape
  | Besides Diagram Diagram
  | Below Diagram Diagram
  | Attributed Attr Diagram
  | Align (Float,Float) Diagram

data Shape = Circle Radius | Rectangle Width Height
  ...
```

Attributes record colour, but no size or position information.

Derived combinators

We can define derived combinators for assembling complex images:

```
hcat :: [Diagram] -> Diagram
hcat = foldr Besides empty

vcat :: [Diagram] -> Diagram
vcat = foldr Below empty

circle :: Radius -> Diagram
circle r = Primitive (Circle r)

circles = hcat $ replicate 20 (circle (Radius 5))
```

Of course, the generated SVG will still need to contain positions…

Diagrams – calculating size

As we have a deep embedding, we can compute the size of a diagram:

```
size :: Diagram -> (Width, Height)
size (Circle (Radius r)) = (With r, Height r)
size (Rectangle w h) = (w,h)
size (Besides l r) =
  let (wl,hl) = size l
      (wr,hr) = size r
  in (wl + wr, hl `max` hr)
...
```

Diagrams – computing coordinates

But we can also use this size information to compute coordinates:

```
type Canvas = Canvas Width Height Position

fit :: (Float, Float) -> Size -> Canvas -> Canvas
fit (alignX, alignY) size c = ...
```

This requires a bit of programming with coordinates, but nothing too complex.

Diagrams – generating SVG

We can now generate SVG diagrams easily enough:

```
render :: Diagram -> Canvas -> SVG
-- draw a circle on the origin with suitable size
render (Primitive (Circle (Radius r))) c = ...
-- split the available canvas in two and recurse
render d@(Besides d1 d2) c =
  let (c1,c2) = splitX (size d1 / size d) c in
  render c1 d1 ++ render c2 d2
...
```

The important observation is that we can **compute** coordinates if we would like to.

Embedding Domain Specific Languages

This shows that we can have different approaches:

- embed SVG directly in Haskell;
- define our own language – tailored to our purposes – and compile this to SVG.

Note that this second approach relies on having a deep embedding - we compute the size of a diagram and its rendering.

This harder to accomplish with only a shallow embedding.

Designing embedded languages

- What are the primitive concepts?
- What are the derived combinators?
- What are the interperations that you would like to define?
- What safety guarantees would you like to enforce? Types?

How you answer these questions, leads to a different EDSL for SVG generation.

Which host language?

I've chosen Haskell as my host language:

- algebraic data types and pattern matching enable deep embedding and different interpretations;
- lightweight syntax & operators make allow code and examples to fit on one slide;
- you are already familiar with the syntax and semantics of Haskell.

The same ideas work in many other languages – but the syntax & semantics of the **host language** has a huge impact on what the embedded language is like.

- What type safety do we want to provide?
- What other bugs or errors do we want to rule out?

Embedded Domain Specific Languages

Let's revisit our deep embedding of SVG:

```
newtype SVG = SVG [Element]
data Element =
  Circle [Attr]
  | Rectangle [Attr]
  | Group SVG [Attr]
  | ...
```

Although we have mentioned how to group elements into a single diagram, how can we extend this to cope with `use` tags?

Adding use tags

We can add a new `Use` constructor to our language:

```
newtype SVG = SVG [Element]
data Element =
  Circle [Attr]
  | Rectangle [Attr]
  | Group SVG [Attr]
  | Use [Attr]
  | ..
```

Question: How can we use this to construct ill-formed SVG diagrams?

Adding use tags

We can construct ill-formed SVG diagrams by referring to undefined IDs:

```
oops = Use [attr href "UNDEFINED"]
```

This kind of problem might not be too bad in the context of SVG – we won't see the image we were expecting and this shouldn't be too hard to debug.

Dynamic scope checking

One way to avoid referencing undefined images is to write a function that checks that all references are well-defined.

We can do this in two steps:

1. Traverse the SVG image to collect all the defined ids;
2. Check that all use tags refer to an existing name.

Dynamic scope checking - I

We can traverse an SVG diagram easily enough, collecting all the id's that are defined:

```
getBoundIDs : SVG -> [Name]
getBoundIDs (SVG elts) = concatMap getIDs elts
  where
  getIDs (Circle attrs) = getID attrs
  getIDs (Group svg attrs) = getIDd attrs ++ getBoundI
  ...
```

I've left out some helper functions, like `getID` that tries to find the "id" tag associated with any given node.

Dynamic scope checking - II

Given a list of names, we can now check that all `use` tags refer to an existing id:

```
scopeCheck : [Name] -> SVG -> Bool
scopeCheck ctx (SVG elts) = all (map (check ctx) elts)
  where
  check :: [Name] -> Element -> Bool
  check ctx (Circle _) = True
  check ctx (Group svg) = scopeCheck ctx svg
  check ctx (Use attrs) = (getAttr "id" attrs) `elem`
  ...
```

But this is a *dynamic check* – there is nothing in (Haskell's) type system that rules out the creation of illegitimate SVG diagrams.

The problem of variable binding

But what about embedding more complex (programming) languages?

How can we handle **variable binding** in our object language?

Haskell has its own notion of variables – couldn't we just use those?

What if we want to add even more semantic checks to ensure we can only construct well-formed object terms?

We'll see compare and contrast several different solutions next week.

Summary

- Stand alone DSLs have limited expressive power – often by design.
- We can *embed* a DSL in a general purpose host language.
- Doing so raises several design questions:
  - deep or shallow?
  - typed or not?
  - which interpretations should you support?
  - how to treat variable binding?

These design issues boil down to:

- What static guarantees do you want to enforce?
- What dynamic semantics do you want to support?