CSci 658: Software Language Engineering
Sandwich DSL Case Study (Haskell)

H. Conrad Cunningham

3 March 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of March 2018 is a recent version of Firefox from Mozilla.

TODO:

- (For Haskell book) Integrate into existing or new chapter
- Add additional exercises (e.g., construction of a Menu of named sandwiches).

Sandwich DSL Case Study

Introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs). For more discussion of DSL concepts and terminology, see the accompanying notes on Domain-Specific Languages.

In this case study, we design and implement a simple *internal DSL*. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* using the DSLs data types.

The case study first builds a package of functions for creating and manipulating the abstract syntax trees. It then extends the package to translate the abstract syntax trees to a sequence of instructions for a simple "machine".

Building the DSL

Suppose Emerald de Gassy, the owner of the Oxford-based catering business Deli-Gate, hires us to design a domain-specific language (DSL) for describing sandwich platters. The DSL scripts will direct Deli-Gate's robotic kitchen appliance SueChef (Sandwich and Utility Electronic Chef) to assemble platters of sandwiches.

In discussing the problem with Emerald and the Deli-Gate staff, we discover the following:

- A sandwich platter consists of zero or more sandwiches. (Zero? Why not! Although a platter with no sandwiches may not be a useful, or profitable, case, there does not seem to be any harm in allowing this degenerate case. It may simplify some of the coding and representation.)
- Each sandwich consists of layers of ingredients.
- The categories of ingredients are breads, meats, cheeses, vegetables, and condiments.
- Available breads are white, wheat, and rye.
- Available meats are turkey, chicken, ham, roast beef, and tofu. (Okay, tofu is not a meat, but it is a good protein source for those who do not wish to eat meat. This is a college town after all. Oh, there is also a special meat served for football games Thanksgiving week called "bulldog", but it is really just chicken, so we can ignore that choice for our purposes here.)

- Available cheeses are American, Swiss, jack, and cheddar.
- Available vegetables are tomato, lettuce, onion, and bell pepper.
- Available condiments are mayo, mustard, relish, and Tabasco. (Of course, this being the South, the mayo is Blue Plate Mayonnaise and the mustard is a Creole mustard.)

Let's define this as an internal DSL—in particular, by using a relatively *deep embedding*.

What is a sandwich? ... Basically, it is a stack of ingredients.

Should we require the sandwich to have a bread on the bottom? ... Probably. ... On the top? Maybe not, to allow "open-faced" sandwiches. ... What can the SueChef build? ... We don't know at this point, but let's assume it can stack up any ingredients without restriction.

For simplicity and flexibility, let's define a Haskell data type `Sandwich` to model sandwiches. It wraps a possibly empty list of ingredient layers. *We assume the head of the list to be the layer at the top of the sandwich.* We derive `Show` so we can display sandwiches.

```
data Sandwich = Sandwich [Layer]
              deriving Show
```

Note: In this case study, we use the same name for an algebraic data type and its only constructor. Above the `Sandwich` after `data` defines a type and the one after the "`=`" defines the single constructor for that type.

Data type `Sandwich` gives the specification for a sandwich. When "executed" by the SueChef, it results in the assembly of a sandwich that satisfies the specification.

As defined, the `Sandwich` data type does not require there to be a bread in the stack of ingredients. However, we add function `newSandwich` that starts a sandwich with a bread at the bottom and a function `addLayer` that adds a new ingredient to the top of the sandwich. We leave the implementation of these functions as exercises.

```
newSandwich :: Bread -> Sandwich
addLayer    :: Sandwich -> Layer -> Sandwich
```

Ingredients are in one of five categories: breads, meats, cheeses, vegetables, and condiments.

Because both the categories and the specific type of ingredient are important, we choose to represent both in the type structures and define the following types. A value of type `Layer` represents a single ingredient. Note that we use names such as `Bread` both as a constructor of the `Layer` type and the type of the ingredients within that category.

```
data Layer     = Bread Bread          | Meat Meat
               | Cheese Cheese       | Vegetable Vegetable
               | Condiment Condiment
                 deriving (Eq,Show)

data Bread     = White | Wheat | Rye
                 deriving (Eq, Show)

data Meat      = Turkey | Chicken | Ham | RoastBeef | Tofu
                 deriving (Eq, Show)

data Cheese    = American | Swiss | Jack | Cheddar
                 deriving (Eq, Show)

data Vegetable = Tomato | Onion | Lettuce | BellPepper
                 deriving (Eq, Show)

data Condiment = Mayo | Mustard | Ketchup | Relish | Tabasco
                 deriving (Eq, Show)
```

We need to be able to compare ingredients for equality and convert them to strings. Because the automatically generated default definitions are appropriate, we derive both classes `Show` and `Eq` for these ingredient types.

We do not derive `Eq` for `Sandwich` because the default element-by-element equality of lists does not seem to be the appropriate equality comparison for sandwiches.

To complete the model, we define type `Platter` to wrap a list of sandwiches.

```
data  Platter = Platter [Sandwich]
              deriving Show
```

We also define functions `newPlatter` to create a new `Platter` and `addSandwich` to add a sandwich to the `Platter`. We leave the implementation of these functions as exercises.

```
newPlatter  :: Platter
addSandwich :: Platter -> Sandwich -> Platter
```

Exercise Set A

Please put these functions in a Haskell module `SandwichDSL`. You may use functions defined earlier in the exercises to implement those later in the exercises.

1. Define and implement the Haskell functions `newSandwich`, `addLayer`, `newPlatter`, and `addSandwich` described above.

2. Define and implement the Haskell query functions below that take an ingredient (i.e., `Layer`) and return `True` if and only if the ingredient is in the specified category.

```
isBread     :: Layer -> Bool
isMeat      :: Layer -> Bool
isCheese    :: Layer -> Bool
isVegetable :: Layer -> Bool
isCondiment :: Layer -> Bool
```

3. Define and implement a Haskell function `noMeat` that takes a sandwich and returns `True` if and only if the sandwich contains no meats.

```
noMeat :: Sandwich -> Bool
```

4. According to a proposed City of Oxford ordinance, in the future it may be necessary to assemble all sandwiches in *Oxford Standard Order (OSO)*: a slice of bread on the bottom, then zero or more meats layered above that, then zero or more cheeses, then zero or more vegetables, then zero or more condiments, and then a slice of bread on top. The top and bottom slices of bread must be of the same type.

   Define and implement a Haskell function `inOSO` that takes a sandwich and determines whether it is in OSO and another function `intoOSO` that takes a sandwich and a default bread and returns the sandwich with the same ingredients ordered in OSO.

```
inOSO   :: Sandwich -> Bool
intoOSO :: Sandwich -> Bread -> Sandwich
```

   Hint: Remember Prelude functions like `dropWhile`.

   Note: It is impossible to rearrange the layers into OSO if the sandwich does not include exactly two breads of the same type. If the sandwich does not include any breads, then the default bread type (second argument) should be specified for both. If there is at least one bread, then the bread type nearest the *bottom* can be chosen for both top and bottom.

5. Suppose we store the current prices of the sandwich ingredients in an association list with the following type synonym:

```
type PriceList = [(Layer,Int)]
```

   Assuming that the price for a sandwich is base price plus the sum of the prices of the individual ingredients, define and implement a Haskell function `priceSandwich` that takes a price list, a base price, and a sandwich and returns the price of the sandwich.

```
priceSandwich :: PriceList -> Int -> Sandwich -> Int
```

   Hint: Consider using the `lookup` function from the Prelude. The library `Data.Maybe` may also include helpful functions.

Use the following price list as a part of your testing:

```
prices = [ (Bread White, 20), (Bread Wheat, 30),
           (Bread Rye, 30),
           (Meat Turkey, 100), (Meat Chicken, 80),
           (Meat Ham, 120), (Meat RoastBeef, 140),
           (Meat Tofu, 50),
           (Cheese American, 50), (Cheese Swiss, 60),
           (Cheese Jack, 60), (Cheese Cheddar, 60),
           (Vegetable Tomato, 25), (Vegetable Onion, 20),
           (Vegetable Lettuce, 20), (Vegetable BellPepper,25),
           (Condiment Mayo, 5), (Condiment Mustard, 4),
           (Condiment Ketchup, 4), (Condiment Relish, 10),
           (Condiment Tabasco, 5)
         ]
```

6. Define and implement a Haskell function `eqSandwich` that compares two sandwiches for equality.

   What does equality mean for sandwiches? Although the definition of equality could differ, you can use "bag equality". That is, two sandwiches are equal if they have the same number of layers (zero or more) of each ingredient, regardless of the order of the layers.

   ```
   eqSandwich :: Sandwich -> Sandwich -> Bool
   ```

   Hint: The "sets" operations in library `Data.List` might be helpful

7. Give the Haskell declaration needed to make `Sandwich` an instance of class `Eq`. You may use `eqSandwich` if applicable.

Compiling the Program for the SueChef Controller

In this section, we look at compiling the `Platter` and `Sandwich` descriptions to issue a sequence of commands for the SueChef's controller.

The SueChef supports the special instructions that can be issued in sequence to its controller. The data type `SandwichOp` below represents the instructions.

```
data SandwichOp = StartSandwich    | FinishSandwich
                | AddBread Bread   | AddMeat Meat
                | AddCheese Cheese | AddVegetable Vegetable
                | AddCondiment Condiment
                | StartPlatter | MoveToPlatter | FinishPlatter
                  deriving (Eq, Show)
```

We also define the type `Program` to represent the sequence of commands resulting from compilation of a `Sandwich` or `Platter` specification.

```
data Program = Program [SandwichOp]
               deriving Show
```

The flow of a program is given by the following pseudocode:

```
StartPlatter
for each sandwich needed
    StartSandwich
    for each ingredient needed
        Add ingredient on top
    FinishSandwich
    MoveToPlatter
FinishPlatter
```

Consider a sandwich defined as follows:

```
Sandwich [ Bread Rye, Condiment Mayo, Cheese Swiss,
           Meat Ham, Bread Rye ]
```

The corresponding sequence of SueChef commands would be the following:

```
[ StartSandwich, AddBread Rye, AddMeat Ham, AddCheese Swiss,
  AddCondiment Mayo, AddBread Rye, FinishSandwich, MoveToPlatter ]
```

Exercise Set B

1. Define and implement a Haskell function `compileSandwich` to convert a sandwich specification into the sequence of SueChef commands to assemble the sandwich.

   ```
   compileSandwich :: Sandwich -> [SandwichOp]
   ```

2. Define and implement a Haskell function `compile` to convert a platter specification into the sequence of SueChef commands to assemble the sandwiches on the platter.

   ```
   compile :: Platter -> Program
   ```

Source Code

The Haskell source code for this case study is in file `SandwichDSL_base.hs`.

Acknowledgements

I devised the first version of the Sandwich DSL problem for a question on a take-exam in the Lua-based, Fall 2013 offering of CSci 658 (Software Language Engineering). I subsequently developed a full Haskell-based case study for the

Fall 2014 offering of CSci 450 (Organization of Programming Languages). I then converted the case study to use Scala for the Spring 2016 offering of CSci 555 (Functional Programming).

In Spring and Fall 2017, I converted case study document from HTML to Pandoc Markdown and updated it for use in the Haskell-based, Fall 2017 offering of CSci 450. In Spring 2018, I updated the Haskell case study to be more compatible with the other CSci 658 course materials.

In Spring 2018, I also recreated a separate Scala-based version of this case study by combining aspects of this document with the Scala-based version from Spring 2016.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed. The HTML version of this document may require use of a browser that supports the display of MathML.

References

TODO

Concepts

TODO