DSL Summer School Intro

Martin Odersky
EPFL

Why DSLs?

1. Give users ways to express themselves in a notation tailored to their domain.

→ Better abstraction

→ Fewer errors

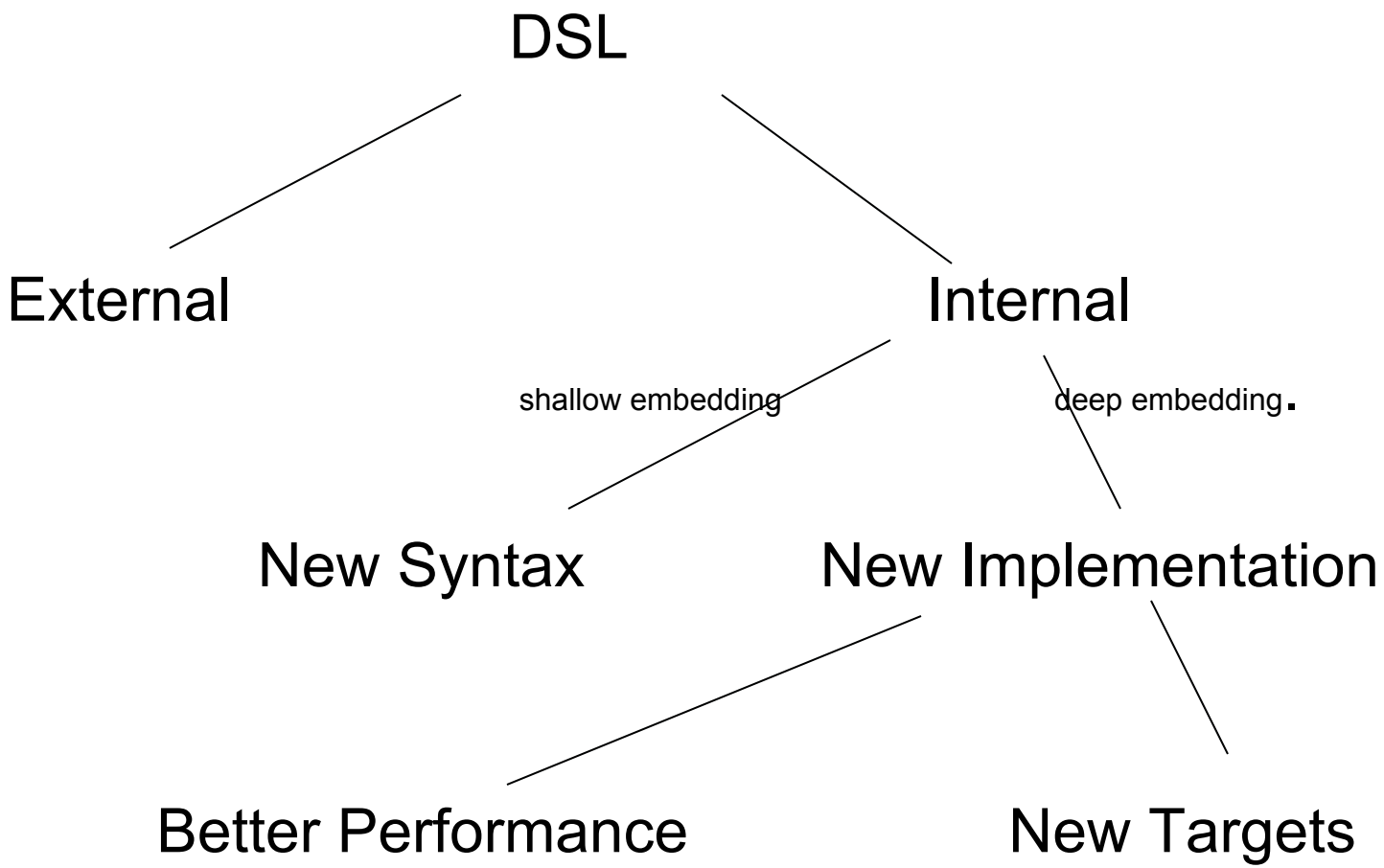
→ More concise and clearer notation (sometimes)

2. Allow new implementation capabilities that are specialized for a domain.

→ Higher performance

→ Non-traditional target platforms

Implementation Decisions



Scala and DSLs

Scala has proven to be fertile ground for building DSLs

- can be molded into new languages by adding libraries (domain specific or general)

See: “Growing a language” (Guy Steele, 1998)

Ecosystem provides many tools (parser combinators, macros, etc), to define new DSLs.

A Growable Language

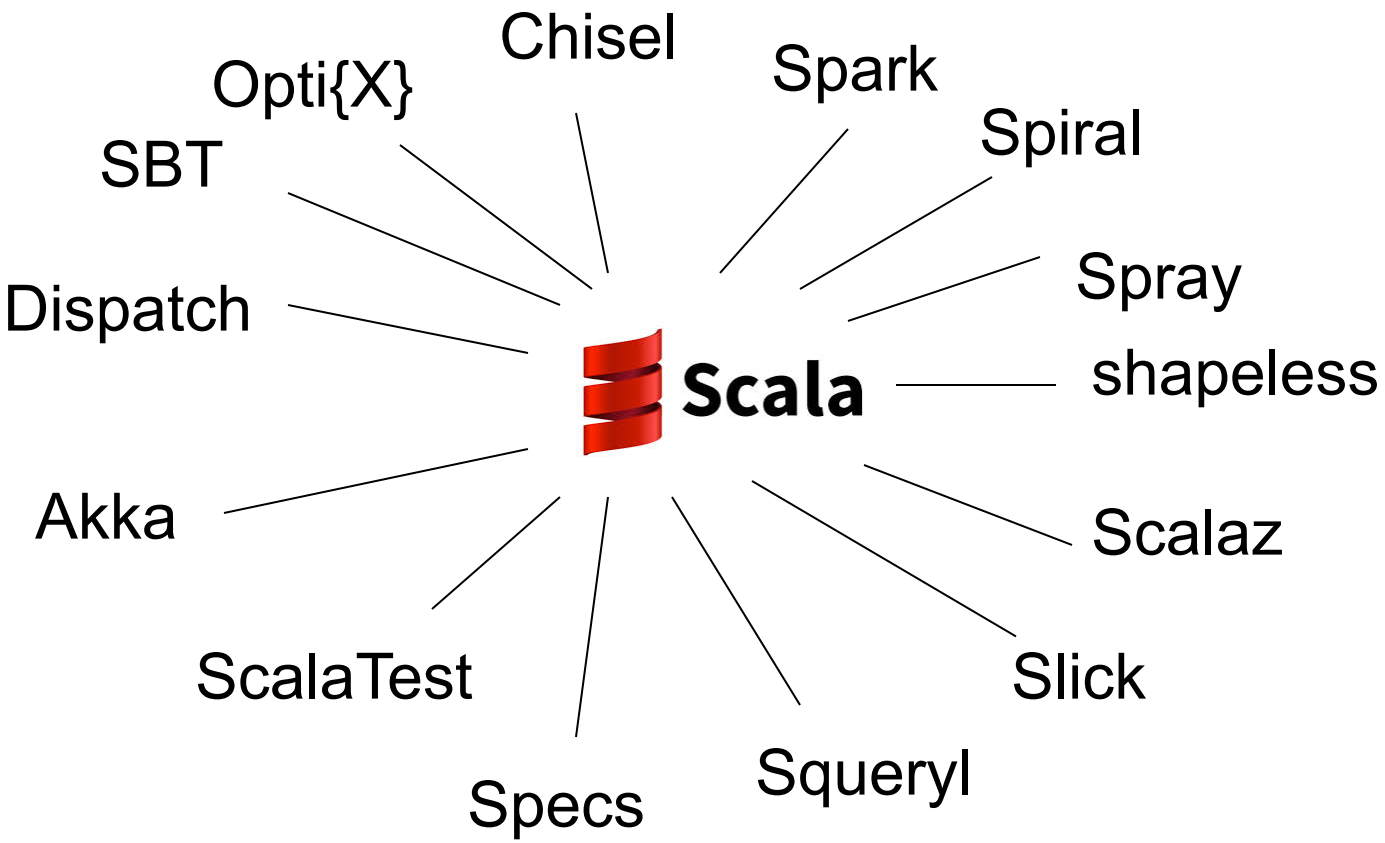
- Flexible Syntax
- Flexible Types
- User-definable operator
- Higher-order functions
- Implicits

...

Make it relatively easy to build new DSLs on top of Scala

And where this fails, we can always use the (experimental) macro system.

A Growable Language



Growable = Good?

In fact, it's a double edged sword.

Growable is great because it does not presume that the language designers know everything a priori about the "right way" to program.

But it has its challenges:

- DSLs can fracture the user community
- Besides, no language is liked by everyone, no matter whether its a DSL or general purpose.
- Host languages get the blame for the DSLs they embed.

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
  case ((word,path),n) => (word,(path,n))
```

```
sc.textFile("data/crawl")
.map { line =>
  val array = line.split("\t", 2)
  (array(0), array(1))
}
.flatMap {
  case (path, text) =>
   text.split("""\W+""") map {
    word => (word, path)
   }
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
```

```
}
.map {
  case (w, p) => ((w, p), 1)
}
.reduceByKey {
  (n1, n2) => n1 + n2
}
.map {
  case ((word,path),n) => (word,(path,n))
}
.groupByKey
.mapValues { iter =>
  iter.toSeq.sortBy {
   case (path, n) => (-n, path)
  }.mkString(", ")
}
.saveAsTextFile(argz.outpath)

sc.stop()
```

((word1, path1), n1)
((word2, path2), n2)
...

Pitfalls

- The Lisp Curse
- Syntactic Flexibility
- Interpretation Indirection
- Tooling

The Lisp Curse

“The power of Lisp is its own worst enemy”

~

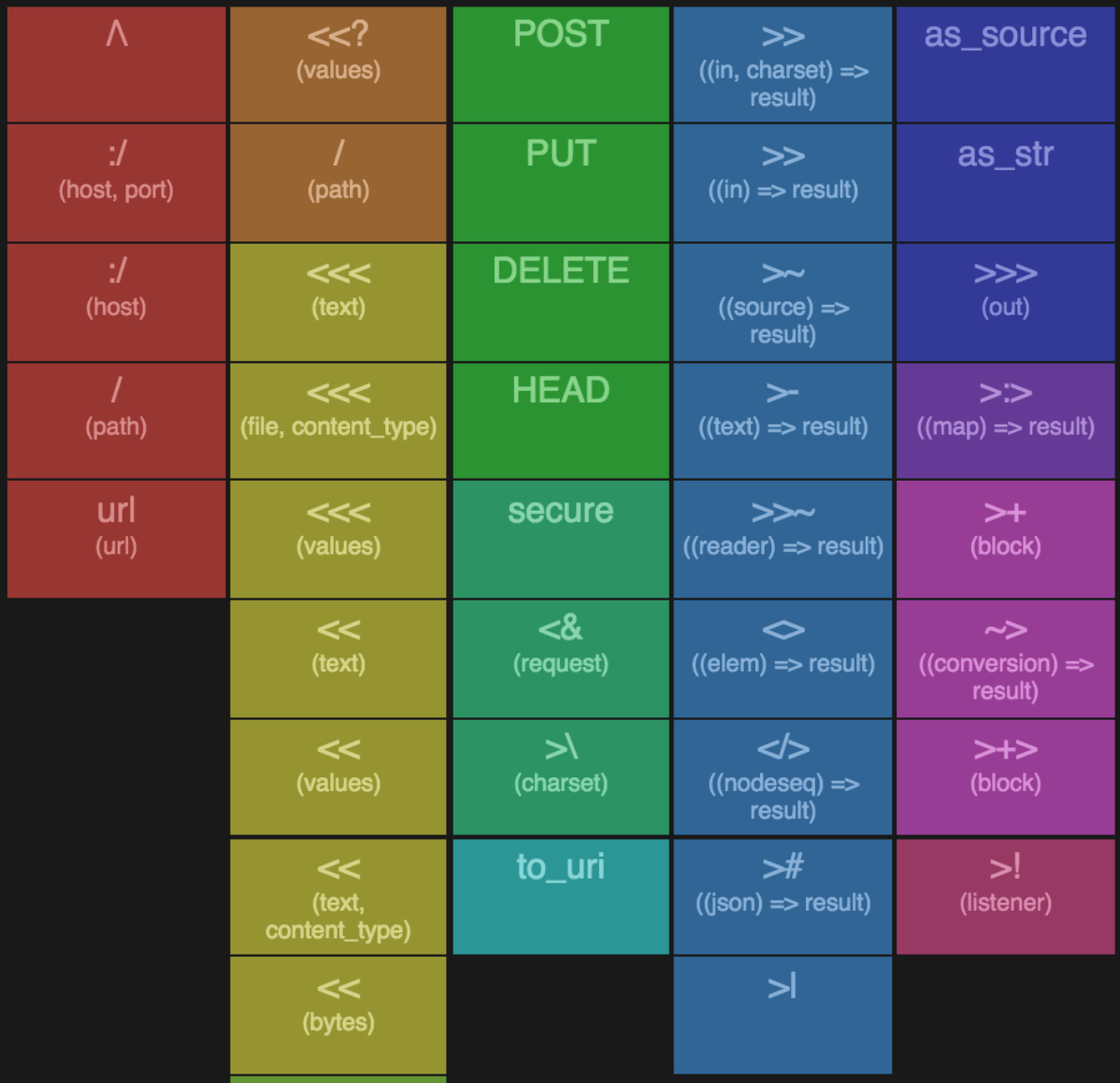
“Lisp is so powerful that problems which are technical issues in other programming languages are social issues in Lisp.”

(Rudolf Winestock)

Syntactic Flexibility

- What is the single thing people have complained most about Scala programs?

HTTP Dispatch Library



Syntactic Flexibility

- What is the single thing people have complained most about Scala programs?

  Symbolic Names!

Symbolic names are great for people who know a DSL inside out.

They are terrifying for everyone else.

The Story of SBT

- SBT
  - was: Simple Build Tool
  - now: Scala Build Tool (because people find it anything but simple).
- SBT 0.7: Essentially a Scala library to write programs that do builds. Direct mapping of all features
- SBT 10.x: Essentially a new language very cleverly embedded in Scala
  - Build definitions manipulate global maps of settings and tasks (in an imperative way!
  - Strange syntax
  - Hard to debug builds because of interpretation indirection.

SBT Example

```
lazy val scalatraSettings = Defaults.defaultSettings ++ ls.Plugin.lsSettings ++ Seq(
  organization := "org.scalatra",
  crossScalaVersions := Seq("2.10.0"),
  scalaVersion <<= (crossScalaVersions) { versions => versions.head },
  scalacOptions ++= Seq("-unchecked", "-deprecation", "-Yinline-warnings", "-Xcheckinit
  scalacOptions ++= Seq("-language:higherKinds", "-language:postfixOps", "-language:imp
  javacOptions ++= Seq("-target", "1.6", "-source", "1.6", "-Xlint:deprecation"),
  manifestSetting,
  resolvers ++= Seq(Opts.resolver.sonatypeSnapshots, Opts.resolver.sonatypeReleases),
  (LsKeys.tags in LsKeys.lsync) := Seq("web", "sinatra", "scalatra", "akka"),
  (LsKeys.docsUrl in LsKeys.lsync) := Some(new URL("http://www.scalatra.org/guides/"))
) ++ mavenCentralFrouFrou
```

This was version 0.11, later versions have simplified the syntax.

A Story about Macros

- Scala 2.10 got experimental macros
- Could invoke arbirary Scala code during compilation.
- The Play framework designers had a really clever idea:

A macro that would automatically validate a query against a database schema.

- When seeing a query, go to the database, get the schema, validate the query text against it.
- What could go wrong?

- In an IDE the typechecker is run on every keystroke.
- So the macro expansion also happens on every keystroke.

→ IDE slows to a crawl.

→ Consider Tooling for DSLs

Problems with Slick

- Slick is a database connectivity layer for Scala.
- Allows data to be expressed as Scala case classes that correspond to some database schema.
- Allows queries to be expressed in terms of for-expressions (which translate to map/flatmap/filter).

Challenges:

- Compilation times due to encodings of HLists and HMaps
- Error messages for type errors involving schemas.

Some Research Directions

- How can we balance expressiveness and uniformity?
- How can we restrict capabilities of DSLs?
- How can we ensure DSL tooling (e.g. error diagnostics, IDE experience, debugging) is as good as for the host language?

Contents of the Program

Platforms: What are good ways to define and implement DSLs?

Exploiting Domain Knowledge for

- Performance: How can we leverage domain-specific knowledge to get faster programs?
- New Targets: What are good techniques to translate DSL source code to non-standard targets?

Program

Mon	Tue	Wed	Thu	Fri
Declare your language Visser	DSL Embedding in Scala Rompf	DSL Embedding in Haskell Newton	Exploiting DS Knowledge: Spiral Püschel & Ofenbeck	Dynamic Compilation Wuerthinger
Quoted DSLs Wadler	DSL Embedding in Racket Flatt	Exploiting DS Knowledge: Databases and Data Analytics Koch	Hetero-geneous Computing Olukotun	Re-configurable Computing Bachrach