(Embedded) domain-specific languages

Haskell and Cryptocurrencies

Dr. Lars Brünjes, IOHK
Alejandro Garcia, IOHK
Dr. Andres Löh, Well-Typed LLP

2020-08-30

Goals

- Explain what (E)DSLs are.
- Look at various examples.
- Syntax vs. semantics.
- Shallow vs. deep embeddings.
- Expressive power.

Introduction: arithmetic expressions

Arithmetic expressions

Let's revisit an old example:

```
data Expr  -- abstract
lit    ::  Int -> Expr
(<+>) ::  Expr -> Expr -> Expr
```

Arithmetic expressions

Let's revisit an old example:

```
data Expr  -- abstract
lit    ::  Int -> Expr
(<+>) ::  Expr -> Expr -> Expr
```

Example program:

```
lit 3 <+> lit 5 <+> lit 0
```

Let's revisit an old example:

```
data Expr  -- abstract
lit   :: Int -> Expr
(<+>) :: Expr -> Expr -> Expr
```

Example program:

```
lit 3 <+> lit 5 <+> lit 0
```

What does the program above mean?

Syntax versus semantics

Syntax

Which strings (or abstract syntax trees) constitute a valid program?

Syntax versus semantics

Syntax

Which strings (or abstract syntax trees) constitute a valid program?

Semantics

Given a syntactically valid program, what meaning does it have?

Syntax versus semantics

Syntax

Which strings (or abstract syntax trees) constitute a valid program?

Semantics

Given a syntactically valid program, what meaning does it have?

We haven't yet provided implementations of `Expr`, `lit` and `(<+>)`.

Meaning of arithmetic expressions

Here is one option:

```
type Expr = Int

lit :: Int -> Expr
lit _ = 0

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2 + 1
```

Meaning of arithmetic expressions

Here is one option:

```
type Expr = Int

lit :: Int -> Expr
lit _ = 0

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2 + 1
```

Comments?

Meaning of arithmetic expressions

Here is one option:

```
type Expr = Int

lit :: Int -> Expr
lit _ = 0

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2 + 1
```

Comments?

Note that we implement expressions directly by their semantics!

Meaning of arithmetic expressions

Here is one option:

```haskell
type Expr = Int

lit :: Int -> Expr
lit _ = 0

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2 + 1
```

Comments?

Note that we implement expressions directly by their semantics!

The meaning of `e1 <+> e2` is computed from the meaning of `e1` and the meaning of `e2`!

Compositional semantics

Definition

If the semantics of a term can be computed (straight-forwardly) from the semantics of its subterms, the semantics is called compositional.

Compositional semantics

Definition

If the semantics of a term can be computed (straight-forwardly) from the semantics of its subterms, the semantics is called compositional.

Why is compositionality desirable?

Definition

If the semantics of a term can be computed (straight-forwardly) from the semantics of its subterms, the semantics is called compositional.

Why is compositionality desirable?

- Program fragments can be reasoned about in isolation.
- Easier to modify, extend or combine.

Multiple semantics

The semantics for arithmetic expressions we defined before was perhaps not the “expected” one:

- It computes the “cost” of an expression.
- Literals are free.
- Every adddition costs one.
- So in essence, we count the number of additions.

Multiple semantics

The semantics for arithmetic expressions we defined before was perhaps not the “expected” one:

- It computes the “cost” of an expression.
- Literals are free.
- Every adddition costs one.
- So in essence, we count the number of additions.

Many other semantics are possible:

- Evaluation to an integer.
- A textual representation.
- Simplification that removes additions of zero.

Evaluation of expressions

```haskell
type Expr = Int

lit :: Int -> Expr
lit i = i

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2
```

Textual representation

```haskell
type Expr = String

lit :: Int -> Expr
lit = show

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 ++ " + " ++ e2
```

Observation

- Implementing an expression directly by its semantics seems to require us to make a choice.
- Different semantics imply conflicting implementations of the components of our interface.
- The simplification semantics defined above turns an expression into another semantics. How would that work in our setting?

Observation

- Implementing an expression directly by its semantics seems to require us to make a choice.
- Different semantics imply conflicting implementations of the components of our interface.
- The simplification semantics defined above turns an expression into another semantics. How would that work in our setting?

Certainly,

```
type Expr = Expr
```

would lead to problems …

Combining semantics by tupling

Cost and value:

```
type Expr = (Int, Int)

lit :: Int -> Expr
lit i = (0, i)

(<+>) :: Expr -> Expr -> Expr
(c1, v1) <+> (c2, v2) = (c1 + c2 + 1, v1 + v2)
```

Combining semantics by tupling

Cost and value:

```haskell
type Expr = (Int, Int)

lit :: Int -> Expr
lit i = (0, i)

(<+>) :: Expr -> Expr -> Expr
(c1, v1) <+> (c2, v2) = (c1 + c2 + 1, v1 + v2)
```

Works, but:

- We have to define both semantics at the same time.
- In this case, because both have the same target type, it is quite easy to make mistakes.

Going via a datatype

```haskell
data Expr = Lit Int | Add Expr Expr

lit :: Int -> Expr
lit = Lit

(<+>) :: Expr -> Expr -> Expr
(<+>) = Add
```

Going via a datatype

```
data Expr = Lit Int | Add Expr Expr

lit :: Int -> Expr
lit = Lit

(<+>) :: Expr -> Expr -> Expr
(<+>) = Add
```

This can be seen as a particular choice of semantics as well (called initial semantics).

Going via a datatype

```
data Expr = Lit Int | Add Expr Expr

lit :: Int -> Expr
lit = Lit

(<+>) :: Expr -> Expr -> Expr
(<+>) = Add
```

This can be seen as a particular choice of semantics as well (called initial semantics).

This semantics yields the abstract syntax of the original expression, and we can perform interpretations of the `Expr` datatype as a second phase.

Cost semantics interpretation

```haskell
cost :: Expr -> Int
cost (Lit _)      = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
```

Cost semantics interpretation

```haskell
cost :: Expr -> Int
cost (Lit _)      = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
```

Applying the semantics now requires applying the `cost` function.

Cost semantics interpretation

```
cost :: Expr -> Int
cost (Lit _)     = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
```

Applying the semantics now requires applying the `cost` function.

This is still compositional:

- Semantics of an expression defined in terms of semantics of its components.

Cost semantics interpretation

```
cost :: Expr -> Int
cost (Lit _)     = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
```

Applying the semantics now requires applying the `cost` function.

This is still compositional:

- Semantics of an expression defined in terms of semantics of its components.
- In this for, semantics being compositional means in essence that the function is written using the standard design pattern for `Expr`!

Simplification semantics interpretation

```haskell
simplify :: Expr -> Expr
simplify (Lit i)      = Lit i
simplify (Add e1 e2) =
  case (simplify e1, simplify e2) of
    (Lit 0, e2') -> e2'
    (e1', Lit 0) -> e1'
    (e1', e2')   -> Add e1' e2'
```

Simplification semantics interpretation

```haskell
simplify :: Expr -> Expr
simplify (Lit i)      = Lit i
simplify (Add e1 e2) =
  case (simplify e1, simplify e2) of
    (Lit 0, e2') -> e2'
    (e1', Lit 0) -> e1'
    (e1', e2')   -> Add e1' e2'
```

Producing another expression as a result is now not a problem.

Abstracting from the compositional structure

Just as we have for other types, we can capture the standard design pattern:

```
data Expr = Lit Int | Add Expr Expr
```

Abstracting from the compositional structure

Just as we have for other types, we can capture the standard design pattern:

```
data Expr = Lit Int | Add Expr Expr
```

```
type ExprSem d = (Int -> d, d -> d -> d)
```

Abstracting from the compositional structure

Just as we have for other types, we can capture the standard design pattern:

```
data Expr = Lit Int | Add Expr Expr
```

```
type ExprSem d = (Int -> d, d -> d -> d)
```

```
foldExpr :: ExprSem d -> Expr -> d
foldExpr (lit_, add_) = go
  where
    go (Lit n)      = lit_ n
    go (Add e1 e2) = add_ (go e1) (go e2)
```

This sort of traversal is known as the fold or the catamorphism of the `Expr` datatype.

Using `foldExpr`

```
eval :: Expr -> Int
eval = foldExpr (id, (+))

cost :: Expr -> Int
cost = foldExpr (const 1, \ c1 c2 -> c1 + c2 + 1)

simplify :: Expr -> Expr
simplify = foldExpr (Lit, simplifyAdd)
  where
    simplifyAdd (Lit 0) e2 = e2
    simplifyAdd e1 (Lit 0) = e1
    simplifyAdd e1 e2      = Add e1 e2
```

Questions

What do the following interpretations do?

- `foldExpr (Lit, Add)`
- `foldExpr ((> 0), (&&))`

Intermediate summary

What we have seen so far:

- a simple language for arithmetic expressions,
- syntax and semantics,
- compositional evaluation semantics expressed directly,
- abstract syntax expressed via a datatype,
- several semantics as interpretation functions.

Domain-specific languages

Domain-specific languages (DSLs)

Definition

A language that is not necessarily general-purpose, but describes instances of problems in a particular domain is called a domain-specific language.

Domain-specific languages (DSLs)

Definition

A language that is not necessarily general-purpose, but describes instances of problems in a particular domain is called a domain-specific language.

The language of arithmetic expressions we have considered so far is a domain-specific language.

Domain-specific languages come in multiple flavours …

Standalone DSLs

Advantages:

- limited syntax makes programs easier to write, understand and maintain,
- programs can potentially be written by non-programmers.

Disadvantages:

- language has to be designed and implemented,
- lack of tool support (editors, debuggers, compilers, ...),
- difficult to add general-purpose features (module system, abstraction mechanisms, type system, ...).

Embedded domain-specific languages (EDSLs)

Embed a DSL as a library in a general-purpose host language (such as Haskell):

- we inherit useful features from the host language,
- we can reuse the tools available for the host language,
- knowing host language makes it easy to work with the DSL,
- multiple EDSLs can be combined and used together.

Embedded domain-specific languages (EDSLs)

Embed a DSL as a library in a general-purpose host language (such as Haskell):

- we inherit useful features from the host language,
- we can reuse the tools available for the host language,
- knowing host language makes it easy to work with the DSL,
- multiple EDSLs can be combined and used together.

Disadvantages:

- syntax and type system constrained by the host language,
- knowledge of host language is helpful or even required,
- error messages usually in terms of the general-purpose language.

Combining the two approaches

Not every (E)DSL is clearly one or the other:

- A frontend compiler intended for non-programmers that parses a custom and user-friendly syntax, and ideally reports understandable errors in terms of the application domain.
- The frontend translates source programs into an EDSL, i.e., into programs in a high-level general-purpose language for which a domain-specific library exists.

Shallow versus deep

There are also different degrees of embedding:

Shallow

EDSL constructs are directly represented by their semantics.

Deep

EDSL constructs are represented by their abstract syntax, and interpreted in a separate phase.

Shallow embeddings

Recall:

```haskell
type Expr = Int

lit :: Int -> Expr
lit i = i

(<+>) :: Expr -> Expr -> Expr
e1 <+> e2 = e1 + e2
```

- Shallow embeddings are very direct and often very efficient.
- Easy to add new language constructs.
- Difficult to add / change semantics.
- Difficult to (de)serialise.

Deep embeddings

Recall:

```
data Expr = Lit Int | Add Expr Expr

lit :: Int -> Expr
lit = Lit

(<+>) :: Expr -> Expr -> Expr
(<+>) = Add

eval :: Expr -> Int
eval (Lit i)     = i
eval (Add e1 e2) = eval e1 + eval e2
```

- Easy to define multiple semantics.
- Serialisation is just another interpretation function.
- More difficult to add new language constructs.

Other EDSLs

What we've seen in this course already:

- QuickCheck for testing (and for defining generators)
- STM for shared-memory concurrency
- Parser combinators
- Optics
- Streaming
- Stack programs (from exercises W3)
- ...

Other EDSLs

What we've seen in this course already:

- QuickCheck for testing (and for defining generators)
- STM for shared-memory concurrency
- Parser combinators
- Optics
- Streaming
- Stack programs (from exercises W3)
- …

Other classic examples:

- Pretty-printing
- HTML
- JSON
- SQL

Questions

- Can you think of more EDSLs?
- Can you think of a non-embedded DSL (perhaps even in the context of Haskell)?
- Are the EDLSs we have seen so far deep or shallow?

Extending arithmetic expressions to random expressions

A new language construct

Let's add a new construct:

```
rnd :: Int -> Int -> Expr
```

Parameters are lower and upper bound.

- Shallow embedding: provide an additional implementation.
- Deep embedding: change the datatype, possibly the fold, and all existing interpretation functions.

Extending a deep embedding

```haskell
data Expr = Lit Int | Add Expr Expr | Rnd Int Int

rnd = Rnd
```

```haskell
cost :: Expr -> Int
cost (Lit _)     = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
cost (Rnd _ _)   = 1
```

Extending a deep embedding

```haskell
data Expr = Lit Int | Add Expr Expr | Rnd Int Int
rnd = Rnd
```

```haskell
cost :: Expr -> Int
cost (Lit _)     = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
cost (Rnd _ _)   = 1
```

What about `eval`?

Extending a deep embedding

```
data Expr = Lit Int | Add Expr Expr | Rnd Int Int
rnd = Rnd
```

```
cost :: Expr -> Int
cost (Lit _)     = 0
cost (Add e1 e2) = cost e1 + cost e2 + 1
cost (Rnd _ _)   = 1
```

What about `eval`?

As discussed in the "Evolving an interpreter" case study, we switch to an applicative or monadic style.

Recall random numbers

```haskell
import System.Random
randomRIO :: Random a => (a, a) -> IO a
```

Adapting `eval`

```
eval :: Expr -> IO Int
eval (Lit i)     = pure i
eval (Add e1 e2) = pure (+) <*> eval e1 <*> eval e2
eval (Rnd l u)   = randomRIO (l, u)
```

Note that this is still compositional.

Exercises

- Define an interpretation of `Expr` that computes the lower and upper bound of evaluation.
- Adapt `foldExpr` to the new `Rnd` case.
- Rewrite `cost` and `eval` in terms of the new `foldExpr`.

Reusing host language constructs

We can define our own abstractions:

```
die :: Expr
die = rnd 1 6
```

```
dbl :: Expr -> Expr
dbl e = e <+> e
```

```
times :: Int -> Expr -> Expr
times n e
  | n > 0     = e <+> times (n - 1) e
  | otherwise = lit 0
```

Question

Is there a difference between the following two expressions?

```
die <+> die
```

```
let x = die in x <+> x
```

Question

Is there a difference between the following two expressions?

```
die <+> die
```

```
let x = die in x <+> x
```

No, we inherit Haskell semantics.

Another question

What is the cost of the following expression?

```
dbl (dbl (dbl (dbl (dbl (Lit 1)))))
```

Another question

What is the cost of the following expression?

```
dbl (dbl (dbl (dbl (dbl (Lit 1)))))
```

It is `31`, and it roughly doubles with every additional application of `dbl`.

Yet another question

Can we define an expression for which the evaluation loops?

Yet another question

Can we define an expression for which the evaluation loops?

Yes, trivially

```
loop :: Expr
loop = loop
```

Yet another question

Can we define an expression for which the evaluation loops?

Yes, trivially

```
loop :: Expr
loop = loop
```

However:

- Every finitely representable value of `Expr` terminates.
- Every expression with finite cost terminates.

Protecting ourselves from doing too much work

Given an “untrusted” expression in an EDSL, we can:

- first determine its cost before truly evaluating it,
- or bound the evaluator by some maximum cost.

A bounded evaluator

```
beval :: Expr -> BEval Int
```

What features do we need in the `BEval` type?

A bounded evaluator

```haskell
beval :: Expr -> BEval Int
```

What features do we need in the `BEval` type?

- state, to maintain the remaining budget for the cost of evaluation,
- failure, to abort if we run out of budget.

So one option is:

```haskell
type BEval = StateT Int Maybe
```

(There are other options to achieve the same result.)

Exercise

Implement the bounded evaluator.

Another example: propositional logic

Propositions

We focus on a deep embedding:

```
data Prop =
    Var String
  | T
  | F
  | Not Prop
  | And Prop Prop
  | Or Prop Prop
```

Exercises

Define the following interpretations:

- A cost function.
- An evaluator.
- A variable extractor.
- A tautology checker.
- A pretty-printer.
- A simplifier.

Discussion / outlook: time-changing values

Idea

We want to model values that change over time, let's say every day.

We should have:

- constants (unchanging values),
- variables (representing time-changing values),
- addition and multiplication,
- possibly conditions.

The big question here is how to represent time-changing values in the semantics.

Recap

We have discussed:

- a number of different EDSLs, in particular arithmetic expressions and propositions, and many Haskell libraries we have already covered in the course.
- shallow and deep embeddings.
- how to write interpretations on deep embeddings in a compositional style, possibly by using catamorphisms.
- that reusing host-language constructs in EDSL can have implications both for semantics and for performance.
- that one can use special-purpose interpretations to perform a form of static analysis.

What’s next?

We will look at more EDSLs, in particular moving towards Marlowe, an EDSL for expressing smart contracts that can run on a blockchain.

We will revisit the usefulness of having multiple interpretations for different purposes (cost measurement, safety, serialisation, simulation / debugging, execution).