Haskell for (E)DSLs

Andres Löh

Well-Typed LLP

Functional Programming eXchange, 16 March 2012

My completely biased world view

Languages are everywhere!

My completely biased world view

Languages are everywhere!

- Nearly every (CS) concept is based on a language (even if you never see it).
- Nearly every tool is a compiler (translating one language into another).

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?

In this talk,

- we will look at a number of reasons why Haskell is particularly well-suited as a host language for EDSLs,
- and also at what might still be tricky.

Why?

In this talk,

- we will look at a number of reasons why Haskell is particularly well-suited as a host language for EDSLs,
- and also at what might still be tricky.

Many (**but not all**) points are valid for other FP languages as well.

Starting point: how do we design an EDSL?

More than one way . . .

Starting point: how do we design an EDSL?

More than one way ...

...but it makes sense to talk separately about:

- the (inter)face of the DSL,
- and its implementation.

What is an interface?

Syntax, essentially.

What is an interface?

Syntax, essentially.

In EDSL terms: **functions** and their **types**.

Syntactic flexibility is nice

It allows us to create a familiar look-and-feel.

Haskell features that help:

- user defined operators and priorities,
- overloading, in particular overloaded literals,
- `do`-notation,
- function calls without parentheses,
- quasi-quoting,
- …

Syntax samples

HaskellDB:

```
query =
  do cust ← table customers
     restrict (cust ! city .==. "London")
     project (cust ! customerID)
```

Parser combinators:

```
expr  =  Let <$ keyword "let" <*> decl <*
                keyword "in"  <*> expr
     <|> operatorExpr
```

See `haskelldb` and `uu-parsinglib` on HackageDB.

More syntax examples

(X)HTML:

```
htmlPage content =
  (header << ((thetitle << "An awesome page")
      +++ (script ! [thetype "text/javascript",
                     src "http://ajax.google..."] << "")
  ))
   +++ (body << content)
```

More syntax examples

(X)HTML:

```
htmlPage content =
  (header << ((thetitle << "An awesome page")
     +++ (script ! [thetype "text/javascript",
                     src "http://ajax.google..."] << "")
  ))
   +++ (body << content)
```

But let's not focus on syntax today . . .

See `html` and `xhtml` on HackageDB.

Have a look at `BASIC` on HackageDB by Lennart Augustsson and "Techniques for Embedding Postfix Languages in Haskell" by Chris Okasaki for some ideas of what you can do.

Types are important!

Normally, we're told that types

- **prevent errors**, or perhaps

Types are important!

Normally, we're told that types

- **prevent errors**, or perhaps
- **serve as checked documentation**.

Types are important!

Normally, we're told that types

- **prevent errors**, or perhaps
- **serve as checked documentation**.

In an EDSL (but also in other programs), they also

- **guide the programmer**.

A classic example: pretty-printing

Excerpts from the interface of `Text.Pretty`:

$$
\begin{array}{ll}
\mathsf{text} & :: \mathsf{String} \rightarrow \mathsf{Doc} \\
\mathsf{empty} & :: \mathsf{Doc} \\
(<>) & :: \mathsf{Doc} \rightarrow \mathsf{Doc} \rightarrow \mathsf{Doc} \\
\mathsf{sep} & :: [\mathsf{Doc}] \rightarrow \mathsf{Doc} \\
\mathsf{render} & :: \mathsf{Doc} \rightarrow \mathsf{String}
\end{array}
$$

Also see the `pretty` package on HackageDB.

Another example: parallel computations

Excerpts from the interface of `Control.Monad.Par`:

```
return :: a → Par a
new    :: Par (IVar a)
get    :: IVar a → Par a
put_   :: IVar a → a → Par ()

fork   :: Par () → Par ()
(>>=)  :: Par a → (a → Par b) → Par b

runPar :: Par a → a
```

Another example: parallel computations

Excerpts from the interface of `Control.Monad.Par`:

$$
\begin{array}{ll}
\mathsf{return} & :: \mathsf{a} \to \mathsf{Par}\ \mathsf{a} \\
\mathsf{new} & :: \mathsf{Par}\ (\mathsf{IVar}\ \mathsf{a}) \\
\mathsf{get} & :: \mathsf{IVar}\ \mathsf{a} \to \mathsf{Par}\ \mathsf{a} \\
\mathsf{put\_} & :: \mathsf{IVar}\ \mathsf{a} \to \mathsf{a} \to \mathsf{Par}\ () \\
\mathsf{fork} & :: \mathsf{Par}\ () \to \mathsf{Par}\ () \\
(\gg\!=) & :: \mathsf{Par}\ \mathsf{a} \to (\mathsf{a} \to \mathsf{Par}\ \mathsf{b}) \to \mathsf{Par}\ \mathsf{b} \\
\mathsf{runPar} & :: \mathsf{Par}\ \mathsf{a} \to \mathsf{a}
\end{array}
$$

Note:

- `IVar` and `Par` are library-specific parameterized types,
- $(\gg\!=)$ is a **higher-order** function.

Also see the `monad-par` package on HackageDB, by Ryan Newton and Simon Marlow.

Higher-order functions and Laziness

- Higher-order functions enable to define **glue** or **control** operators that allow us to combine code in various ways, so that the EDSL feels natural to use.
- Together with laziness, they make us quite independent of the evaluation behaviour of the embedded language (i.e., we can embed a strict language in Haskell).
- Laziness strengthens modularity and code reuse.

See Lennart Augustsson's blog post "More points for lazy evaluation" for a great summary.

User-defined parameterized types

The presence of user-defined parameterized types gives us:

- a way to describe different “kinds of computation” (e.g. `Par`)
- that are related to the underlying Haskell types (such as `Par a` is related to `a`),
- but we have **complete control** on how to construct, combine, or eliminate these computations.

Control over the types of terms we construct

Example:

$$
\begin{array}{ll}
\mathrm{mkInt} & :: \mathrm{Int} \to \mathrm{X}\ \mathrm{Int} \\
\mathrm{mkChar} & :: \mathrm{Char} \to \mathrm{X}\ \mathrm{Char} \\
\mathrm{combine} & :: \mathrm{X}\ a \to \mathrm{X}\ a \to \mathrm{X}\ a \\
\mathrm{pair} & :: \mathrm{X}\ a \to \mathrm{X}\ b \to \mathrm{X}\ (a, b) \\
\mathrm{run} & :: \mathrm{X}\ a \to a
\end{array}
$$

Control over the types of terms we construct

Example:

```
mkInt   :: Int → X Int
mkChar  :: Char → X Char
combine :: X a → X a → X a
pair    :: X a → X b → X (a, b)
run     :: X a → a
```

Note:

- We can never create an `X Bool`!
- Useful if the EDSL has more limited types than Haskell.
- Could for example be used to represent values that allow a particular compact representation.

See `adaptive-containers` and `repa` on HackageDB for examples of packages that use adaptive compact representations of “embedded” values.

Regions

Classic problem:

- allocate a reference/resource in one computation,
- pass it to an independent computation,
- access it there.

Regions

Classic problem:

- allocate a reference/resource in one computation,
- pass it to an independent computation,
- access it there.

Type system solution:

- computations and resources are parameterized by an (unknown) region,
- computations must not make assumptions about the region they ultimately run in,
- passing a reference to another computation assumes their regions are the same, so they can no longer be run independently.

Region example: state threads

From `Control.Monad.ST` and `Data.STRef` (base):

```
newSTRef  :: a → ST s (STRef s a)
readSTRef :: STRef s a → ST s a

runST     :: (∀s.ST s a) → a
```

Region example: state threads

From `Control.Monad.ST` and `Data.STRef` (`base`):

$$
\begin{array}{lll}
\mathsf{newSTRef} & :: & a \rightarrow \mathsf{ST}\ s\ (\mathsf{STRef}\ s\ a) \\
\mathsf{readSTRef} & :: & \mathsf{STRef}\ s\ a \rightarrow \mathsf{ST}\ s\ a \\
\mathsf{runST} & :: & (\forall s. \mathsf{ST}\ s\ a) \rightarrow a
\end{array}
$$

Note:

- in Haskell, we can make **no assumptions** about a **universally quantified** type;
- if we want run-time type analysis of some sort, we have to change the type, and lose the guarantees.

Also see “Lightweight monadic regions” by Oleg Kiselyov and Chung-chieh Shan for a more general approach, and `regions` on HackageDB by Bas van Dijk.

Effects

Haskell is (relatively) **pure**. We have control over **effects** that are explicit in the types.

Effects

Haskell is (relatively) **pure**. We have control over **effects** that are explicit in the types.

```
         a   -- some type, no effect
IO       a   -- IO, exceptions, random numbers, concurrency, ...
Gen      a   -- random numbers only
ST s     a   -- mutable variables only
STM      a   -- software transactional memory log variables only
State s  a   -- (persistent) state only
Error    a   -- exceptions only
Signal   a   -- time-changing value
...
```

Effects

Haskell is (relatively) **pure**. We have control over **effects** that are explicit in the types.

```
          a   -- some type, no effect
IO        a   -- IO, exceptions, random numbers, concurrency, ...
Gen       a   -- random numbers only
ST s      a   -- mutable variables only
STM       a   -- software transactional memory log variables only
State s   a   -- (persistent) state only
Error     a   -- exceptions only
Signal    a   -- time-changing value
...
```

A type gives us valuable information, from “no effects allowed” (e.g. `Int`) to “everything is allowed” (e.g. `IO Int`).

Effects example: software transactional memory

Software transactional memory is a lock-free approach to concurrency and shared data:

- groups of actions in a thread can be executed **atomically**,
- each such atomic transaction is run speculatively, creating a transaction log rather than mutating the shared state directly,
- at the end of the transaction, the system checks if the log is consistent with the memory and either commits the transaction or restarts it.

Effects example: software transactional memory

Software transactional memory is a lock-free approach to concurrency and shared data:

- groups of actions in a thread can be executed **atomically**,
- each such atomic transaction is run speculatively, creating a transaction log rather than mutating the shared state directly,
- at the end of the transaction, the system checks if the log is consistent with the memory and either commits the transaction or restarts it.

For this to work, it is **mandatory** that all effects in a transaction are effects that can be logged!

Software transactional memory in Haskell

From `Control.Concurrent.STM`:

```
atomically :: STM a → IO a

newTVar   :: a → STM (TVar a)
readTVar  :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
...
```

Software transactional memory in Haskell

From `Control.Concurrent.STM` :

```
atomically :: STM a → IO a

newTVar   :: a → STM (TVar a)
readTVar  :: TVar a → STM a
writeTVar :: TVar a → a → STM ()
...
```

Note:

- a **limited** set of effectful operations is available in `STM`,
- nothing else (e.g. random numbers, file IO) is possible,
- `STM` can be turned into `IO`, but not the other way round.

See the `stm` package on HackageDB.

Common interfaces

- Many EDSLs in Haskell work on some parameterized type `X a`.
- Many EDSLs have similar ways of combining such computations!

Common interfaces

- Many EDSLs in Haskell work on some parameterized type `X a`.
- Many EDSLs have similar ways of combining such computations!

Examples:

```
(<|>) :: X a → X a → X a               -- some form of choice
(≫)   :: X a → X b → X b               -- some form of sequence
(≫=)  :: X a → (a → X b) → X b         -- sequence using result
(<*>) :: X (a → b) → X a → X b         -- some form of application
...
```

Monads, applicative functors, . . .

In Haskell, we can abstract from the common interfaces and give them names:

- **monads** for computations that support sequencing where the rest of the computation can depend on previous results;
- **applicative functors** for computations that support effectful application;
- . . .

Monads, applicative functors, . . .

In Haskell, we can abstract from the common interfaces and give them names:

- **monads** for computations that support sequencing where the rest of the computation can depend on previous results;
- **applicative functors** for computations that support effectful application;
- . . .

EDSLs following established interfaces
are easier to learn and understand!

Algebraic properties and laws

Being explicit about effects encourages the design of EDSLs that allow us to reason about small programs locally:

- neutral elements,
- zero elements,
- associative operators,
- commutative operators,
- idempotent operators.

Algebraic properties and laws

Being explicit about effects encourages the design of EDSLs that allow us to reason about small programs locally:

- neutral elements,
- zero elements,
- associative operators,
- commutative operators,
- idempotent operators.

These properties can be stated, type-checked, and often automatically tested using yet another EDSL – QuickCheck.

Everything so far has been about the interface . . .

How do we implement the interface?

Everything so far has been about the interface . . .

How do we implement the interface?

Turns out there are several different approaches.

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).

Shallow embedding example: Lenses

From Data.Label.Abstract (`fclabels`):

```
data Point (⇝) f i o = Point {_get :: f ⇝ o,
                              _set :: (i, f) ⇝ f}
newtype Lens (⇝) f a = Lens {unLens :: Point (⇝) f a a}
```

Shallow embedding example: Parsec

From `Text.Parsec.Prim` (parsec):

```
newtype ParsecT s u m a = ParsecT
  {unParser ::
    ∀b.
    State s u →
    (a → State s u → ParseError → m b) →
    (                ParseError → m b) →
    (a → State s u → ParseError → m b) →
    (                ParseError → m b) →
    m b
  }
```

Deep embedding example: multisets

Multisets and operations on multisets:

```
data MSet a where
  MSet :: [a] → MSet a                 -- embedded list
  U    :: MSet a → MSet a → MSet a     -- union
  X    :: MSet a → MSet b → MSet (a, b) -- product
```

Deep embedding example: multisets

Multisets and operations on multisets:

```
data MSet a where
  MSet :: [a] → MSet a                      -- embedded list
  U    :: MSet a → MSet a → MSet a          -- union
  X    :: MSet a → MSet b → MSet (a, b)     -- product
```

```
list  :: MSet a → [a]
count :: MSet a → Int   -- efficient due to delayed products
```

See "Generic Multiset Programming with Discrimination-based Joins and Symbolic Cartesian Products" by Fritz Henglein and Ken Friis Larsen for more on this idea.

Deep embedding example: Accelerate

Excerpts from `Data.Array.Accelerate.AST` :

$$
\begin{array}{llll}
\textbf{data}\ \mathrm{PreOpenAcc}\ \mathrm{acc}\ \mathrm{env}\ \mathrm{a}\ \textbf{where} & & & \\
\quad \mathrm{PairArrays} :: (\ldots) \Rightarrow & & \mathrm{acc}\ \mathrm{env}\ (\mathrm{Array}\ \mathrm{sh}_1\ \mathrm{e}_1) \rightarrow & \\
 & & \mathrm{acc}\ \mathrm{env}\ (\mathrm{Array}\ \mathrm{sh}_2\ \mathrm{e}_2) \rightarrow & \\
 & \mathrm{PreOpenAcc} & \mathrm{acc}\ \mathrm{env}\ (\mathrm{Array}\ \mathrm{sh}_1\ \mathrm{e}_1, \mathrm{Array}\ \mathrm{sh}_2\ \mathrm{e}_2) & \\
\quad \mathrm{Acond} :: (\ldots) \Rightarrow & \mathrm{PreExp} & \mathrm{acc}\ \mathrm{env}\ \mathrm{Bool} \rightarrow & \\
 & & \mathrm{acc}\ \mathrm{env}\ \mathrm{arrs} \rightarrow & \\
 & \mathrm{PreOpenAcc} & \mathrm{acc}\ \mathrm{env}\ \mathrm{arrs} & \\
\quad \mathrm{Map} :: (\ldots) \Rightarrow & \mathrm{PreFun} & \mathrm{acc}\ \mathrm{env}\ (\mathrm{e} \rightarrow \mathrm{e}') \rightarrow & \\
 & & \mathrm{acc}\ \mathrm{env}\ (\mathrm{Array}\ \mathrm{sh}\ \mathrm{e}) \rightarrow & \\
 & \mathrm{PreOpenAcc} & \mathrm{acc}\ \mathrm{env}\ (\mathrm{Array}\ \mathrm{sh}\ \mathrm{e}') & \\
\quad \ldots & & &
\end{array}
$$

See `accelerate` on HackagedDB by Manuel Chakravarty et. al.

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.

So what's the problem with sharing?

Let us consider an extremely simple DSL:

$(\oplus)\ ::\ \mathsf{Expr} \rightarrow \mathsf{Expr} \rightarrow \mathsf{Expr}$
$\mathsf{one}\ ::\ \mathsf{Expr}$

$\mathsf{eval}\ ::\ \mathsf{Expr} \rightarrow \mathsf{Int}$

So what's the problem with sharing?

Let us consider an extremely simple DSL:

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

Shallow implementation:

```
type Expr = Int
(⊕) = (+)
one = 1
eval = id
```

A user-defined abstraction

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

With the shallow embedding, this is fine:

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

Now let us move to a deep embedding

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

Now let us move to a deep embedding

$(\oplus)$ :: Expr $\rightarrow$ Expr $\rightarrow$ Expr
one :: Expr
eval :: Expr $\rightarrow$ Int

data Expr = Pl Expr Expr | One

$(\oplus)$ = Pl
one = One

eval (Pl e_1 e_2) = eval e_1 + eval e_2
eval One = 1

Now let us move to a deep embedding

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

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

We are no longer tied to one interpretation ...

Showing expressions

$$\begin{aligned}
&\mathrm{disp} :: \mathrm{Expr} \rightarrow \mathrm{String}\\
&\mathrm{disp}\ (\mathrm{Pl}\ e_1\ e_2) = \texttt{"("} \mathbin{+\!\!+} \mathrm{disp}\ e_1 \mathbin{+\!\!+} \texttt{" + "} \mathbin{+\!\!+} \mathrm{disp}\ e_2 \mathbin{+\!\!+} \texttt{")"}\\
&\mathrm{disp}\ \mathrm{One} \qquad\quad\ \ = \texttt{"1"}
\end{aligned}$$

Showing expressions

$$
\begin{aligned}
&\mathsf{disp} :: \mathsf{Expr} \to \mathsf{String} \\
&\mathsf{disp}\ (\mathsf{Pl}\ e_1\ e_2) = \texttt{"("} +\!\!+ \mathsf{disp}\ e_1 +\!\!+ \texttt{" + "} +\!\!+ \mathsf{disp}\ e_2 +\!\!+ \texttt{")"} \\
&\mathsf{disp}\ \mathsf{One} \qquad\quad\; = \texttt{"1"}
\end{aligned}
$$

Similarly, we could:

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

But now reconsider ...

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

But now reconsider . . .

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

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) !

Solving the sharing problem

- We have to integrate sharing explicitly into our representation.
- This means we have to deal with variables and binding.
- There are several ways to do this.

Solving the sharing problem

- We have to integrate sharing explicitly into our representation.
- This means we have to deal with variables and binding.
- There are several ways to do this.

One particularly attractive approach to capturing sharing is **parametric higher-order abstract syntax** (PHOAS).

For more information on PHOAS and sharing see “Parametric Higher Order Abstract Syntax for Mechanized Semantics” by Adam Chlipala, and “Functional Programming with Structured Graphs” by Bruno Oliveira and William Cook.

Extending the expression datatype

```
data Expr a = Pl (Expr a) (Expr a) | One
            | Var a | Let (Expr a) (a → Expr a)
```

Extending the expression datatype

$$\begin{aligned} &\textbf{data}\ \text{Expr a} = \text{Pl (Expr a) (Expr a)} \mid \text{One} \\ &\qquad \mid\ \text{Var a} \mid \text{Let (Expr a) (a} \rightarrow \text{Expr a)} \end{aligned}$$

Note:

- two new constructors, for variables and for **let**,
- a Let takes a **shared expression** and a **function**.

Extending the expression datatype

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Expr}\ a = \mathsf{Pl}\ (\mathsf{Expr}\ a)\ (\mathsf{Expr}\ a) \mid \mathsf{One}\\
&\qquad\qquad\qquad \mid \mathsf{Var}\ a \mid \mathsf{Let}\ (\mathsf{Expr}\ a)\ (a \to \mathsf{Expr}\ a)
\end{aligned}
$$

Note:

- two new constructors, for variables and for **let**,
- a Let takes a **shared expression** and a **function**.

Similar to regions, a proper expression should not assume anything about its variables:

$$
\begin{aligned}
&\mathsf{tree} :: \mathsf{Int} \to \mathsf{Expr}\ a\\
&\mathsf{tree}\ 0 = \mathsf{one}\\
&\mathsf{tree}\ n = \mathsf{Let}\ (\mathsf{tree}\ (n - 1))\ (\lambda \mathsf{shared} \to \mathsf{Var}\ \mathsf{shared} \oplus \mathsf{Var}\ \mathsf{shared})
\end{aligned}
$$

Extending the evaluator

We have to add (trivial) cases for `Let` and `Var` to `eval`:

```
eval :: Expr Int → Int
eval (Var x)   = x
eval (Let e f) = eval (f (eval e))
```

Here, `e` is shared.

Redefining the printer

```haskell
disp :: Expr String → Int → String
disp (Pl e1 e2) c = "(" ++ disp e1 c ++ " + " ++ disp e2 c ++ ")"
disp One        c = "1"
disp (Var x)    c = x
disp (Let e f)  c = let v = "x" ++ show c
                    in "let " ++ v ++ " = " ++ disp e (c + 1) ++
                       " in " ++ disp (f v) (c + 1)
```

Note:

- Sharing really is observable now.
- We decide what to do with shared expressions.

Can we still use Haskell `let`?

Yes, but it requires a "hack":

- We can write a function that observes the internal sharing of Haskell.
- This is a side effect, so the result type is tagged (in `IO`).
- But we can then convert the observed sharing into a datatype with explicit `Let` and work with that in a robust way.

See `data-reify` on HackageDB by Andy Gill.

Conclusions

- Think of your (business, research, hobby, ...) problems from a language viewpoint.
- Haskell is a great language for implementing (E)DSLs.
- Types help you in various amazing ways.

Conclusions

- Think of your (business, research, hobby, . . .) problems from a language viewpoint.
- Haskell is a great language for implementing (E)DSLs.
- Types help you in various amazing ways.
- Consider designing your own EDSLs!
- Domains for EDSLs are everywhere . . .
- . . . and Haskell makes it (relatively easy) . . .
- . . . and lets you focus on the important things . . .
- . . . such as a clear and easy-to-understand interface.

Conclusions

- Think of your (business, research, hobby, ...) problems from a language viewpoint.
- Haskell is a great language for implementing (E)DSLs.
- Types help you in various amazing ways.
- Consider designing your own EDSLs!
- Domains for EDSLs are everywhere ...
- ... and Haskell makes it (relatively easy) ...
- ... and lets you focus on the important things ...
- ... such as a clear and easy-to-understand interface.
- You probably want a deep embedding,
- as turning things into **data** gives you a lot of control,
- but be careful with sharing.
- Learn from the many existing examples!

Thank you!

Questions?

andres@well-typed.com