A Generic Abstract Syntax Model for Embedded Languages

Emil Axelsson
Chalmers University of Technology

ICFP 2012, Copenhagen

Grand plan

Grand plan

Modular, reusable DSL implementations

Premise

`let` DSL = deeply embedded, compiled DSL

Background

Different DSLs often have a lot in common

- Similar constructs (e.g. conditionals, tuples, etc.)
- Similar interpretations/transformations (evaluation, constant folding, etc.)

Even within the same DSL there are opportunities for reuse

- E.g. many constructs introduce new variables

Background

Haskell is often said to be a good host for embedded DSLs, but. . .

Background

Haskell is often said to be a good host for embedded DSLs, but. . .

Making a realistic compiled DSL in Haskell is still hard work

- How to deal with variable binding?
- How to deal with sharing?
- Unpacking/packing of product types
- Etc.

These issues are

- nontrivial
- reimplemented over and over again

Problem

Lack of implementation reuse

- ASTs modeled as <u>closed</u> data types
- AST traversals not generic

This work

A generic data type model suitable for ASTs

- Direct support for generic traversals
- Easily combined with existing techniques for composing data types
- All inside Haskell

The AST model

```
data AST dom sig
  where
    Sym  :: dom sig → AST dom sig
    (:$) :: AST dom (a :→ sig) → AST dom (Full a) → AST dom sig

data Full a
data a :→ b
```

- Typed abstract syntax modeled as *application tree*
- Parameterized on *symbol domain* `dom`

Example: arithmetic expressions

Reference type

```
data Expr' a where
  Num' :: Int → Expr' Int
  Add' :: Expr' Int → Expr' Int → Expr' Int
  Mul' :: Expr' Int → Expr' Int → Expr' Int
```

Example: arithmetic expressions

Reference type

```
data Expr' a where
  Num' :: Int → Expr' Int
  Add' :: Expr' Int → Expr' Int → Expr' Int
  Mul' :: Expr' Int → Expr' Int → Expr' Int
```

AST encoding

```
data Arith a where
  Num :: Int → Arith (Full Int)
  Add :: Arith (Int :→ Int :→ Full Int)
  Mul :: Arith (Int :→ Int :→ Full Int)

type ASTF dom a = AST dom (Full a)
type Expr a     = ASTF Arith a
```

- Expr and Expr' isomorphic

Example: arithmetic expressions

Smart constructors

```
num      :: Int → Expr Int
add, mul :: Expr Int → Expr Int → Expr Int

num a   = Sym (Num a)
add a b = Sym Add :$ a :$ b
mul a b = Sym Mul :$ a :$ b
```

Example: arithmetic expressions

Smart constructors

```
num      :: Int → Expr Int
add, mul :: Expr Int → Expr Int → Expr Int

num a   = Sym (Num a)
add a b = Sym Add :$ a :$ b
mul a b = Sym Mul :$ a :$ b
```

$1 + 2 * 3$

```
ex₁' :: Expr' Int
ex₁' = Add' (Num' 1) (Mul' (Num' 2) (Num' 3))

ex₁  :: Expr  Int
ex₁  = add  (num  1) (mul  (num  2) (num  3))
```

Example: arithmetic expressions

Evaluation:

```
eval' :: Expr' a → a
eval' (Num' a)             = a
eval' (Add' a b)           = eval' a + eval' b
eval' (Mul' a b)           = eval' a ∗ eval' b

eval  :: Expr  a → a
eval  (Sym (Num a))        = a
eval  (Sym Add :$ a :$ b) = eval  a + eval  b
eval  (Sym Mul :$ a :$ b) = eval  a ∗ eval  b
```

- No loss of type-safety

Summary so far

- Recursive GADTs encoded as symbol types
- Small syntactic overhead
- No type safety lost

Summary so far

- Recursive GADTs encoded as symbol types
- Small syntactic overhead
- No type safety lost

What have we gained?

Key observation

Symbol types are non-recursive!

- AST can be traversed without matching on symbols (generic traversals)
- Symbol types can be composed (composable data types)

Generic traversal

Count the number of symbols in an expression

```
size :: AST dom a → Int
size (Sym _)   = 1
size (s :$ a) = size s + size a
```

- Independent of symbol domain

Generic traversal

Find the free variables in an expression

```
type VarId = Integer

freeVars :: Binding dom ⇒ AST dom a → Set VarId
freeVars (Sym (viewVar → Just v))          = singleton v
freeVars (Sym (viewBnd → Just v) :$ body) = delete v (freeVars body)
freeVars (Sym _)  = empty
freeVars (s :$ a) = freeVars s ‘union‘ freeVars a

class Binding dom
  where
    viewVar :: dom a          → Maybe VarId
    viewBnd :: dom (a :→ b) → Maybe VarId

    viewVar _ = Nothing
    viewBnd _ = Nothing
```

- Minimal assumptions of symbol domain
- Small encoding overhead
- Close to recursive traversal of ordinary data types

Composable data types

Direct sum of two symbol domains

```
data (dom₁ :+: dom₂) a
  where
    Inj_L :: dom₁ a → (dom₁ :+: dom₂) a
    Inj_R :: dom₂ a → (dom₁ :+: dom₂) a
```

Composable data types

Direct sum of two symbol domains

```
data (dom₁ :+: dom₂) a
  where
    Inj_L :: dom₁ a → (dom₁ :+: dom₂) a
    Inj_R :: dom₂ a → (dom₁ :+: dom₂) a
```

Increases overhead

```
type Expr a = ASTF (A :+: B :+: C :+: Arith :+: D) a

add :: Expr Int → Expr Int → Expr Int
add a b = Sym (Inj_R (Inj_R (Inj_R (Inj_L Add)))) :$ a :$ b
```

Composable data types

Solution: automating injections

```
num :: (Arith :<: dom) ⇒ Int → ASTF dom Int
add :: (Arith :<: dom) ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int
mul :: (Arith :<: dom) ⇒ ASTF dom Int → ASTF dom Int → ASTF dom Int

num a   = inj (Num a)
add a b = inj Add :$ a :$ b
mul a b = inj Mul :$ a :$ b
```

- `(:+:)`, `(:<:)` and `inj` borrowed from Data Types à la Carte [Swierstra, 2008]
- Also a projection function `prj` used for pattern matching

Extend `Arith` with variable binding

New constructs:

```
data Lambda a
  where
    Var :: VarId → Lambda (Full a)
    Lam :: VarId → Lambda (b :→ Full (a → b))

var :: (Lambda :<: dom) ⇒ VarId → ASTF dom a
var v = inj (Var v)

lam :: (Lambda :<: dom) ⇒ VarId → ASTF dom b → ASTF dom (a → b)
lam v a = inj (Lam v) :$ a
```

Extend `Arith` with variable binding

New constructs:

```
data Lambda a
  where
    Var :: VarId → Lambda (Full a)
    Lam :: VarId → Lambda (b :→ Full (a → b))

var :: (Lambda :<: dom) ⇒ VarId → ASTF dom a
var v = inj (Var v)

lam :: (Lambda :<: dom) ⇒ VarId → ASTF dom b → ASTF dom (a → b)
lam v a = inj (Lam v) :$ a
```

Example: $\lambda v_0 \rightarrow v_1 + (v_0 * v_2)$

```
ex₂ :: ASTF (Arith :+: Lambda) (Int → Int)
ex₂ = lam 0 $ add (var 1) (mul (var 0) (var 2))
```

Give meaning to the symbols

Explain which symbols are variables or binders

```
instance Binding Arith

instance (Binding dom₁, Binding dom₂) ⇒ Binding (dom₁ :+: dom₂)
  where
    viewVar (Inj_L s) = viewVar s
    viewVar (Inj_R s) = viewVar s
    viewBnd (Inj_L s) = viewBnd s
    viewBnd (Inj_R s) = viewBnd s

instance Binding Lambda
  where
    viewVar (Var v) = Just v
    viewVar _       = Nothing
    viewBnd (Lam v) = Just v
```

Generic traversal of composable AST

Example: λ`v`$_0$ $\rightarrow$ `v`$_1$ `+ (v`$_0$ `* v`$_2$`)`

```
ex₂ :: ASTF (Arith :+: Lambda) (Int → Int)
ex₂ = lam 0 $ add (var 1) (mul (var 0) (var 2))
```

```
*Main> freeVars ex₂
fromList [1,2]
```

The Syntactic library

AST model available in the Syntactic library:

```
cabal install syntactic
```

- Lots of utility functions
- Recursion schemes (`fold`, `everywhereTop`, etc.)
- A collection of common language constructs
- A collection of interpretations/transformations (evaluation, rendering, CSE, etc.)
- Utilities for host language interaction

Practical use: the Feldspar EDSL built upon Syntactic

Summary

AST model a good foundation for a general EDSL building library (Syntactic)

- Small encoding overhead
- Generic traversals out of the box
- Mixes well with sum types for compositional data types
- Traversals in familiar recursive style

Acknowledgements

This work was funded by

- Ericsson
- The Swedish Foundation for Strategic Research (SSF)
- Swedish Basic Research Agency (Vetenskapsrådet)