(Algebraically oriented) domain modeling in Haskell

Ralf Lämmel
Software Languages Team
Faculty of Computer Science
University of Koblenz-Landau

Acknowledgment: This is joint work with Magne Haveraaen, Bergen
Language Design Laboratory, Institutt for informatikk, Universitetet i Bergen

© 2015, Ralf Lämmel, rlaemmel@acm.org

(Algebraically oriented) domain modeling

- Develop domain model:
  - Provide **signature** (sorts + function symbols with types).
  - Provide **properties**.
- Implement domain model:
  - Provide **data representation** for sorts.
  - Provide **function definitions**.
- Engineer domain model:
  - **Modularize**, **test**, **extend**, …

© 2015, Ralf Lämmel, rlaemmel@acm.org

Domain modeling in *Haskell*

- Develop domain model:
  - Provide **signature** as a *parametrized record type*.
  - Provide **properties** as *predicates over signature*.
- Implement domain model:
  - Provide **data representation** for sorts in terms of *algebraic datatypes*.
  - Provide **function definitions** as a *value of the record type*.
- Engineer domain model:
  - **Test** data model with *QuickCheck*.
  - …

Haskell concepts are shown in *italics.*

© 2015, Ralf Lämmel, rlaemmel@acm.org

The Haskell code underlying this lecture

- See **wiki** documentation:
  - http://101companies.org/wiki/Contribution:haskellSpec
- Access code on **GitHub**:
  - https://github.com/101companies/101haskell
  - See contributions/haskellSpec

© 2015, Ralf Lämmel, rlaemmel@acm.org

Types in Haskell

- Primitives types
- Type constructors (e.g., lists)
- Algebraic data types
- Record types

We need types for both the signature and the data representation used in the implementation.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Primitive datatypes

- Int
- Float
- …

Use these types in defining the data representation for the domain.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Predefined type *constructors*

These are parameterized types or so-called type constructors.

- Lists: **[a]**
- Tuples (products): **(a, b), (a, b, c)**, …
- Eithers (sums): data **Either a b** = Left a | Right b
- Maybes (partiality): **Maybe a** = Nothing | Just a
- …

© 2015, Ralf Lämmel, rlaemmel@acm.org

Algebraic datatypes

```haskell
data Shape = Circle Point Int
           | Triangle Point Point Point
           | Composite Shape Shape

data Point = Point Int Int
```


© 2015, Ralf Lämmel, rlaemmel@acm.org

Algebraic datatypes

```
data Company = Company Name [Employee]
data Employee = Employee Name Address Salary
```

Use algebraic datatypes for the data representation for the domain.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Record types

(Syntactic sugar on top of algebraic datatypes)

```
data Shape = Circle Point Int
           | Triangle Point Point Point
           | Composite Shape Shape
```

We only turn *Point* into a record type here, but it's possible for *Shape* too.

```
data Point = Point { getX :: Int,
                     getY :: Int }
```

© 2015, Ralf Lämmel, rlaemmel@acm.org

Plain algebraic datatypes for comparison

```
data Shape = Circle Point Int
           | Triangle Point Point Point
           | Composite Shape Shape

data Point = Point Int Int
```

© 2015, Ralf Lämmel, rlaemmel@acm.org

Using type synonym for comparison

```haskell
data Shape = Circle Point Int
           | Triangle Point Point Point
           | Composite Shape Shape

type Point = (Int, Int)
```

In this case, *Point* is not a new type. It's essentially a macro for another type.

© 2015, Ralf Lämmel, rlaemmel@acm.org

(Parametrized) record types

```
data System company employee name address salary format
   = System {

       -- Constructors
       mkCompany :: name -> [employee] -> Maybe company,
       mkEmployee :: name -> address -> salary -> Maybe employee,

       -- Getters
       getCompanyName :: company -> name,
       getEmployees :: company -> [employee],

       ...

       -- Functionality (query and transformation)
       total :: company -> salary,
       cut :: company -> company,

       ...
     }
```

Use parameterized record types to represent the signature (sorts and function symbols with types) of the domain model.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Values of record types

```
system :: System Company Employee Name Address Salary Format
system = System {

       -- Constructors with precondition checking
       mkCompany = ...,
       mkEmployee = ...,

       -- Getters defined by pattern matching
       getCompanyName = \(Company n _) -> n,

       ...
     }
```

We implement the domain model by defining values of the record tyoe.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Properties

- Invariants — properties of types
- Pre-/post-conditions — properties of functions
- Axioms — other properties over signature

© 2015, Ralf Lämmel, rlaemmel@acm.org

Properties in Haskell

```
prop_map xs = sum (map (+1) xs) == sum xs + length xs
```

```
> :t prop_map
prop_map :: [Int] -> Bool
> prop_map []
True
> prop_map [1]
True
> prop_map [1,3]
True
```

Properties are encoded as predicates.

© 2015, Ralf Lämmel, rlaemmel@acm.org

Properties in Haskell

```
-- | Employee invariant: the salary is non-negative
prop_employee i e
  = getSalary i e >= 0

-- | Company invariant: employee names are unique
prop_company i c
  = names == unames
      where
        -- The list of employee names
        names = map (getEmployeeName i) (getEmployees i c)
        -- The list of names after removing doubles
        unames = nub names
```

Invariants for the types of the 101companies system

© 2015, Ralf Lämmel, rlaemmel@acm.org

Automated testing with QuickCheck in Haskell

```
> quickCheck prop_addition_commutative
+++ OK, passed 100 tests.
> quickCheck no'prop_take
*** Failed! Falsifiable (after 3 tests and 2 shrinks):
2
[]
```

Exploring some basic properties addition and take

See http://101companies.org/wiki/Technology:QuickCheck

© 2015, Ralf Lämmel, rlaemmel@acm.org

Test-data generation for QuickCheck

```
arbitraryEmployee i
  = do
       n <- suchThat arbitrary (/="")
       a <- suchThat arbitrary (/="")
       int <- choose (1::Int,123456)
       let s = fromIntegral int
       return (fromJust (mkEmployee i n a s))

instance Arbitrary Employee
  where
    arbitrary = arbitraryEmployee system
```

© 2015, Ralf Lämmel, rlaemmel@acm.org

Test suite

```
tests i
  = [
      -- HUnit test cases involving the sample company
      testSample "company1" i prop_company True,
      testSample "no_employees1" i prop_total_no_employees True,
      ...,

      -- QuickCheck properties on arbitrary input
      testProperty "nonnegative" (prop_nonnegative i),
      testProperty "company2" (prop_company i),
      ...
    ]

  where
    -- HUnit test case involving the sample company
    testSample l i p b = testCase l (b @=? p i (sampleCompany i))
```

© 2015, Ralf Lämmel, rlaemmel@acm.org

Test log

```
company1: [OK]
no_employees1: [OK]
position1: [OK]
total: [OK]
cut1: [OK]
serialization: [OK]
employee: [OK, passed 100 tests]
company2: [OK, passed 100 tests]
no_employees2: [OK, passed 100 tests]
position2: [OK, passed 100 tests]
cut2: [OK, passed 100 tests]
setSalary: [OK, passed 100 tests]
add_commutative: [OK, passed 100 tests]
zero_unit: [OK, passed 100 tests]
halveSalary: [OK, passed 100 tests]

         Properties  Test Cases  Total
 Passed  9           6           15
 Failed  0           0           0
 Total   9           6           15
Test suite basic-tests: PASS
Test suite logged to: dist/test/haskellSpec-0.1.0.0-basic-tests.log
```

Unit tests for specific test data

Testing with randomized test data

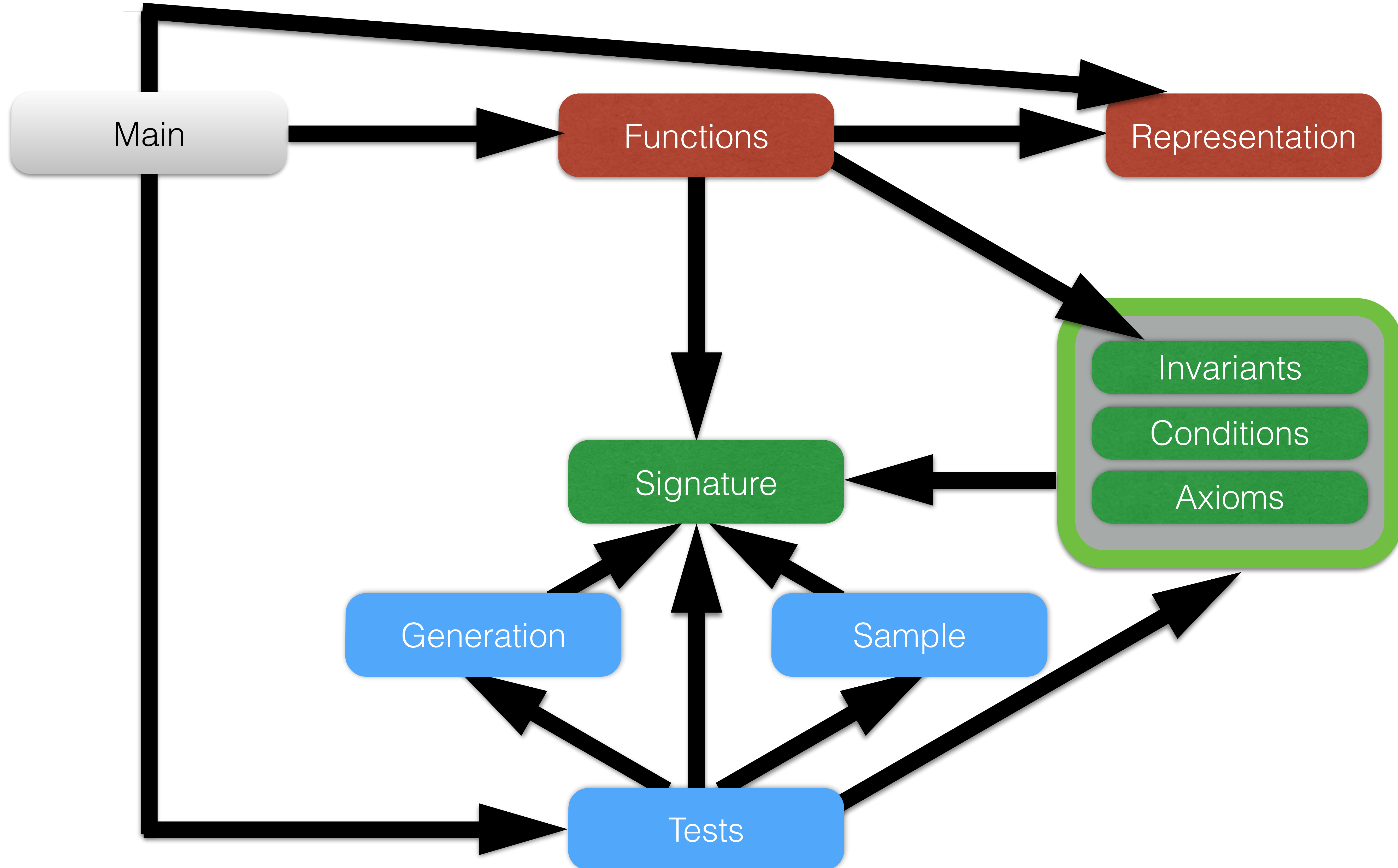
© 2015, Ralf Lämmel, rlaemmel@acm.org

Modularization of Haskell realization

- Domain model
  - Signature — Sorts and function symbols with types
  - Properties
    - Invariants — Invariants for types
    - Conditions — Pre-/post-conditions for functions
    - Axioms — Other properties (axioms over signature)
- Implementation
  - Representation — Data representation for implementation
  - Functions — Implementation of functions
- Testing
  - Sample — Sample data
  - Generation — Test-data generation for data model
  - Tests — Unit tests and QuickCheck tests for properties
- Main — Test driver for concrete implementation

© 2015, Ralf Lämmel, rlaemmel@acm.org

Import relationships



© 2015, Ralf Lämmel, rlaemmel@acm.org

Advanced topics omitted

- Use of *zipper* to model editable data structure
- Use of *monad* to model updatable data structure
- Use of *extensible data types* (or encodings thereof)
- …

© 2015, Ralf Lämmel, rlaemmel@acm.org

End of slide deck

© 2015, Ralf Lämmel, rlaemmel@acm.org