Polymorphic Embedding of DSLs

How does this fit into dOvs?

- EDSLs and compilers have a lot in common:
  - translation of higher-level entities into lower-level entities
  - checking well-formedness
  - optimizations
- Main difference: EDSLs are compositional by construction (which is both good and bad)
- This approach shows how to stick with the basic EDSL approach while making it almost as flexible as a compiler
- It can also be seen as a tutorial on how to use compiler techniques in libraries.
- You’ll learn some Scala on the way, too.
The Traditional Approach

- P. Hudak, Modular Domain Specific Languages and Tools
- DSL as library, not as a separate language
- DSL as an algebra, not via building ASTs
- Example: A Regions language

```plaintext
type Region = Vector ⇒ boolean

def univ : Region = p ⇒ true
def circle : Region
    = p ⇒ p._1 * p._1 + p._2 * p._2 < 1
def union(x : Region, y : Region) : Region
    = p ⇒ x(p) || y(p)
...
Pros and Cons

**Pros**
- **Reuse** of language infrastructure (incl. type checking)
- Interpretation is **compositional** (defined by an algebra)
- Allows combining several DSLs

**Cons**
- The interpretation is integral part of the language
  - Alternative interpretations cannot be supplied
- Interpretations are not **components**
  - In particular: Optimizations cannot be applied to them
Contributions

- Pure Embedding with **multiple** interpretations
  - Analyses and optimizations as “yet another” interpretation
- Interpretations and languages as **components**
- Scala as implementation language in OO context
  - Show-case for advanced OO features (partly experimental)
trait Regions {
  // Ordinary type synonyms
  type Vector = (double, double)

  // Abstract domain types
  type Region

  // Abstract domain operations
  def univ : Region
  def circle : Region
  def union(x : Region, y : Region) : Region
  def scale(v : Vector, r : Region) : Region
  ...
}

Explicit language interface
Explicit language interface II

- **Abstract type members** represent domain types

```python
type Region
```

- Compositional by construction:
  Interface is the **signature of the algebra**

```python
def union(x : Region, y : Region) : Region
```

**Architecture Overview**

Program Oblivious Client

Evaluator

Pretty Printer

Optimization

Regions

type Region

- type Region = Vec \( \Rightarrow \) Bool
- type Region = String

OptimizePrint
Architecture Overview

- **Evaluator**
  - `type Region = Vec ⇒ Bool`

- **Pretty Printer**
  - `type Region = String`

- **Program Oblivious Client**

- **Regions**
  - `type Region`

- **Optimization**
  - **OptimizePrint**
trait Evaluation extends Regions {
  type Region = Vector ⇒ boolean

  def univ : Region = p ⇒ true
  def circle : Region
    = p ⇒ p._1 * p._1 + p._2 * p._2 < 1
  def union(x : Region, y : Region) : Region
    = p ⇒ x(p) || y(p)

  ...
}

object Eval extends Evaluation

• Same definitions as in the traditional approach
A Pretty Printer

```
trait Printing extends Regions {
  type Region = String

  def univ : Region = "univ"
  def circle : Region = "circle"
  def union(x : Region, y : Region) : Region = "union(" + x + "," + y + ")"
  ...
}

object Print extends Printing
```
Programs as Oblivious Clients

// A simple program
def program(semantics : Regions)
    : semantics.Region = {
        import semantics._
        val ellipse24 = scale((2, 4), circle)
        union(univ, ellipse24)
    }

- A DSL program has path-dependent type: semantics.Region
- println(program(Eval)((1, 2))) prints true
- println(program(Print)) prints
  union(univ, scale((2, 4), circle))
Architecture Overview

Program
Oblivious Client

Regions

Evaluator

Pretty Printer

Optimization

OptimizePrint

type Region = Vec ⇒ Bool

type Region = String

17 / 31
A DSL with Polymorphism

- Example: Functions language (inspired by Carette et al.)
- User-defined bindings

```scala
trait Functions {
  // Abstract domain types
  type Rep[X]
  // Abstract domain operations
  def fun[S, T](f : Rep[S] ⇒ Rep[T]) : Rep[S ⇒ T]
  def app[S, T](f : Rep[S ⇒ T], v : Rep[S]) : Rep[T]
}
```

- Using higher-kindred abstract type member Rep
- Using higher-order abstract syntax (HOAS)
Two Example Interpretations

```scala
trait FunEval extends Functions {
  type Rep[T] = T
  def fun[S,T](f : S => T) = f
  def app[S,T](f : S => T, v : S) : T = f(v)
}
trait FunPrinting extends Functions {
  type Rep[X] = String
  def fun[S,T](f : String => String) : String = {
    val v = variables.next
    "fun(" + v + " => " + f(v) + ")"
  }
  ...
}
```
Architecture Overview

- Program
  - Oblivious Client
- Regions
  - type Region = Vec → Bool
- Evaluator
  - type Region = Vec → Bool
- Pretty Printer
  - type Region = String
- Optimization
  - OptimizePrint
Interpretations as Components

- Example: Optimization

```scala
trait Optimization extends Regions {
  val semantics : Regions

  type Region = (semantics.Region, boolean)
  def univ : Region = (semantics.univ, true)
  def circle : Region =
    (semantics.circle, false)
  def union(x : Region, y : Region) : Region =
    if (x._2 || y._2) (semantics.univ, true)
    else (semantics.union(x._1, y._1), false)
...
```
Architecture Overview

- Program
  Oblivious Client

- Regions
  type Region

- Evaluator
  type Region = Vec ⇒ Bool

- Pretty Printer
  type Region = String

- Optimization

- OptimizePrint
Reuse of Interpretations

- Interpretations can be regarded as **reusable components**
  - Odersky / Zenger: Scalable Component Abstractions
- Example: An **optimizing** interpretation can work on several interpretations

```scala
object OptimizePrint extends Optimization {
  val semantics = Print
}

println(program(OptimizePrint)) prints (univ, true)
while println(program(Print)) prints union(univ, scale((2, 4), circle))
Hierarchical Composition

- Example: A Vectors sublanguage

```scala
trait Vectors {
  type Vector
  ...
}

trait Regions {
  val vec : Vectors
  import vec._
  ...
}
```
Interplay with Interpretation Components

- Example: Optimization for refactored Regions language
- Needs `singleton types`

```
trait Optimization extends Regions {
  val semantics : Regions
  val vec : semantics.vec.type = semantics.vec
  import vec._
  ...
}
```
Peer Composition

- Example: Combine Regions with Functions language
- Problem: Representations have to be translated

```scala
trait FunReg extends Regions with Functions {
  implicit def fromRegion(r : Region) : Rep[Region]
  implicit def toRegion(r : Rep[Region]) : Region
}
```

- Using Scala’s `implicit` conversions for less verbosity
Peer Composition: Overview

**Regions**
- type Region

**Evaluator**
- type Region = Vec ⇒ Bool

**FunReg**
- type Region ↔ Rep[Region]

**FunRegEval**
- type Region = Rep[Region]

**Functions**
- type Rep[X]

**FunEval**
- type Rep[X] = X

**Motivation**

**Architecture**

**Summary**

**Core Elements**

**Interpretations as Components**

**Languages as Components**
Peer Composition of Interpretations

- Integration for both evaluation and printing semantics
- Example: Evaluation

```scala
object FunRegEval extends FunReg
  with Evaluation with FunEval {
    implicit def fromRegion(r : Region)
      : Rep[Region] = r
    implicit def toRegion(r : Rep[Region])
      : Region = r
  }
```
Summary

- Reuse of the language infrastructure in **pure embedding** style
- Interpretation components
  - In particular: Application of **optimizations** on them
- Language components

**Outlook**

- Compositionality can be limiting
- Regard Scala arithmetics, etc. as language interfaces
- Alternative approaches
  - Type classes (Haskell)
  - Virtual classes (gbeta)