Tagless final DSLs
(Metaprogramming 2018)

Jeremy Yallop
Embedded DSLs

Why might we embed a Domain Specific Language?

**Aim:** reuse host language facilities

- syntax (parsing, binding)
- type system (static semantics)
- evaluation (dynamic semantics)
- tooling (pragmatics)
Example: Feldspar

Write high-level Haskell code

\[ f\ i = (\text{testBit}\ i\ 0)\ ?\ (2\*i, i) \]

Generate idiomatic C code

```c
void test (struct array *mem, int32_t v0, int32_t *out) {
    if (testBit_fun_int32(v0, 0)) { (*out) = (v0 << 1); }
    else { (*out) = v0; }
}
```

The Design and Implementation of Feldspar: An Embedded Language for Digital Signal Processing
E. Axelsson et al.
IFL (2010)
Example: Hansei

Hansei is an embedded DSL for probability distributions

Example: compute $\Pr(\text{rain} | \text{grass is wet})$:

```
let_ (flip 0.3) (fun rain \to
let_ (flip 0.5) (fun sprinkler \to
let_ (dis (con (flip 0.9) rain)
(dis (con (flip 0.8) sprinkler)
   (flip 0.1))) (fun grass_is_wet \to
if_ grass_is_wet (fun () \to rain) (fun () \to dist [])))
```

---

*Embedded Probabilistic Programming*

O. Kiselyov and C. Shan

DSL (2009)
Example: QUEΛ

QUEΛ is an embedded DSL for database queries

Example: select all orders with id xoid:

```plaintext
database.query = 
foreach (fun () \rightarrow table_orders) @@ fun o \rightarrow 
  where ((o %. oid) \%= xoid) @@ fun () \rightarrow 
  yield o
```

Finally, safely-extensible and efficient language-integrated query
K. Suzuki et al.
PEPM (2016)
Example: Contracts

A verified embedded DSL for financial contracts (options, futures, &.)

Example: a contract is cancelled when an FX rate exceeds a threshold:

\[
\begin{align*}
\text{let } x &= \text{obs}(\text{FX}(\text{EUR}, \text{USD}), 0) \text{ in} \\
\text{if } \text{obs}(\text{FX}(\text{EUR}, \text{USD}), 0) &\geq 1.1 \times x \text{ within } 30 \\
&\quad \text{then empty} \\
&\quad \text{else c_option}
\end{align*}
\]

Certified symbolic management of financial multi-party contracts
P. Bahr et al.
ICFP (2015)
These DSLs use the **tagless final** style which enjoys some pleasant properties:

- typed
- compositional
- modular & extensible
- efficient
- declarative

---

*Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages*

J. Carette et al.

*Journal of Functional Programming (2009)*
Tagged interpreters

Algebraic data types offer a natural way to define DSLs:

```scala
sealed abstract trait Exp;
case class Lit(x: Int) extends Exp;
case class Add(x: Exp, y: Exp) extends Exp;

def eval(e : Exp) : Int = e match {
  case Lit(x) => x
  case Add(x, y) => eval(x) + eval(y)
}
```

However, this approach has a number of drawbacks:

- difficult to extend
- inefficient
- leads to complicated types
Folds

\[
\text{foldr} \ (+) \ 0 \ [x, y, z] \\
\sim \ (x + y + z + 0)
\]
Folds over syntax

```python
    exp match {
        case Lit(x) => lit(x)
        case Add(x,y) => add(fold(lit, add, x)) (fold(lit, add, y))
    }
}

def eval2(e: Exp) = fold(x => x,
         x => y => x + y,
         e)

Note: this becomes awkward when there are many constructors
```
trait Add[T] {
  def int : Int => T
  def add : T => T => T
}

object Eval extends Add[Int] {
  def int: Int => Int = x => x
  def add: Int => Int => Int = x => y => x + y
}

object Show extends Add[String] {
  def int: Int => String = x => x.toString
  def add: String => String => String = x => y => "("+x+"+"+y+")"
}
Tagless final ingredients

Constituents of a tagless final DSL:

1. a **signature** $S$ with
   - an abstract type $T$
   - abstract functions that build values of type $T$

2. **implementations** of $S$, such as
   - an evaluator
   - a pretty-printer
   - a compiler
   - an optimizer
   - . . .
Declarative DSLs

But to my mind there was never a clear definition of a “declarative language”, and hence no way to tell what is declarative and what is not.

https://existentialtype.wordpress.com/2013/07/18/

Working definition

Declarative means what, not how.

(best exemplified by multiple “how”s)
Extensible DSLs

The tagless style supports composition of independent components

```scala
trait Add[T[_]] {
  def int : Int => T[Int]
  def add : T[Int] => T[Int] => T[Int]
}

trait Bools[T[_]] {
  def bool : Boolean => T[Boolean]
  def if_[A] : T[Boolean] => (=> T[A]) => (=> T[A]) => T[A]
}

trait Exp[T[_]] extends Bools[T] with Add[T];
```

Unembedding Domain-Specific Languages
R. Atkey et al.
Haskell (2010)
Higher-order abstract syntax supports straightforward embedding of languages with binding

```scala
trait Lambda[T[_]] {
  def $(A, B) : T[A => B] => (T[A] => T[B])
}

// λf.λx.f x
lam (f =>
  lam (x =>
    f $( x))
```

Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism
G. Washburn and S. Weirich
Demo: ctypes

Example: treat C types as an embedded DSL:

```haskell
let puts = foreign "puts" (string @→ returning int)
puts "Hello, world!"
```

Multiple interpretations

- interpretation (dynamic binding)
- compilation (static binding)
- specialized error handling
- concurrency

A modular foreign function interface
J. Yallop et al.