Domain-specific languages
and functional programming

Claus Reinke
Computing Laboratory
University of Kent at Canterbury

Motivation – why languages?

Languages in Computer Science

Are often under-represented:

- they are your tools, you ought to know a few, but that's it, they are *"really just tools"*..
- unless your area of research is in language design and implementation, you'll probably *"really want to focus on other things"*..
  - such as computers (hardware)
  - or programs (software)
  - or theory (noware?-)

Languages in Computer Science

Are often under-valued:

- you ought to have a well-stocked toolbox, preferably covering different paradigms
- you ought to know how to use these tools, (programming/reasoning, patterns, design)
- you need ways to express your ideas, to formulate and to communicate your theories
- without good notations, proofs become complicated, ideas remain inaccessible; software remains unwritten, computers unused

Languages in Computer Science

Are often under-valued:

- you ought to have a well-stocked toolbox, preferably covering different paradigms
- you ought to know how to use these tools, (programming/reasoning, patterns, design)
- you need ways to express your ideas, to formulate and to communicate your theories
- without good notations, proofs become complicated, ideas remain inaccessible; software remains unwritten, computers unused

Languages in Computer Science

Should really be central:

- you can think without an explicit notation, but you cannot use computers as helpful tools
- "computer users" don't want to think about computers, but about their problem domain
- languages tailor-made for your domain can help your thinking, even without computers
- computer systems (theory, hard- & software) can support the use of domain-specific languages, in many ways

Languages in Computer Science

Should really be central:

- you can think without an explicit notation, but you cannot use computers as helpful tools
- "computer users" want to think about their problem domain, not about computers
- languages tailor-made for your domain can help your thinking, even without computers
- computer systems (theory, hard- & software) can support the use of domain-specific languages, in many ways

Languages in Computer Science

Languages are the interfaces between computers and users, between computer scientists and experts in other domains

- domain experts can focus on using their languages to work on their problems
- computer scientists can focus on supporting languages through computer systems

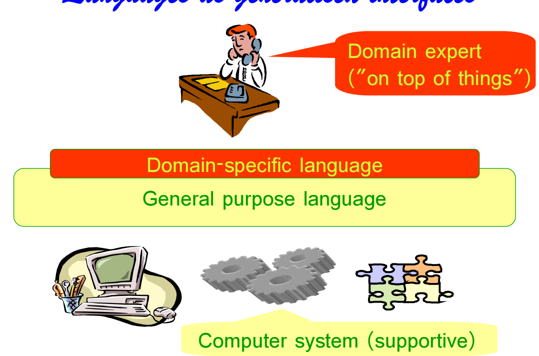
Language details differ between domains, but there is also common structure, which can focus computer science research

In pictures

User interfaces, standard approach

User interfaces, standard approach

Languages as generalised interfaces



Session outline

(enough propaganda - let's get going;-)

What you'll see

Implementing DSLs in functional languages
- algebraic types for abstract grammars
- parser combinators for concrete grammars
- semantics-based interpreters
- embedding DSLs

Examples
- λ as a small functional language
- Mini-Prolog
- DSLs for graphics: a calculus of cubes, SpaceTurtles (Mini-Logo in 3d)

What you should take away

DSLs are a useful way of organising thoughts and computer systems
- they serve as generalised interfaces between system components, and between users and developers

DSLs in functional languages: nice&easy
- some techniques and examples..
- host language does not usually get in the way
- good support for abstraction makes it possible to think in terms of composition of little languages
- embedded DSLs inherit rich framework of generic language features (abstraction, recursion, DSL elements are "first-class" data ..)

What you should take away

DSLs are a useful way of organising thoughts and computer systems
- they serve as generalised interfaces between system components, and between users and developers

DSLs in functional languages: nice&easy
- some techniques and examples..
- host language does not usually get in the way
- good support for abstraction makes it possible to think in terms of composition of little languages
- embedded DSLs inherit rich framework of generic language features (abstraction, recursion, DSL elements are "first-class" data ..)

Content

(time to wake up?-)

Processing a computer language

A typical compiler pipeline:

scanning (lexical analysis)
parsing (syntax analysis)
static analysis (types, scopes, ..)
optimisation
code generation

followed by a runtime system

code execution
memory management, ..

Meta-languages for language processing

A typical compiler pipeline:
- scanning (lexical analysis)
- parsing (syntax analysis)
- static analysis (types, scopes, ..)
- optimisation
- code generation

followed by a runtime system
- code execution
- memory management, ..

Regular expressions
BNF, or railroad diagrams
Inference rules
Attribute grammars
Control-flow graphs, data-flow graphs,..
Transformation Rules, rewriting

A small functional language

Concrete syntax
Abstract syntax
Parsing

A small example language, syntax

The λ-calculus grammar:

```
<expr> → <var>
       | (<expr> <expr>)
       | λ<var>.<expr>
```

An abstract syntax, as a data type:

```
data Expr = Var String
          | App Expr Expr
          | Lam String Expr
```


A small example language, syntax

A parser, from concrete to abstract syntax:

```
import Monad
import ParserCombinators
expr = var `mplus` app `mplus` abstr
var = do { v <- litp "var expected" isAlpha
         ; return $ Var [v] }
app = do { lit '(' ; e1 <- expr ; e2 <- expr ; lit ')'
          ; return $ App e1 e2 }
abstr = do { lit '\\' ; Var v <- var ; lit '.' ; e <- expr
           ; return $ Lam v e }
```


What's going on? - Monads

For monad, think "monoid with extras" ..

Monoid - type, associative binary operator, and its unit

e.g., functions, composition, identity:

id . (+2) . id . (*3) . id ≅ \x-> x*3+2

Favourite usage: sequential composition ("a then b")

Monad – type constructor, assoc. binary op., its unit

e.g., Maybe, guarded composition, failure

Just "hi" >>= \v-> Just (v++"ho") ≅ Just "hiho"

Nothing >>= \v-> Just (v++"ho") ≅ Nothing

One usage: sequential composition with hidden parts

What's going on? - Monads

For monad, think "monoid with extras" ..

Monoid - type, associative binary operator, and its unit

e.g., functions, composition, identity:

id . (+2) . id . (*3) . id ≅ \x-> x*3+2

Favourite usage: sequential composition ("a then b")

Monad – type constructor, assoc. binary op., its unit

e.g., Maybe, guarded composition, failure

return "hi" >>= \v-> return (v++"ho") ≅ return "hiho"

fail "oops" >>= \v-> return (v++"ho") ≅ fail "oops"

One usage: sequential composition with hidden parts

What's going on? - Grammars

BNF is a language for describing grammars..

- It offers literals (terminal symbols), non-terminals, sequences, alternatives, and recursion (plus lots of refinements)
- Monadic composition will give us sequences; recursion and non-terminals come for free through the embedding in Haskell; so we only need alternatives and literals
- **MonadPlus – a Monad with an "addition" and its unit**

  for Maybe, mzero is just failure, mplus selects first non-failure:

  return "hi" \`mplus\` x ≅ return "hi"

  fail "oops" \`mplus\` x ≅ x

  for lists, mzero is [], mplus is (++)

  {can you work out >>= ? }

What's going on? - Parsers

BNF is a language for describing grammars..

So what about parsing literals?

Parsing a text means recognising syntactically correct prefixes, chopping them off the input, and generating an abstract syntax tree:

```
lit c = \s->
  case s of
    (c':s') | c==c' -> return (c,s')
   _                -> fail m
```

We can hide the string handling in a new Monad:

What's going on? - Parser combinators

```
module ParserCombinators where
data MParser m s a = P{ applyP :: s -> m (a,s) }
instance Monad m => Monad (MParser m s) where
  a >>= b  = P $ \s->
   applyP a s >>= \(ar,s')-> applyP (b ar) s'
  return r = P $ \s-> return (r,s)

instance MonadPlus m => MonadPlus (MParser m s)
  where
  mzero       = P $ const (fail "no parse")
  a `mplus` b = P $ \s->
   applyP a s `mplus` applyP b s
```


A small functional language

Reduction rules

Reduction strategies

Interpreters

A small example language, semantics

λ-calculus reduction semantics:

$(\lambda v.M)\ N \rightarrow_{\beta} M[v \leftarrow N]$

{context-free reduction}

refined by a reduction strategy:

$C_{nor}[(\lambda v.M)\ N] \rightarrow_{\beta,nor} C_{nor}[M[v \leftarrow N]]$

{context-sensitive, normal-order reduction}

$C_{nor}[] \rightarrow [] \mid (C_{nor}[]\ \text{<expr>})$

{reduction contexts; contexts as expressions with a hole }

A small example language, semantics

β-reduction in Haskell:

```
beta (App (Lam v m) n) =
  return $ substitute v n m
beta _ = fail "not a redex"
```

reduction strategy in Haskell:

```
norStep e@(App m n) = beta e `mplus`
  (norStep m >>= (\m'-> return (App m' n)))
norStep _ = fail "not an application"

nor e = (norStep e >>= (\e' -> nor e'))
  `mplus`(return e)
```

A small example language, summary

That wasn't too bad, was it?-)

- implemented a tiny language in no time
- instead of focussing on details of the implementation language, we focussed on other small languages (BNF, contexts, rewrite rules) that are used in the application domain ⇒ most of our code is reusable
- our implementation is not too far from a formal specification of the problem (concrete and abstract syntax, reduction rules and reduction strategy)

A small example language - what next?

Lots of room for improvement:

- there are more grammar constructs (*,+, [],..); add parser combinators for those
- extend the language (let, arithmetic, strings, booleans, lists, i/o?, ..)
- add a type (inference) system
- change the reduction strategy (reduction under λs, applicative order reduction)
- add tracing of reductions

A small example language - what next?

Other things to try:

- can you modify the combinators so that the error messages do not get lost?
- if you look closely, only the literal parsers do any "real" work (from string to AST), the combinators just coordinate. Can you modify things so that a single grammar specification can coordinate both parsing and unparsing?
- write parsers and runtime systems for Simon's picture language, for (propositional) logic, or for your own DSL

A small example language - embedding?

λ-calculus is a FPL, Haskell is a FPL, so why do we have to write an interpreter at all? Can't we simply reuse Haskell's λs?

```
Prelude> (\x->x x) (\x->x x)
ERROR: Type error in application
*** Expression     : x x
*** Term           : x
*** Type           : a -> b
*** Does not match : a
*** Because        : unification would give infinite type

This naive approach fails - Haskell's additional safety net,
   the type system, does restrict the expressiveness a bit..
```


A small example language - embedding?

For an embedding of full λ in a typed language, we need a type that is a solution to the equation U = U → U

```
data U = In{ out ::U -> U}

((out (In $ \x->((out x) x)))
  (In $ \x->((out x) x)))

This one works (no result, though ..☺)
```


Languages, languages, ..

Parser combinators, pretty-printing combinators, attribute grammars, strategy combinators for rewriting; monads (a language "pattern"); hardware specification languages (BlueSpec,Hawk,Lava,..); FRP (functional reactive programming): Fran (animation), Frob (robotics), Fvision (computer vision), FRP-based user interface libraries (FranTk, Frappe, Fruit,..), Lula (stage lighting);

Languages, languages, ..

GUI combinators (Fudgets, Haggis, ..); DSLs for graphics (G-calculus, Pan, ..) and music (both sound and scores; Haskore, Elody,..); Prolog; Coloured Petri Nets; stock market (composing contracts involving options, composing price history patterns); VRML (virtual reality); XML (data interchange); HTML/CGI (web pages); SQL (database query language);..

Mini-Prolog

Embedding Prolog in Haskell

Prolog, by example

A predicate-logic programming language: you define predicates via facts and rules, then ask Prolog to find solutions to queries about your "knowledge base":

```
app([],Y,Y).
app([X | XS],Y,[X | ZS]):-app(XS,Y,ZS).

?- app(X,Y,[1,2]).
X=[], Y=[1,2];
X=[1],  Y=[2];
X=[1,2], Y=[];
no
```


Prolog, by example

Where's the logic?

```
∀Y:app([],Y,Y) ⇐ true
∀X,XS,Y,ZS:
app([X|XS],Y,[X|ZS])⇐ app(XS,Y,ZS).

?- ∃X,Y: app(X,Y,[1,2]).
X=[], Y=[1,2];
X=[1],  Y=[2];
X=[1,2], Y=[];
no
```


Prolog, by example

Closed-world assumption and de-sugaring:

```
∀A,B,C:
App(A,B,C) ⇔
  (A=[] ∧ B=C)
  ∨
  ∃X,XS,Y,ZS:
  (A=[X|XS] ∧ C=[X|ZS] ∧ app(XS,Y,ZS))
```

where "=" is unification, i.e., equation solving with variable instantiation on both sides

Prolog, embedded in Haskell

```
app a b c = (do { a === Nil ; b === c })
            +++
            (exists "" $ \x->
             exists "" $ \xs->
             exists "" $ \zs->
             do { a === (x:::xs) ; c === (x:::zs)
                ; app xs b zs })

x2 = exists "x" $ \x-> exists "y" $ \y->
     app x y (Atom "1":::Atom "2":::Nil)

Prolog> solve x2
[("y_1",1:::2:::[]),("x_0",[])]
[("y_1",2:::[]),("x_0",1:::[])]
[("y_1",[]),("x_0",1:::2:::[])]
```

Prolog, embedded in Haskell

What's going on here? Nothing much, actually (about two pages of code). Mostly, just our good old friends, Monads:

```
∧ : Sequential composition
∨ : Alternative composition
true  : return
false : fail
⇔ : =
Predicates : substitution transformers
Unification : explicit code
∃v : fresh variables
```


Domain-specific languages for graphics

Embedding a subset of VRML

Fractal cubes

Space turtles

(a bit of Fran in 3d, perhaps)

VRML

VRML'97: Virtual Reality Modeling Language

International Standard ISO/IEC 14772-1:1997

- ``a file format that integrates graphics and multimedia''
- ``3D time-based space that contains graphic and aural objects that can be dynamically modified through a variety of mechanisms''
- portable, human-readable text format
- several free browsers and commercial tools available (Windows, Mac, Linux, Java,..)

Embedding VRML in Haskell

FP and graphics fit together nicely, but:

how do we get easy access to graphical output?

1. define an abstract syntax for VRML as a Haskell data type
2. now we can construct VRML scenes using standard Haskell programming techniques
3. finally, the functionally constructed AST of a VRML scene is translated into a VRML file
4. which can then be rendered by any VRML browser

Embedding VRML in Haskell, AST

```
data Scene = ..
  | Anchor{children::[Scene],url::[String]}
  | Appearance{material::Material,texture::Texture}
  | Box{} | Sphere{radius::Double}
  | Cone{bottomRadius::Double,height::Double}
  | Cylinder{radius::Double,height::Double}
  | Shape{appearance::Appearance,geometry::Geometry}
  | Group [Scene] | Translate Pos Scene | Rotate O Scene
  | ScaleIn Dim Double Scene
  | Invisible
  | ..
  | AudioClip Bool [String]
  | Text Material [String]
  | ..
```


Embedding VRML in Haskell, shortcuts

```
colour r g b  = defaultMaterial{diffuseColour=Colour r g b}
white = colour 1 1 1
red   = colour 1 0 0
blue  = colour 0 0 1
green = colour 0 1 0

shape col geom = Shape{appearance=Appearance{
                                    material=col
                                   ,texture=EmptyTexture}
                       ,geometry=geom}
xAxis o a = o 1 0 0 a
yAxis o a = o 0 1 0 a
zAxis o a = o 0 0 1 a
a .||. b = Group [a,b]
r .&. c = Rotate r c
(.*.) d s c = ScaleIn d s c
(.+.) d s c = Translate (offset d s) c
```

A DSL for fractal cube graphics

Now that we have access to 3d graphics, we can use it directly or build on that basis

- some folks at a french Computer Music Research Laboratory have developed several functional languages for the creation of music and graphics, but their nice Graphic-Calculus implementation is only available on Macintosh
- we can recreate most of that very quickly, by combining features from our functional host language and from the VRML backend

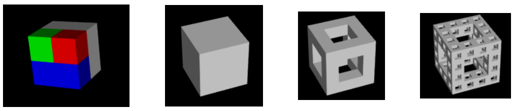
A DSEL for fractal cube graphics

```
u = 1.0 -- unit size
-- some basic coloured cubes to start with
redC   = XYZ .*. u $ shape red   Box{}
greenC = XYZ .*. u $ shape green Box{}
whiteC = XYZ .*. u $ shape white Box{}

-- the cube combinators, rescaling to unit size;
-- a left of b, a on top of b, a before b
a .|. b = X .*. 0.5 $
   (X .+. (-0.5*u) $ a) .||. (X .+. (0.5*u)  $ b)
a .-. b = Y .*. 0.5 $
   (Y .+. (0.5*u)  $ a) .||. (Y .+. (-0.5*u) $ b)
a ./. b = Z .*. 0.5 $
   (Z .+. (0.5*u)  $ a) .||. (Z .+. (-0.5*u) $ b)
```


A DSL for fractal cube graphics

```
((greenC .|. redC) .-. blueC) ./. whiteC
```



```
rcube 0 = Cache "rcube0" $ shape white Box{}
rcube n = Cache ("rcube"++(show n)) $
      (s1 ./. s2) ./.  (s2 ./. s1)
  where
    s2  = (s11 .-. invisible) .-. (invisible .-. s11)
    s1  = (s12 .-. s11) .-. (s11 .-. s12)
    s11 = (white .|. invisible) .|. (invisible .|. white)
    s12 = (white .|. white) .|. (white .|. white)
    white = rcube (n-1)
```


A DSL for fractal cube graphics

Exercise (hint: remember Simon's pictures?)

try to define the following cubes:

- diagonal cube
- checkerboard cube, size n
- can you do a pyramid?

SpaceTurtles

Logo in 3d

Logo

Created by Seymour Papert, based on experience with Lisp, but specifically targeted to enable children to learn

He needed a simple, still powerful language, so that children could start doing interesting things in it, right from the start.

We need a simple, still powerful language, so that you can implement it, and do interesting things with it, right from the start..

Logo

Turtle talk (controlling a cursor with position, orientation, and drawing pen):

```
forward d, backward d,
turnright a, turnleft a,
turnup a, turndown a,
spinright a, spinleft a,
pendown, penup
```

Control structures:

```
repeat n cmds, ifelse c cmds cmds,
to procname params cmds, procname
```


Logo

Can you do it?-) You have:

```
import VRML
```

That gives you:

- Line graphics (relative, cartesian coordinates)
- Rotating scenes around xAxis, yAxis, zAxis
- Translating scenes along X, Y, Z

Hint: focus on terminating turtle paths only

- Use Haskell to construct the path of the turtle as an embedded VRML scene

Conclusions

That's it!

What you should take away

DSLs are a useful way of organising thoughts and computer systems

- they serve as generalised interfaces between system components, and between users and developers

DSLs in functional languages: nice&easy

- some techniques and examples..
- host language does not usually get in the way
- good support for abstraction facilitates thinking in terms of composition of little languages
- embedded DSLs inherit rich framework of generic language features (abstraction, recursion, DSL elements are "first-class" data ..)

Languages are generalised interfaces