Alloy tutorial

Neil C. C. Brown

June 8, 2009

Introduction

This document is a tutorial for the Alloy generics library. Alloy is similar to other generics libraries, such as Scrap Your Boilerplate (SYB), Uniplate, EMGM and all the rest. Alloy tends to be quite fast (see our paper for benchmarks) because it avoids traversing parts of the data structure that it does not need to.

This is accomplished by generating type-class instances based on the can-contain relation between types. The current set of operations (opset) is trimmed dynamically to remove types that can no longer be contained in the data item being traversed. For more details, see the draft paper.

1 Paradise Benchmark

Below are some sample data types, originally created by Ralf Lämmel as part of the paradise benchmark. They are taken directly from `http://www.cs.vu.nl/boilerplate/testsuite/paradise/CompanyDatatypes.hs`. We will use them for our first few examples of using Alloy.

```haskell
{-# LANGUAGE DeriveDataTypeable #-}
module CompanyDatatypes where

import Data.Generics hiding (Unit)

-- The organisational structure of a company

data Company = C [Dept]                 deriving (Eq, Ord, Show, Typeable, Data)
data Dept    = D Name Manager [Unit]    deriving (Eq, Ord, Show, Typeable, Data)
data Unit    = PU Employee | DU Dept    deriving (Eq, Ord, Show, Typeable, Data)
data Employee = E Person Salary         deriving (Eq, Ord, Show, Typeable, Data)
data Person  = P Name Address           deriving (Eq, Ord, Show, Typeable, Data)
data Salary  = S Float                  deriving (Eq, Ord, Show, Typeable, Data)
type Manager = Employee
type Name    = String
type Address = String

-- An illustrative company
genCom :: Company
genCom = C [D "Research" laemmel [PU joost, PU marlow],
            D "Strategy" blair   []]

-- A typo for the sake of testing equality;
-- (cf. lammel vs. laemmel)
genCom' :: Company
genCom' = C [D "Research" lammel [PU joost, PU marlow],
             D "Strategy" blair   []]

lammel, laemmel, joost, blair :: Employee
lammel  = E (P "Lammel" "Amsterdam") (S 8000)
laemmel = E (P "Laemmel" "Amsterdam") (S 8000)
joost   = E (P "Joost"  "Amsterdam") (S 1000)
marlow  = E (P "Marlow" "Cambridge") (S 2000)
 blair  = E (P "Blair"  "London")    (S 100000)

-- Some more test data
person1 = P "Lazy" "Home"
dept1   = D "Useless" (E person1 undefined) []
```

1.1 The Basics

To generate instances, you must write a short Haskell program that uses the Data.Generics.Alloy.GenInstances module. Here is the example for the CompanyDatatypes module:

```haskell
import CompanyDatatypes
import Data.Generics.Alloy.GenInstances

main :: IO ()
main = writeInstancesTo ( allInstances  GenWithoutOverlapped)
         [genInstance (undefined :: Company)]
         (["module Instances where"
          ,"import qualified CompanyDatatypes"
          ] ++ instanceImports)
         "Instances.hs"
```

The configuration options (the allInstances call) can be ignored for now, but we will return to them later. This program will generate a file named "Instances.hs" which is a complete module with instances for all the data types that can possibly be contained in the Company data type. Note that the Company datatype, and anything it contains, must have a Data instance. This can be done automatically in GHC by simply adding Typeable and Data to the deriving clause for your data types.

You supply the header for the module yourself. The three requirements for Alloy are that you must import the Data.Generics.Alloy module, and (as a qualified import) the module(s) that contain the types you are generating instances for. If you generate all instances as we are, you must also import the Control.Applicative and Control.Monad modules (which we will return to later). Having generated the instances, we can now write the paradise benchmark, that modifies all the salaries in the company. Since we are operating on all instances of a particular data-type, we can use the helper function applyBottomUp (akin to everywhere in SYB):

```haskell
import CompanyDatatypes
import Data.Generics.Alloy
import Instances

 increase  :: Float -> Company -> Company
 increase  k = applyBottomUp (incS k)

incS  :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

main = print $ increase 0.1 genCom
```

This is the most basic use of Alloy. There is also an applyBottomUp2 function that takes two functions operating on distinct types, and applies both of them throughout the data structure.

1.2 Multiple Target Types and Controlled Descent

The previous example applied the salary increase to all employees in the company. Often, traversals need to be more selective, based on nodes further up (i.e. closer to the root) in the tree. We will now consider how to increase the salary of all employees except those that are anywhere in the research department. We must bear in mind that departments may contain departments:

```
{-# LANGUAGE TypeOperators #-}
import CompanyDatatypes
import Data.Generics.Alloy
import Instances

increaseAllButResearch :: Float -> Company -> Company
increaseAllButResearch k = makeRecurse ops
  where
    ops :: Dept :- Salary :- BaseOp
    ops = doDept :- incS k :- baseOp

    doDept :: Dept -> Dept
    doDept d@(D name _ _)
      | name == "Research" = d
      | otherwise = makeDescend ops d

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

main = print $ increaseAllButResearch 0.1 genCom
```

There are several new concepts here. The main concept is the opset (short for operations set). An opset is built using the :- constructor in a *cons*-fashion, terminated by the baseOp function (of BaseOp type). The type of an opset mirrors its construction, showing that it is an opset on the two types Dept and Salary. Usually the type of an opset can be inferred and thus it is a matter of style whether to include the type.

An opset is used primarily with two functions: makeRecurse and makeDescend. Broadly, makeRecurse is used to begin a traversal, and makeDescend is used to continue it; makeRecurse applies the operations to all the largest types (the first ones encountered in a depth-first search) it can find, potentially including the argument you have given it – in contrast, makeDescend begins with the type's children. The increaseAllButResearch function uses makeRecurse to begin the traversal of the company. However, doDept must use makeDescend in order to operate on the children of the Dept. If doDept had used makeRecurse, an infinite loop would have resulted from doDept continually being applied to the same department.

The function works by examining the department name. If the name is "Research", the department is returned unaltered (as we do not wish to alter any employees' salaries in research, even in sub-departments). Otherwise, the traversal continues across the department, looking for further sub-departments, and also salaries to increase as before.

1.3 Type-Class Constraints

So far, we have always used Alloy on known, definite types. When you do this, no type-class constraints are required as the compiler can go and find the type-class instances for the definite types. If you want to operate on parameterised types, you will need to manually add some type-class constraints. In essence, you will need to copy the type-class constraints from any Alloy function you make use of, such as makeDescend, applyBottomUp, etc, that involves the parameterised type. You can see all the constraints in the documentation. We will re-use our previous example to demonstrate:

```haskell
{-# LANGUAGE TypeOperators #-}
import CompanyDatatypes
import Data.Generics.Alloy
import Instances

increaseAllButResearch :: Alloy a (Dept :- Salary :- BaseOp) BaseOp =>
  Float -> a -> a
increaseAllButResearch k = makeRecurse ops
  where
    ops :: Dept :- Salary :- BaseOp
    ops = doDept :- incS k :- baseOp

    doDept :: Dept -> Dept
    doDept d@(D name _ _)
      | name == "Research" = d
      | otherwise = makeDescend ops d

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

main = print $ increaseAllButResearch 0.1 genCom
```

The extra constraint included is taken from makeRecurse. The first parameter of the Alloy type-class is the type that the operation (makeRecurse) is applied to. For makeRecurse the second operation set is full and the third is empty; for makeDescend the reverse would be true. We only need include the constraint for makeRecurse, and not makeDescend because the former operates on a whereas the latter here acts on a definite type, with a definite opset.

1.4 Effects

So far we have seen Alloy operating with pure functions. Often, traversals need to have effects. Alloy supports effects with applicative functors, and as a helpful common case of applicative functors: monads. Consider the case where we want to increase salaries in the company, until we run out of budget. For our example, which salaries are increased will be fairly arbitrary (the order of the tree traversal), but such is life! We will maintain a remaining budget total in a state monad as we traverse.

To use effectful transformations, we must use the AlloyA type-class instead of Alloy. All of the helper functions we have seen so far are available, with an A suffix (for Applicative) and an M suffix (for Monad).

Here is the code for increasing the salaries up to a given budget:

```
import CompanyDatatypes
import Data.Generics.Alloy
import Instances
import Control.Monad.State

increase :: Float -> Company -> Company
increase k c = evalState (applyBottomUpM (incS k) c) 1000

incS :: Float -> Salary -> State Float Salary
incS k (S s)
  = do budget <- get
       if diff > budget
         then return (S s)
         else do put $ budget - diff
                 return (S s')
  where
    s' = s * (1+k)
    diff = s' - s

main = print $ increase 0.1 genCom
```

We can now put together two of our previous examples, to selectively increase the salary of all those not in the research department, up to a given budget:

```
{-# LANGUAGE TypeOperators #-}
import CompanyDatatypes
import Data.Generics.Alloy
import Instances
import Control.Monad.State

increaseAllButResearch :: Float -> Company -> Company
increaseAllButResearch k c = evalState (makeRecurseM ops c) 15000
  where
    ops :: (Dept :-* Salary :-* BaseOpA) (State Float)
    ops = doDept :-* incS k :-* baseOpA

    doDept :: Dept -> State Float Dept
    doDept d@(D name _ _)
      | name == "Research" = return d
      | otherwise = makeDescendM ops d

incS :: Float -> Salary -> State Float Salary
incS k (S s)
  = do budget <- get
       if diff > budget
         then return (S s)
         else do put $ budget - diff
                 return (S s')
  where
    s' = s * (1+k)
    diff = s' - s

main = print $ increaseAllButResearch 0.1 genCom
```

The changes in the increaseAllButResearch function are that the :− constructor has become :−∗ in the effectful version, and similarly baseOp has become baseOpA. The terminator is oblivious to whether the effect in question is an applicative functor or a monad, hence there is only the A-suffixed version. The opset is then parameterised by the monad in question (the bracketing in the type of ops is important).

Apart from these small textual changes, it can be seen that the code is roughly the same.

1.5 Queries

A better example of increasing salaries with a limited budget might be to set a fixed proportional raise, based on the total salaries across the company. An easy way to accomplish this is to first run a query on the company to find the salaries, and secondly to traverse the tree performing the increases on the salaries:

```haskell
import CompanyDatatypes
import Data.Generics.Alloy
import Instances

increase :: Float -> Company -> Company
increase k = applyBottomUp (incS k)

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

totalSalary :: Company -> Float
totalSalary = sum . map (\(S s) -> s) . listifyDepth (const True)

main = print $ increase (5000 / totalSalary genCom) genCom
```

This code uses the listifyDepth function, which is akin to SYB's listify . Given a function of type s -> Bool, listifyDepth returns a list of all items of type s that result in True. Here, all salaries are needed so const True is the suitable definition. listifyDepth is implemented using a traversal with the State monad, and this method can be used to implement other similary query operations.

1.6 Routes

As another example we will consider how to find the employee(s) with the lowest salary in the company and increase just their salary. This could be done with a two-pass query, first finding the lowest salary, and second traversing the entire tree to increment all employees with a matching salary. We instead use this example to demonstrate routes, an experimental zipper-like feature.

```haskell
import CompanyDatatypes
import Data.Generics.Alloy
import Instances
import Control.Monad.State

 increase :: Float -> Route Salary Company -> Company -> Company
 increase k r = routeModify r (incS k)

incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

findMin :: Company -> [Route Salary Company]
findMin c = snd $ execState (applyBottomUpMRoute minSalary c) (Nothing, [])
  where
    minSalary :: (Salary, Route Salary Company)
                 -> State (Maybe Float, [Route Salary Company]) Salary
    minSalary (S s, r)
      = do (curMin, rs) <- get
           case fmap (compare s) curMin of
             Nothing -> put (Just s, [r])
             Just LT -> put (Just s, [r])
             Just EQ -> put (curMin, r : rs)
             Just GT -> return ()
           return (S s)

main = print $ foldr (increase 0.1) genCom (findMin genCom)
```

The route is a path into a tree of type Company, to an item of type Salary. This route can be used for getting, setting or modifying, when applied to the same tree that it was derived from. This means that the whole tree does not need to be traversed again to alter a couple of salaries, which can be a useful saving with large trees.

This strategy is vaguely similar to zippers, but uses mutation rather than any more complex manipulations. Multiple routes can be used to modify the same tree, as long as the final nodes are disjoint (i.e. one does not contain another).

1.7 Maps and Sets

Alloy builds its type-classes using the Data instance for the types given to it. If you derive Data and Typeable using the built-in GHC feature, this will work fine. One problem is that the popular container types, Map and Set do not derive Data in this way and by default Alloy will fail to work with them properly.

As a workaround, Alloy includes two special functions, genMapInstance and genSetInstance. These functions provide a view on maps as a collection of key-value pairs, and also allow processing of elements in sets. We will demonstrate this with a simple example, first some new data types:

```
module MapSet where

import qualified Data.Map as Map
import qualified Data.Set as Set
import Data.Generics
import CompanyDatatypes

type Payroll = Map.Map Person Salary

type Managers = Set.Set Manager

data CompanyInfo = CompanyInfo Payroll Managers
  deriving (Typeable, Data, Show)
```

We will then need to generate some instances:

```
import CompanyDatatypes
import MapSet
import Data.Generics.Alloy.GenInstances

main :: IO ()
main = writeInstancesTo (allInstances GenWithoutOverlapped)
         [genInstance (undefined :: Company)
         ,genInstance (undefined :: CompanyInfo)
         ,genMapInstance (undefined :: Person) (undefined :: Salary)
         ,genSetInstance (undefined :: Manager)]
         (["module MapSetInstances where"
          ,"import qualified CompanyDatatypes"
          ,"import qualified MapSet"
          ] ++ instanceImportsMapSet)
         "MapSetInstances.hs"
```

This is similar to our previous code for generating instances. We call genInstance for Company and CompanyInfo (neither contains the other, but between them they both contain all the data types). We call genMapInstance for our map, passing the key and value types as parameters, and similarly we call genSetInstance. Finally, we use instanceImportsMapSet instead of instanceImports. We can now use these instances to perform some operations on the data types. First, we will define some operations to derive the CompanyInfo information, using a state monad:

```
import CompanyDatatypes
import MapSet
import MapSetInstances
import Data.Generics.Alloy
import qualified Data.Map as Map
import qualified Data.Set as Set
import Control.Monad.State

companyInfo :: Company -> CompanyInfo
companyInfo c = execState (applyBottomUpM2 doEmployee doDept c)
                          (CompanyInfo Map.empty Set.empty)
  where
    doEmployee :: Employee -> State CompanyInfo Employee
    doEmployee (E p s)
      = do modify $ \(CompanyInfo es ms) ->
             CompanyInfo (Map.insert p s es) ms
           return (E p s)

    doDept :: Dept -> State CompanyInfo Dept
    doDept d@(D _ m _)
      = do modify $ \(CompanyInfo es ms) ->
             CompanyInfo es (Set.insert m ms)
           return d
```

We can then perform further operations on the CompanyInfo type. For example, we can increase the salary of all employees with the letter 'o' in their name:

```
incS :: Float -> Salary -> Salary
incS k (S s) = S (s * (1+k))

increaseOs :: Float -> CompanyInfo -> CompanyInfo
increaseOs k = applyBottomUp inc
  where
    inc :: (Person, Salary) -> (Person, Salary)
    inc (P n a, s)
      | 'o' `elem` n = (P n a, incS k s)
      | otherwise = (P n a, s)

main = print $ increaseOs 0.1 $ companyInfo genCom
```

Notice how we define the function to work on key-value pairs in order to process the map entries. If you wish to process the map itself differently, you can define an operation on the map; the map instances we use here are particularly useful for descending into maps (for example if the value in a map can contain types you wish to process).