

# Functional Programming

---

SS 2025

Torsten Grust  
University of Tübingen

## Domain-Specific Languages (DSLs)

---

- DSLs are “small” languages designed to easily and directly express the concepts/idioms of a specific domain. *Not* Turing complete in general.
- Examples:

Domain	DSL
OS automation	Shell scripts
Typesetting	(La)TeX
Queries	SQL
Game Scripting	UnrealScript, Lua
Parsing	Bison, ANTLR

## Two Main Flavors of DSLs

---

- **Standalone DSL:** separate parser, compiler, and runtime.
  - However: many DSLs are PL-like and feature variables, definitions (macros), conditionals, ... This leads to:
- **Embedded DSL:** retain given host PL syntax (DSL raises level of abstraction), reuse parser/compiler/runtime.
  - Familiarity and syntactic conventions carry over, less implementation effort. DSL comes in form of family of functions/operators (library) and possibly higher-order functions to represent new control flow constructs.

## Embedded DSL in Functional Programming Languages

---

- Functional languages make for good hosts for **embedded DSLs**:
  - algebraic data types (e.g., to model ASTs)
  - higher-order functions (abstraction, control constructs)
  - lightweight syntax (layout/whitespace, non-alphabetic identifiers, juxtaposition for application)

Examples (program syntax matches notation used in the domain):

1. In Haskell, we can define infix binary operator `==>` to denote Boolean implication  $\Rightarrow$ .
2. We can use Unicode symbols like `∪` to denote set union, ...

## DSL Design Space: Library

---

Example (an embedded DSL for finite sets of integers):

```
type IntegerSet = ...
```

```
empty  :: IntegerSet
insert :: Integer -> IntegerSet -> IntegerSet
delete :: Integer -> IntegerSet -> IntegerSet
member :: Integer -> IntegerSet -> Bool
```

} constructors  
observer

```
member 3 (insert 1 (delete 3 (insert 2 (insert 3 empty)))) → False
```


DSL programs are compositions of constructor and observer applications. Haskell syntax of composition, applications, and literal elements reused.

## DSL Design Space: Library

---

- **DSL implementation option ①:** Representation of integer set fully exposed:

```
type IntegerSet = [Integer]  -- unsorted, duplicates allowed
```

- Introduction of new operators is straightforward. Can adopt domain-specific notation (e.g.,  $\in$ ,  $\subseteq$ ) if desired.
-  **But:** Any such extension of the “library” is based on the current exposed implementation. A later change of representation is impossible/requires reimplementation (if possible) of the extensions.

## Interlude: Haskell Modules

---

- Group related definitions (values, types) in single file `M.hs`:

```
module M where

type Predicate a = a -> Bool

id :: a -> a
id = \x -> x
```

- Module hierarchy: module `A.B.C.M` lives in file `A/B/C/M.hs`.
- Explicit export lists hide all other definitions:

```
module M (id) where      -- only the definition of id is exported,
  :                      -- type Predicate a not exported
```

- Access definitions in other module `M`:

```
import M
```

## Modules and Abstract Data Types

---

- **Abstract data types:** export algebraic data type, but *not* its constructor functions:

```
module M (Rose, leaf) where    -- constructor Node not exported

data Rose a = Node a [Rose a]

leaf :: a -> Rose a
leaf x = Node x []
```

- If you must, explicitly export the constructors:

```
module M (Rose(Node), leaf) where    -- export constructor Node
.....
module M (Rose(..), leaf) where    -- export all constructors
```

- Instance def.s and `deriving` are exported with their type.



## Importing Modules

---

- Qualified import to partition Haskell's name space:

```
import qualified M
```

```
t :: M.Rose Char  
t = M.leaf 'x'
```

- Partially import module (required definitions only):

```
import Data.List (nub, reverse)  
⋮
```

---

```
import Data.List hiding (reverse)    -- everything but reverse  
⋮
```

## DSL Design Space: Library

---

[Back from the module interlude.]

- **DSL implementation option ②:** integer set representation realized as an abstract data type.
  - Inside module `SetLanguage`, implement one of many possible integer set representations, *e.g.*:
    1. Unordered lists (implementation type `[a]`).
    2. Characteristic functions (implementation type `a -> Bool`).
  - Do *not* expose these implementation details. The clients of module `SetLanguage` can not peek inside and will not be able to tell the difference.

## Shallow vs. Deep DSL Embeddings

---

Recall that our integer set DSL featured two categories of operations:

empty	}	..... <b>constructors</b>	[construct integer sets]
insert			
delete	}	..... <b>observers</b>	[observe elements in integer sets]
member			
card			

DSLs offer two principal design choices to implement the semantics of these operations:

1. Constructors do all the hard work (→ **shallow embedding**).
2. Constructors are trivial, instead observers perform actual work (→ **deep embedding**).

## Shallow DSL Embedding

---

- In a **shallow DSL embedding**, the semantics of DSL operations are directly expressed in terms of host language values (*e.g.*, lists or characteristic functions).
- For the integer set DSL:
  - Constructors `empty`, `insert`, `delete` will perform actual work, *i.e.*, actually compute these values. Harder to add.
  - Observers `member` and `card` will be trivial and merely inspect these values. Trivial to add.

## Deep DSL Embedding

---

- In a **deep DSL embedding**, the DSL operations build an *abstract syntax tree* (AST) that represents operation applications and arguments:
  - Constructors merely build the AST and are very easy to add.
  - Observers interpret (traverse) the AST and thus perform the actual work.

## Using Type Classes to Generate ASTs for Deep DSL Embeddings

---

- Consider a deep DSL embedding for a simple language of arithmetic expressions and its simple `eval` observer.
- Construction of expressions requires the use of constructors. The notation of nested expressions quickly becomes tedious:

File: `expr-deep-num.hs`

```
import ExprDeepNum

-- e1 = 8 * 7 - 14
e1 :: Expr
e1 = Sub (Mul (Val 8) (Val 7)) (Val 14)

main :: IO ()
main = print $ eval e1
```

## Using Type Classes to Generate ASTs for Deep DSL Embeddings

---

- Type `Expr` represents simple arithmetic expressions (over integers). Exactly what is described by type class `Num`:

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a           -- default: negate x ≡ 0 - x
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
```

💡 **Idea:** Make `Expr` an instance of `Num`. Don't let operator `+` perform actual arithmetic, but construct an AST (`Add ... ..`) instead.

## Generalized Algebraic Data Types (GADTs)

---

- Algebraic data types are instrumental in making the deep embedding approach feasible (lightweight construction of ASTs, pattern matching to traverse/interpret ASTs, ...).
- Now consider another example:
  - A deeply embedded expression language *over integers and Booleans*.
  - Evaluation of such an expression via observer `eval` yields a result of type `Either Integer Bool`.



## Generalized Algebraic Data Types (GADTs)

---

- **Problem:** our current deep embedding is *untyped* (or rather: *untyped*): *all* constructors simply yield an AST of type `Expr` regardless of actual expression value.
- Let us make this problem apparent by using a variant of Haskell's syntax when we declare the algebraic data type `Expr`.
  - We will need the **Haskell language extension** `GADTs`. Enable via GHC compiler pragma:

```
{-# LANGUAGE GADTs #-}
```

## Generalized Algebraic Data Types (GADTs)

---

### 💡 Idea:

1. Encode the type of a DSL expression (here: `Integer` or `Bool`) in its *Haskell type*.
  - In a nutshell, let us have ASTs of types `Expr Integer` and `Expr Bool` (not just `Expr`).
2. Use Haskell's type checker to ensure at compile time that only well-typed DSL expressions can be built.

## Generalized Algebraic Data Types (GADTs)

---

- Haskell language extension: `{-# LANGUAGE GADTs #-}`
- Define entirely new parameterized type  $T$ , its constructors  $K_i$ , and their type signatures:

```
data  $T$   $a_1$   $a_2$  ...  $a_n$  where
```

```
   $K_1$  ::  $b_{11}$  -> ... ->  $b_{1(n1)}$  ->  $T$   $t_{11}$   $t_{12}$  ...  $t_{1n}$ 
```

```
   $K_2$  ::  $b_{21}$  -> ... ->  $b_{2(n2)}$  ->  $T$   $t_{21}$   $t_{22}$  ...  $t_{2n}$ 
```

```
    ⋮
```

```
   $K_r$  ::  $b_{r1}$  -> ... ->  $b_{r(nr)}$  ->  $T$   $t_{r1}$   $t_{r2}$  ...  $t_{rn}$ 
```

```
  [deriving  $C_1, C_2, \dots$ ]
```

the  $t_{ij}$  may vary from  
constructor to constructor

## Type Class Example: One DSL, Multiple Embeddings

---

- **Example:** Define an expression language over integers that supports variable binding (e.g., `let x = e1 in e2`).
- We want to try out **multiple representation types** `a` for the language. Define type class `Expr a` for which we can define multiple instances:
  1. Shallow embedding #1: Represent expressions as Haskell functions `Env -> Integer` that map a given environment (`{x ↦ 42}`) of variable bindings to the expression's value.
  2. Shallow embedding #2: Derive a `String` form of the expression.
  3. Deep embedding: Build a simple abstract syntax tree (`AST Integer`) that represents the expression (this opens up the opportunity to simplify expressions, for example).

## Example: Shallow Embedding of a Pattern Matching DSL

---

- **Example:** Define a shallowly embedded DSL for **string pattern matching**.
- Follow an idea in Phil Wadler's seminal 1985 paper *“How to Replace Failure by a List of Successes”*:
  1. Given an input string, a pattern returns the **list of matches**. If matching fails, return the **empty list**.
  2. One match consists of
    - a result of some type **a** (e.g., the matched characters, constructed token or parse tree) and
    - the residual input string left to match.

```
-- Match against a string, return result of type a  
type Pattern a = String -> [(a, String)]
```

## Example: Shallow Embedding of a Pattern Matching DSL

---

Pattern	DSL function
match literal char	<code>lit :: Char -&gt; Pattern Char</code>
match empty string	<code>empty :: a -&gt; Pattern a</code>
fail always	<code>fail :: Pattern a</code>
alternative	<code>alt :: Pattern a -&gt; Pattern a -&gt; Pattern a</code>
sequence	<code>seq :: (a -&gt; b -&gt; c) -&gt; Pattern a -&gt; Pattern b -&gt; Pattern c</code>
repetition	<code>rep :: Pattern a -&gt; Pattern [a]</code>

Operations of a pattern matching DSL

### Notes:

- Type `Char -> Pattern Char`  $\equiv$  `Char -> String -> [(Char, String)]`.
- Alternative design for sequencing:  
`seq :: Pattern a -> Pattern b -> Pattern (a,b)`.  
 Less flexible, cumbersome deeply nested tuples when longer sequence patterns are constructed

## Example: Shallow Embedding of a Pattern Matching DSL

---

Functional programs are mathematical objects. We can formulate proofs about their behavior. Consider:

1. `rep` returns the longest match first  

$$(\text{rep } (\text{lit } 'a') \text{ "aab"} = [(\text{"aa"}, \text{"b"}), (\text{"a"}, \text{"ab"}), (\text{"", "aab"})])$$
2.  $\text{alt } p \text{ fail} = \text{alt fail } p = p$   
*(if one alternative is failure, only the other alternative remains, proof based on  $[] ++ xs = xs ++ [] = xs$ ).*
3.  $\text{seq } f \text{ } p \text{ (empty } e) = \text{seq } f \text{ (empty } e) \text{ } p = p$ , if  
 $f \text{ } x \text{ } e = f \text{ } e \text{ } x = x$ , i.e.,  $e$  is the identity of  $f$   
 (proof based on comprehension reasoning).