(Embedded) Domain-Specific Languages

The Practice of Haskell Programming

Andres Löh

Well-Typed

May 18, 2012

What is an (E)DSL?

- DSL $=$ domain-specific language (fuzzy concept)
- EDSL $=$ **embedded** DSL

What is an (E)DSL?

- DSL $=$ domain-specific language (fuzzy concept)
- EDSL $=$ **embedded** DSL

In essence, EDSLs are just Haskell libraries:

- a limited set of types and functions;
- certain rules for composing sensible expressions out of these building blocks;
- often a certain unique look and feel;
- often understandable without having to know (all about) the host language.

DSLs vs. EDSLs

DSLs

- complete design freedom,
- limited syntax, thus easy to understand, usable by non-programmers,
- requires dedicated compiler, development tools,
- hard to extend with general-purpose features.

DSLs vs. EDSLs

DSLs

- complete design freedom,
- limited syntax, thus easy to understand, usable by non-programmers,
- requires dedicated compiler, development tools,
- hard to extend with general-purpose features.

EDSLs

- design tied to capabilities of host language,
- compiler and general-purpose features for free,
- complexity of host language available but exposed,
- several EDSLs can be combined and used together.

Haskell (or rather: Hackage) is full of EDSLs!

database queries
pretty-printing
workflows
parallelism
testing
web applications
(de)serialization
parsing
JavaScript
animations
hardware descriptions
data accessors / lenses
music
(attribute) grammars
HTML
concurrency
GUIs
array computations
images

Why?

Several reasons:

- syntactic freedom (user-defined operators and priorities, overloading, overloaded literals, `do`-notation, ...),
- higher-order functions,
- lazy evaluation,
- strong type system, type inference,
- algebraic datatypes,
- explicit effects,
- good partial evaluator,
- user-defined optimizations (rewrite rules).

EDSLs already encountered

QuickCheck:

- the language to construct **properties**,
- the language to construct **generators**.

In Ralf's lecture:

- an EDSL for describing **music**.

In Simon's lecture: EDSLs for

- computations with **software transactional memory**;
- describing **parallel computations**.

EDSL Example: Parser combinators

Parsing

Let us look at a classic EDSL example: **parsing**.

Goal:

- a library for describing parsers,
- no generation approach,
- high degree of abstraction,
- easy to use.

Parser interface

Type of parsers producing a result of type `a`:

```
data Parser a   -- abstract
```

Succeed always, consume nothing:

```
pure :: a → Parser a
```

Consume a single matching character:

```
satisfy :: (Char → Bool) → Parser Char
```

Parser interface – contd.

Change the type of the result:

$(\lt\$\gt) :: (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

Parse one thing followed by another:

$(\lt*\gt) :: \text{Parser } (a \rightarrow b) \rightarrow \text{Parser } a \rightarrow \text{Parser } b$

Parse one thing or another:

$(\lt|\gt) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a$

The type of $(<\!*\!>)$

Why not:

$$(<\!\times\!>) :: \text{Parser a} \rightarrow \text{Parser b} \rightarrow \text{Parser (a, b)}$$

The type of $(<*>)$

Why not:

$(<\times>) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a, b)$

Consider:

data $X = X\ A\ B\ C$
$pA :: \text{Parser } A$
$pB :: \text{Parser } B$
$pC :: \text{Parser } C$

The type of $(<*>)$

Why not:

$$(<\times>) :: \text{Parser a} \rightarrow \text{Parser b} \rightarrow \text{Parser (a, b)}$$

Consider:

$$\textbf{data}\ \text{X} = \text{X A B C}$$
$$\text{pA} :: \text{Parser A}$$
$$\text{pB} :: \text{Parser B}$$
$$\text{pC} :: \text{Parser C}$$

$$(\lambda((\text{a}, \text{b}), \text{c}) \rightarrow \text{X a b c}) <\$> (\text{pA} <\times> \text{pB} <\times> \text{pC}) :: \text{Parser X}$$

Well-Typed

The type of $(<*>)$

Why not:

$(<\times>) :: \text{Parser a} \to \text{Parser b} \to \text{Parser (a, b)}$

Consider:

data X = X A B C
pA :: Parser A
pB :: Parser B
pC :: Parser C

$(\lambda((a, b), c) \to \text{X a b c}) <\$> (\text{pA} <\times> \text{pB} <\times> \text{pC}) :: \text{Parser X}$

$\text{pure X} <*> \text{pA} <*> \text{pB} <*> \text{pC} :: \text{Parser X}$

An fmap function on parsers

The (<$>) operator has the same type as fmap:

```
(<$>) :: (a → b) → Parser a → Parser b
```

An fmap function on parsers

The `(<$>)` operator has the same type as `fmap`:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
(<$>) = fmap
```

An fmap function on parsers

The `(<$>)` operator has the same type as `fmap`:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
(<$>) = fmap
```

```
instance Functor Parser where
  fmap f p = pure f <*> p
```

An fmap function on parsers

The (<\$>) operator has the same type as fmap:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
(<$>) = fmap
```

```
instance Functor Parser where
  fmap f p = pure f <*> p
```

```
pure X <*> pA <*> pB <*> pC :: Parser X
```

An `fmap` function on parsers

The `(<$>)` operator has the same type as `fmap`:

```
(<$>) :: Functor f ⇒ (a → b) → f a → f b
(<$>) = fmap
```

```
instance Functor Parser where
  fmap f p = pure f <*> p
```

```
    X <$> pA <*> pB <*> pC :: Parser X
```

Derived parser combinators

Parsing many occurrences.

BNF:

$$\begin{aligned}\langle\mathrm{ps}\rangle &::= \langle\mathrm{p}\rangle\ \langle\mathrm{ps}\rangle \\ &\mid\ \ \varepsilon\end{aligned}$$

Derived parser combinators

Parsing many occurrences.

BNF:

$$\langle ps \rangle ::= \langle p \rangle\ \langle ps \rangle \mid \varepsilon$$

Haskell:

```
many :: Parser a → Parser [a]
many p  =  (:) <$> p <*> many p
           <|> pure []
```

Derived parser combinators

Parsing many occurrences.

BNF:

$$\langle\text{ps}\rangle ::= \langle\text{p}\rangle\ \langle\text{ps}\rangle \mid \varepsilon$$

Haskell:

```
many :: Parser a → Parser [a]
many p  =  (:) <$> p <*> many p
        <|> pure []
```

```
ident :: Parser String
ident = satisfy isAlpha <*> many (satisfy isAlphaNum)
```

Parsing balanced parentheses

BNF:

$$
\begin{array}{rcl}
\langle\text{bal}\rangle & ::= & \texttt{(}\ \langle\text{bal}\rangle\ \texttt{)}\ \langle\text{bal}\rangle \\
& | & \varepsilon
\end{array}
$$

Parsing balanced parentheses

BNF:

$$\begin{aligned}\langle \text{bal} \rangle &::= \texttt{(}\ \langle \text{bal} \rangle\ \texttt{)}\ \langle \text{bal} \rangle \\ &\mid\ \varepsilon\end{aligned}$$

Haskell:

```
sym :: Char → Parser Char
sym x = satisfy (== x)
bal :: Parser ...
bal   =   ...
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure ...
```

Parsing balanced parentheses

BNF:

$$\langle \text{bal} \rangle ::= \texttt{(}\ \langle \text{bal} \rangle\ \texttt{)}\ \langle \text{bal} \rangle$$
$$\quad\quad\quad \mid\ \varepsilon$$

Haskell:

```haskell
type Total = Int

sym :: Char -> Parser Char
sym x = satisfy (== x)

bal :: Parser Total
bal  =  (\_ m _ n -> (1 + m) + n)
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure 0
```

Well-Typed

Parsing balanced parentheses

BNF:

$$\begin{aligned}\langle \text{bal} \rangle &::= (\ \langle \text{bal} \rangle\ )\ \langle \text{bal} \rangle \\ &\mid\ \varepsilon\end{aligned}$$

Haskell:

```
data Bal = Nest [Bal]

sym :: Char → Parser Char
sym x = satisfy (== x)

bal :: Parser [Bal]
bal  =  (λ_ x _ xs → Nest x : xs)
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure []
```

Parsing balanced parentheses

BNF:

$$\begin{aligned}\langle \mathrm{bal} \rangle &::= (\ \langle \mathrm{bal} \rangle\ )\ \langle \mathrm{bal} \rangle \\ &\quad\mid\ \varepsilon\end{aligned}$$

Haskell:

```haskell
data Bal = Nest Bal Bal | Nil

sym :: Char → Parser Char
sym x = satisfy (== x)

bal :: Parser Bal
bal   =   (λ_ x _ xs → Nest x xs)
    <$> sym '(' <*> bal <*> sym ')' <*> bal
    <|> pure Nil
```

Implementation

List-of-successes semantics

A simple semantics of parsers:

List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,

$$\text{String} \rightarrow \quad \text{String}$$

List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,
- they also produce a result –

$$\text{String} \rightarrow \ (a, \text{String})$$

List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,
- they also produce a result –
- well, actually they can fail (no result) or be ambiguous (several results), too.

$$\text{String} \to [(\text{a}, \text{String})]$$

List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,
- they also produce a result –
- well, actually they can fail (no result) or be ambiguous (several results), too.

Let us try to just use this as an implementation:

$$\textbf{type}\ \mathsf{Parser}\ \mathsf{a} = \mathsf{String} \rightarrow [(\mathsf{a}, \mathsf{String})]$$

List-of-successes semantics

A simple semantics of parsers:

- parsers transform an input string,
- they consume some (but not necessarily all) input,
- they also produce a result –
- well, actually they can fail (no result) or be ambiguous (several results), too.

Let us try to just use this as an implementation:

```
type Parser a = String → [(a, String)]
```

Using functions as semantics is quite common.

Implementing simple parser combinators

```
pure :: a → Parser a
```

Implementing simple parser combinators

$\text{pure} :: a \to \text{String} \to [(a, \text{String})]$

Implementing simple parser combinators

$$\mathsf{pure} :: \mathsf{a} \to \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]$$
$$\mathsf{pure}\ \mathsf{x}\ \mathsf{ys} = [(\mathsf{x}, \mathsf{ys})]$$

Implementing simple parser combinators

```
pure :: a → String → [(a, String)]
pure x ys = [(x, ys)]
```

```
satisfy :: (Char → Bool) → Parser Char
satisfy x (y : ys) | x == y     = [(y, ys)]
                   | otherwise = [ ]
```

Implementing simple parser combinators

```
pure :: a → String → [(a, String)]
pure x ys = [(x, ys)]
```

```
satisfy :: (Char → Bool) → Parser Char
satisfy x (y : ys) | x == y      = [(y, ys)]
                   | otherwise = [ ]
```

```
(<$>) :: (a → b) → Parser a → Parser b
(f <$> p) xs = [(f r, ys) | (r, ys) ← p xs]
```

Implementing simple parser combinators

$$\mathsf{pure} :: \mathsf{a} \to \mathsf{String} \to [(\mathsf{a}, \mathsf{String})]$$
$$\mathsf{pure}\ \mathsf{x}\ \mathsf{ys} = [(\mathsf{x}, \mathsf{ys})]$$

$$\begin{aligned}
&\mathsf{satisfy} :: (\mathsf{Char} \to \mathsf{Bool}) \to \mathsf{Parser\ Char}\\
&\mathsf{satisfy}\ \mathsf{x}\ (\mathsf{y} : \mathsf{ys}) \mid \mathsf{x} == \mathsf{y} \qquad\quad = [(\mathsf{y}, \mathsf{ys})]\\
&\qquad\qquad\qquad\quad\ \mid \mathsf{otherwise} = [\,]
\end{aligned}$$

$$(<\$>) :: (\mathsf{a} \to \mathsf{b}) \to \mathsf{Parser\ a} \to \mathsf{Parser\ b}$$
$$(\mathsf{f} <\$> \mathsf{p})\ \mathsf{xs} = [(\mathsf{f\ r}, \mathsf{ys}) \mid (\mathsf{r}, \mathsf{ys}) \leftarrow \mathsf{p\ xs}]$$

$$\begin{aligned}
&(<*>) :: \mathsf{Parser}\ (\mathsf{a} \to \mathsf{b}) \to \mathsf{Parser\ a} \to \mathsf{Parser\ b}\\
&(\mathsf{p} <*> \mathsf{q})\ \mathsf{xs} = [(\mathsf{f\ r}, \mathsf{zs}) \mid (\mathsf{f}, \mathsf{ys}) \leftarrow \mathsf{p\ xs},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ (\mathsf{r}, \mathsf{zs}) \leftarrow \mathsf{q\ ys}]
\end{aligned}$$

Implementing simple parser combinators

```
pure :: a → String → [(a, String)]
pure x ys = [(x, ys)]
```

```
satisfy :: (Char → Bool) → Parser Char
satisfy x (y : ys) | x == y        = [(y, ys)]
                   | otherwise     = [ ]
```

```
(<$>) :: (a → b) → Parser a → Parser b
(f <$> p) xs = [(f r, ys) | (r, ys) ← p xs]
```

```
(<*>) :: Parser (a → b) → Parser a → Parser b
(p <*> q) xs = [(f r, zs) | (f, ys) ← p xs,
                            (r, zs) ← q ys]
```

```
(<|>) :: Parser a → Parser a → Parser a
(p <|> q) xs = p xs ++ q xs
```

Demo

(Demo.)

Disadvantages of our parsers

- We always compute all alternatives. Potentially lots of backtracking, inefficient.
- Absolutely no error messages.
- Tied to `String` as input type.

Disadvantages of our parsers

- We always compute all alternatives. Potentially lots of backtracking, inefficient.
- Absolutely no error messages.
- Tied to `String` as input type.

But these problems can be fixed. For example:

- `parsec`, limited lookahead, good error messages;
- `Text.ParserCombinators.ReadP`, trying several choices “in parallel”,
- `uu-parsinglib`, more sophisticated variant of `ReadP`, error messages, incremental parsing.

Common interfaces

Abstracting from interfaces

Many EDSLs are focused on one (or several) **parameterized** type(s):

```
Parser a
Gen a
STM a
Par a
```

Abstracting from interfaces

Many EDSLs are focused on one (or several) **parameterized** type(s):

```
Parser a
Gen a
STM a
Par a
```

- All these types can be seen as enriched, effectful versions of the underlying type `a`.
- We always need a way to relate plain types and their effectful versions.
- We always need a way to combine effectful terms.

Monads

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  fail   :: String → m a    -- controversial
```

Monads

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  fail   :: String → m a      -- controversial
```

Monads are common: they allow

- embedding via `return`,
- sequencing via `(>>=)`,
- later parts of the computation can depend on earlier results.

Monads

```haskell
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  fail   :: String → m a    -- controversial
```

Monads are common: they allow

- embedding via `return`,
- sequencing via `(>>=)`,
- later parts of the computation can depend on earlier results.

All of `Gen`, `STM` and `Par` are monads.

Advantages of a common interface

Next to the familiarity, we can define various functions in terms of just the interface and then reuse them on all types that implement the interface:

```
sequence :: Monad m ⇒ [m a] → m [a]
sequence []       = return []
sequence (m : ms) =
  m >>= λx → sequence ms >>= λxs → return (x : xs)
```

Advantages of a common interface

Next to the familiarity, we can define various functions in terms of just the interface and then reuse them on all types that implement the interface:

$$
\begin{array}{l}
\mathsf{sequence} :: \mathsf{Monad}\ \mathsf{m} \Rightarrow [\mathsf{m}\ \mathsf{a}] \rightarrow \mathsf{m}\ [\mathsf{a}] \\
\mathsf{sequence}\ [\,] \qquad\quad = \mathsf{return}\ [\,] \\
\mathsf{sequence}\ (\mathsf{m} : \mathsf{ms}) = \\
\quad \mathsf{m} \gg\!= \lambda \mathsf{x} \rightarrow \mathsf{sequence}\ \mathsf{ms} \gg\!= \lambda \mathsf{xs} \rightarrow \mathsf{return}\ (\mathsf{x} : \mathsf{xs})
\end{array}
$$

$$
\begin{array}{l}
\mathsf{foldM} :: \mathsf{Monad}\ \mathsf{m} \Rightarrow (\mathsf{a} \rightarrow \mathsf{b} \rightarrow \mathsf{m}\ \mathsf{a}) \rightarrow \mathsf{a} \rightarrow [\mathsf{b}] \rightarrow \mathsf{m}\ \mathsf{a} \\
\mathsf{foldM}\ \mathsf{op}\ \mathsf{e}\ [\,] \qquad\quad = \mathsf{return}\ \mathsf{e} \\
\mathsf{foldM}\ \mathsf{op}\ \mathsf{e}\ (\mathsf{x} : \mathsf{xs}) = \mathsf{foldM}\ \mathsf{op}\ \mathsf{e}\ \mathsf{xs} \gg\!= \lambda \mathsf{r} \rightarrow \mathsf{op}\ \mathsf{r}\ \mathsf{x}
\end{array}
$$

Advantages of a common interface

Next to the familiarity, we can define various functions in terms of just the interface and then reuse them on all types that implement the interface:

$$
\begin{aligned}
&\mathsf{sequence} :: \mathsf{Monad}\ m \Rightarrow [m\ a] \to m\ [a] \\
&\mathsf{sequence}\ [\,] \qquad\quad = \mathsf{return}\ [\,] \\
&\mathsf{sequence}\ (m : ms) = \\
&\quad m \ggg\!= \lambda x \to \mathsf{sequence}\ ms \ggg\!= \lambda xs \to \mathsf{return}\ (x : xs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{foldM} :: \mathsf{Monad}\ m \Rightarrow (a \to b \to m\ a) \to a \to [b] \to m\ a \\
&\mathsf{foldM}\ op\ e\ [\,] \qquad\ = \mathsf{return}\ e \\
&\mathsf{foldM}\ op\ e\ (x : xs) = \mathsf{foldM}\ op\ e\ xs \ggg\!= \lambda r \to op\ r\ x
\end{aligned}
$$

For the monad interface, we additionally get to use **do** notation.

Functors

Even more fundamental than the monad interface is the functor interface:

```haskell
class Functor f where
  fmap :: (a → b) → f a → f b

(<$>) = fmap
```

Functors

Even more fundamental than the monad interface is the functor interface:

```
class Functor f where
  fmap :: (a → b) → f a → f b
(<$>) = fmap
```

Every monad is a functor:

```
liftM :: Monad m ⇒ (a → b) → m a → m b
liftM f m = m >>= λx → return (f x)
```

However, this isn't automatically exploited in Haskell's class system.

Functors

Even more fundamental than the monad interface is the functor interface:

```
class Functor f where
  fmap :: (a → b) → f a → f b
(<$>) = fmap
```

Every monad is a functor:

```
liftM :: Monad m ⇒ (a → b) → m a → m b
liftM f m = m >>= λx → return (f x)
```

However, this isn't automatically exploited in Haskell's class system.

Types that are just functors are not very suitable for EDSLs, because there is no way to combine enriched terms . . .

Effectful application

But if we look at our parser interface, we find a different way of combining results:

$$(<\!*\!>) :: \text{Parser } (\text{a} \to \text{b}) \to \text{Parser a} \to \text{Parser b}$$

Effectful application

But if we look at our parser interface, we find a different way of combining results:

$$(<\!*\!>) :: \text{Parser } (a \to b) \to \text{Parser } a \to \text{Parser } b$$

- Another form of sequencing.
- While the results are combined, the second computation cannot depend on the result of the first.
- So $(<\!*\!>)$ is more restrictive than $(\gg\!=)$.
- The interface with embedding (pure , like return) and $(<\!*\!>)$ is called Applicative .

Applicative functors

```
class Functor f ⇒ Applicative f where
  pure  :: a → f a
  (<∗>) :: f (a → b) → f a → f b
```

Applicative functors

```
class Functor f ⇒ Applicative f where
  pure   :: a → f a
  (<∗>) :: f (a → b) → f a → f b
```

The `(<|>)` combinator is also more generally useful:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>)  :: f a → f a → f a
```

Question: How do we define `empty` for our parsers?

Common applicative functions

Again, there are functions that require just the `Applicative` (and `Alternative`) interfaces.

Recall for example `many`:

```
many :: Parser a → Parser [a]
many p  =  (:) <$> p <*> many p
           <|> pure []
```

Common applicative functions

Again, there are functions that require just the `Applicative` (and `Alternative`) interfaces.

Recall for example `many`:

```
many :: Alternative f ⇒ f a → f [a]
many p  =  (:) <$> p <*> many p
           <|> pure []
```

Monads are applicative

Many computations do not need the full power of `(>>=)`.

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mf mx = mf >>= \f -> mx >>= \x -> return (f x)
```

Once again, an `Applicative` instance isn't automatically defined for every monad – but most common monads have `Functor` and `Applicative` instances.

Stylistic comparison

All three are equivalent:

```
comp = do
  x ← f1
  y ← f2
  return (something x y)
```

```
comp = liftM2 something f1 f2
```

```
comp = something <$> f1 <*> f2
```

Degree of embedding

Degree of embedding

Shallow embedding
EDSL constructs are directly represented by their semantics.

Degree of embedding

Shallow embedding
EDSL constructs are directly represented by their semantics.

Deep embedding
EDSL constructs are represented by their abstract syntax, and interpreted in a separate stage.

Degree of embedding

Shallow embedding
EDSL constructs are directly represented by their semantics.

Deep embedding
EDSL constructs are represented by their abstract syntax, and interpreted in a separate stage.

Note:

- These are two extreme points in a spectrum.
- Most EDSLs use something in between (but close to one end).

Examples

Classic example of a shallow embedding:

$$\textbf{type}\ \text{Parser a} = \text{String} \to [(\text{a}, \text{String})]$$

Examples

Classic example of a shallow embedding:

```
type Parser a = String → [(a, String)]
```

The Haskore music language is using a (relatively) deep embedding:

```
data Music = Note Pitch Dur [NoteAttribute]
           | Rest Dur
           | Music :+: Music
           | Music :=: Music
           | Tempo (Ratio Int) Music
           | Trans Int Music
           | Instr IName Music
           | Player PName Music
           | Phrase [PhraseAttribute] Music
```

Shallow vs. deep

Shallow

- Working directly with the (denotational) semantics is often very concise and elegant.
- Relatively easy to use all Haskell features (sharing, recursion).
- Difficult to debug and/or analyze, because we are limited to a single interpretation.

Deep

- Full control over the AST, many different interpretations possible.
- Allows on-the-fly runtime optimization and conversion.
- We can visualize and debug the AST.
- Hard(er) to use Haskell's sharing and recursion.

An example

Let us embed an almost trivially simple language of arithmetic expressions:

```
data Expr   -- abstract

(⊕)  :: Expr → Expr → Expr
one  :: Expr

eval :: Expr → Int
```

An example

Shallow implementation

$$\textbf{type}\ \mathrm{Expr} = \mathrm{Int}$$

$$(\oplus) = (+)$$
$$\mathrm{one} = 1$$

$$\mathrm{eval} = \mathrm{id}$$

- We pick a semantics of expressions: an Int.
- We directly implement language constructs by their semantics.
- Very easy to do, but limited to a single interpretation.

An example

Deep implementation

$$(\oplus) :: \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}$$
$$\mathsf{one} :: \mathsf{Expr}$$
$$\mathsf{eval} :: \mathsf{Expr} \to \mathsf{Int}$$

An example

Deep implementation

$$
\begin{aligned}
&(\oplus) :: \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr} \\
&\mathsf{one} :: \mathsf{Expr} \\
&\mathsf{eval} :: \mathsf{Expr} \to \mathsf{Int}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{data}\ \mathsf{Expr} = \mathsf{Pl}\ \mathsf{Expr}\ \mathsf{Expr} \mid \mathsf{One} \\
&(\oplus) = \mathsf{Pl} \\
&\mathsf{one} = \mathsf{One} \\
&\mathsf{eval}\ (\mathsf{Pl}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2 \\
&\mathsf{eval}\ \mathsf{One} \qquad\quad\ \ = 1
\end{aligned}
$$

An example

Deep implementation

$$(\oplus) :: \mathsf{Expr} \to \mathsf{Expr} \to \mathsf{Expr}$$
$$\mathsf{one} :: \mathsf{Expr}$$
$$\mathsf{eval} :: \mathsf{Expr} \to \mathsf{Int}$$

$$\textbf{data}\ \mathsf{Expr} = \mathsf{Pl}\ \mathsf{Expr}\ \mathsf{Expr} \mid \mathsf{One}$$
$$(\oplus) = \mathsf{Pl}$$
$$\mathsf{one} = \mathsf{One}$$
$$\mathsf{eval}\ (\mathsf{Pl}\ e_1\ e_2) = \mathsf{eval}\ e_1 + \mathsf{eval}\ e_2$$
$$\mathsf{eval}\ \mathsf{One} = 1$$

We are no longer tied to one interpretation ...

Showing expressions

```
disp :: Expr → String
disp (Pl e1 e2) = "(" ++ disp e1 ++ " + " ++ disp e2 ++ ")"
disp One        = "1"
```

Showing expressions

```
disp :: Expr → String
disp (Pl e₁ e₂) = "(" ++ disp e₁ ++ " + " ++ disp e₂ ++ ")"
disp One        = "1"
```

Similarly, we could:

- transform the expression,
- optimize the expression,
- generate some code for the expression in another language,
- ...

Turning a concept into data

Moving from shallow towards deep is an important functional design pattern:

- introducing data types is easy,
- former functions become constructors,
- as a result, the structure of terms becomes observable.

Sharing and recursion

A user-defined abstraction

$$
\begin{aligned}
&\mathsf{tree} :: \mathsf{Int} \rightarrow \mathsf{Expr} \\
&\mathsf{tree}\ 0 = \mathsf{one} \\
&\mathsf{tree}\ \mathsf{n} = \mathbf{let}\ \mathsf{shared} = \mathsf{tree}\ (\mathsf{n} - 1)\ \mathbf{in}\ \mathsf{shared} \oplus \mathsf{shared}
\end{aligned}
$$

With the shallow embedding, this is fine:

- We reuse Haskell's sharing.
- What we share is just an integer.

But now in the deep setting . . .

```
tree :: Int → Expr
tree 0 = one
tree n = let shared = tree (n − 1) in shared ⊕ shared
```

But now in the deep setting ...

```
tree :: Int → Expr
tree 0 = one
tree n = let shared = tree (n − 1) in shared ⊕ shared
```

The call `disp (tree 3)` results in

```
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
```

Sharing is destroyed! We don't want to wait for `eval (tree 30)`!

Parsing our expression language

Our parser combinators work, but they cannot handle left-recursive grammars:

$$\begin{aligned} \mathrm{TM}\ \mathit{expr} ::=\ & \mathrm{NTOne} \\ \mid\ & \mathrm{NTOne}\ \mathrm{NTPlus}\ (\mathrm{TM}\ \mathit{expr})\mathrm{ok} \end{aligned}$$

Parsing our expression language

Our parser combinators work, but they cannot handle left-recursive grammars:

$$\begin{aligned} \text{TM } \mathit{expr} ::= {} & \text{NTOne} \\ \mid {} & (\text{TM } \mathit{expr}) \text{ NTPlus NTOnenot ok} \end{aligned}$$

Parsing our expression language

Our parser combinators work, but they cannot handle left-recursive grammars:

$$
\begin{aligned}
\mathrm{TM}\ \mathit{expr} ::= &\ \mathrm{NTOne} \\
\mid &\ (\mathrm{TM}\ \mathit{expr})\ \mathrm{NTPlus}\ \mathrm{NTOne} \text{not ok}
\end{aligned}
$$

Resulting parser:

$$
\begin{aligned}
&\mathit{expr} :: \mathrm{Parser\ Expr} \\
&\mathit{expr} = \mathrm{const}\ \mathrm{One} \quad <\$>\ \mathrm{sym}\ \texttt{'1'} \\
&\quad <|>\ (\lambda\_\ \_\ e \to \mathrm{Pl}\ \mathrm{One}\ e_2)\ <\$>\ \mathit{expr}\ <*>\ \mathrm{sym}\ \texttt{'+'}\ <*>\ \mathrm{sym}\ \texttt{'1'}
\end{aligned}
$$

- This parser will loop.
- In the second alternative, `expr` is called again before any input has been consumed.

Left recursion and parser combinators

For parsers, not being able to handle left recursion is not actually that serious a problem:

- left recursion can relatively easily be removed,
- common cases of left recursion can be abstracted into specific parser combinators (`chainl`).

Left recursion and parser combinators

For parsers, not being able to handle left recursion is not actually that serious a problem:

- left recursion can relatively easily be removed,
- common cases of left recursion can be abstracted into specific parser combinators (`chainl`).

Nevertheless, for some EDSL applications we would like to **preserve** or **observe** recursion and sharing.

Making sharing (and recursion) explicit

In practice, many EDSLs require preserving and observing sharing and recursion.

We need:

- a way to explicitly represent sharing in our representation,
- a way to conveniently produce terms in that representation.

Making sharing (and recursion) explicit

In practice, many EDSLs require preserving and observing sharing and recursion.

We need:

- a way to explicitly represent sharing in our representation,
- a way to conveniently produce terms in that representation.

Unfortunately, we have time left for only a short look at the options.

Observing sharing

Recall `vacuum`

The `vacuum` package:

- queried GHC's internal representation of data,
- in order to produce visualizations of terms that reveal the sharing.

Recall `vacuum`

The `vacuum` package:

- queried GHC's internal representation of data,
- in order to produce visualizations of terms that reveal the sharing.

Perhaps we can use a similar hack to recover implicit sharing in EDSL terms?

Introducing `data-reify`

The `data-reify` package offers such a function to recover implicit sharing:

$$\mathrm{reifyGraph} :: \mathrm{MuRef}\ s \Rightarrow s \rightarrow \mathrm{IO}\ (\mathrm{Graph}\ (\mathrm{DeRef}\ s))$$

Introducing `data-reify`

The `data-reify` package offers such a function to recover implicit sharing:

$\text{reifyGraph} :: \text{MuRef}\ s \Rightarrow s \to \text{IO}\ (\text{Graph}\ (\text{DeRef}\ s))$

Unfortunately, that looks a bit for complicated than `vacuum`'s:

$\text{view} :: a \to \text{IO}\ ()$

Question: Why?

Introducing `data-reify`

The `data-reify` package offers such a function to recover implicit sharing:

$\mathrm{reifyGraph} :: \mathrm{MuRef}\ s \Rightarrow s \rightarrow \mathrm{IO}\ (\mathrm{Graph}\ (\mathrm{DeRef}\ s))$

Unfortunately, that looks a bit for complicated than `vacuum`'s:

$\mathrm{view} :: a \rightarrow \mathrm{IO}\ ()$

Question: Why?

Because here, we need the results in a typed way.

Using `data-reify`

The `MuRef` class is about revealing the recursive structure of our type:

- we need the option to point at a **marker** rather than an actual value,
- so wherever we have a recursive subterm, we need flexibility.

Using `data-reify`

The `MuRef` class is about revealing the recursive structure of our type:

- we need the option to point at a **marker** rather than an actual value,
- so wherever we have a recursive subterm, we need flexibility.

Example:

```
data Expr    = Pl    Expr Expr | One
data ExprF e = PlusF e    e    | OneF
```

Now:

- the type `ExprF Int` is an expression with integers instead of subexpressions,
- the type `ExprF Expr` is isomorphic to the original `Expr` type.

Instantiating MuRef

```
instance MuRef Expr where
  type DeRef Expr = ExprF
  mapDeRef f One          = pure OneF
  mapDeRef f (Pl e1 e2)   = PlusF <$> f e1 <*> f e2
```

In mapDeRef, we have to explain how to turn an Expr into an ExprF u, for some applicative function f.

Instantiating `MuRef`

```
instance MuRef Expr where
  type DeRef Expr = ExprF
  mapDeRef f One          = pure OneF
  mapDeRef f (Pl e1 e2) = PlusF <$> f e1 <*> f e2
```

In `mapDeRef`, we have to explain how to turn an `Expr` into an `ExprF u`, for some applicative function `f`.

The type of `mapDeRef` is somewhat scary:

```
mapDeRef :: (Applicative f, MuRef a) =>
            (∀b.(MuRef b, DeRef a ~ DeRef b) => b -> f u) ->
            a -> f (DeRef a u)
```

Using reifyGraph

```
⟩ reifyGraph (tree 3)
let [(1, PlusF 2 2), (2, PlusF 3 3), (3, PlusF 4 4), (4, OneF)] in 1
```

Note that this is a pretty-printed version of this type:

```
data Graph e = Graph [(Unique, e Unique)] Unique
type Unique = Int
```

Working with explicitly shared structures

In practice, working with `Graph ExprF` rather than `Expr` can be awkward:

- looking up labels in a list,
- an extra indirection even for unshared subtrees,
- possibility to introduce duplicate or dangling labels.

Working with explicitly shared structures

In practice, working with `Graph ExprF` rather than `Expr` can be awkward:

- looking up labels in a list,
- an extra indirection even for unshared subtrees,
- possibility to introduce duplicate or dangling labels.

There are other options for handling names and binding, including:

- using string-based names,
- using De-Bruijn-indices,
- using (parametric) higher-order abstract syntax.

None of these options come entirely for free, but if you have to observe recursion and sharing, paying a certain price is unavoidable.

Summary

EDSLs:

- are ubiquitous in Haskell,
- often share monadic or applicative interfaces,
- can use shallow or deep embeddings.

It is not difficult to design your own EDSL.

Summary

EDSLs:

- are ubiquitous in Haskell,
- often share monadic or applicative interfaces,
- can use shallow or deep embeddings.

It is not difficult to design your own EDSL.

Features we have not covered in detail:

- adding new effect by changing the underlying monad or applicative functor,
- observing sharing and binding,
- expressing advanced invariants using the type system,
- optimizing by using GHC rewrite rules.