G53IDS - Interim Report

Embedded Domain Specific Language for
Describing Recipes in Haskell

James Burton - 4251529 - psyjb6

November 26, 2017

Contents

1 Introduction

Consider the following recipe to make a cup of tea:

```
- Boil some water
- Pour over a teabag
- Wait for 3 minutes
- Remove the teabag
- Add milk (optional)
```

This is a very simple but useful recipe that many people will perform, in some cases, many times a day over their lives. What we can realise by looking at this recipe is that it actually consists of many smaller recipes, such as boiling water and combining tea with milk, performed in a certain order. This raises the question, to what extent does the order matter and to what extent can we rearrange things in order to make the recipe more efficient? No doubt you have done this, maybe subconsciously, while cooking at home. Furthermore which steps can be done concurrently in the event that multiple people are cooking e.g. in a professional kitchen with a full brigade?

Perhaps closer to computer science, we could also ask, how could we instruct a robot to do this? After doing some research on robotic chefs it appears that not a huge number exist. There is one home cooking robot [1] which uses motion capture in order to learn recipes. In my opinion this is rather restrictive. It presumes that the human performs the recipe in the optimal manner and it would be very difficult to model a brigade system in this way. In reality there is a limited set of fundamental actions that one becomes able to perform when learning to cook. Recipes can then be performed using a sequence of these actions. Representing recipes like this would allow us to take a robot programmed to perform each of the fundamental actions and tell it how to cook literally anything. This is the same principle as taking code written in a high level language and compiling it down into a sequence of low level actions in assembly language.

So we've established that if we can structure recipes in a more formal way then we will have a great amount of freedom in terms of how we process them whether it be optimisation for a human chef or full on automation. My contributions / planned contributions to this are the following:

- Define a set of combinators as an EDSL in Haskell and show that they can be used to describe a wide variety of recipes (Section 2).

- The combinators describe a recipe but we then need to know how to execute a recipe as a sequence of the fundamental actions mentioned above. Fortunately, as described above, recipes are just a combination of simpler recipes meaning that we can recursively define the actions necessary to perform complex recipes as long as we have manually defined which action are required for each combinator (Section 3).
- We now have a way to describe recipes and a way to show the sequence of actions to complete a recipe but now we need to apply them to something. Using the operational semantics of recipes we can optimise and schedule recipes for a given kitchen system. We can then express this in several ways including printing steps, drawing the cooking process as a graph or simulating the recipe within the given kitchen setup (Section 4).
- It may also be useful to provide an intermediate language between informal English and our combinators, some sort of markup language, that can then be parsed in.

2 Describing Recipes

In this section I shall outline the EDSL for describing recipes. The EDSL is implemented in the functional programming language called Haskell which has been used for many EDSLs in the past [2]. Michael Snoyman, creator of Yesod (a Haskell Web Framework), stated many advantages of using Haskell for EDSLs among which were that the type system helps catch mistakes and Haskell allows us to overload almost any syntax [3].

2.1 A Cup of Tea

Consider our cup of tea example from earlier. We can start by defining all of our ingredients:

```
milk, teabag, water :: Recipe
milk = Ingredient "milk"
teabag = Ingredient "teabag"
water = Ingredient "water"
```

The next step is to start describing what we want to transform those ingredients into for example:

```
boilingWater, blackTea :: Recipe
boilingWater = heat 100 water
```

`ingredient :: String -> Recipe`
The recipe `(ingredient s)` simply represents an ingredient with the name s.
`heat :: Temperature -> Recipe -> Recipe`
`heat t r` means to heat the recipe r to the temperature t.
`wait :: Time -> Recipe`
`wait t` simply means do nothing for t amount of time. The decision to for `wait` to not include a recipe as an argument was very deliberate and will be discussed later in this subsection.
`(><) :: Recipe -> Recipe -> Recipe`
The recipe `r1 >< r2` is the combination of r1 and r2. The details of the method you use to combine the recipes are not yet captured.
`(>>>) :: Recipe -> Recipe -> Recipe`
`r1 >>> r2` represents the sequence of r1 and r2 i.e. perform recipe r1 followed by recipe r2.

Figure 1: Combinators for defining recipes

```
blackTea = (teabag >< boilingWater) >>> Wait 5
```

We've introduced three things here. `heat` is simply a function meaning heat the given recipe to the given temperature. `><` is our combine operator and simply means mix the given recipes together. Finally `>>>` is our sequencing operator meaning do the first recipe then the second. The types for these are as follows:

```
heat :: Temperature -> Recipe -> Recipe
(><) :: Recipe -> Recipe -> Recipe
(>>>) :: Recipe -> Recipe -> Recipe
```

We can now use the above to define our cup of tea recipe as follows:

```
cupOfTea :: Recipe
cupOfTea = blackTea >< milk
```

Now you may notice that we haven't mentioned preparation of ingredients, for example measuring how much of them to use. While experimenting with the basic representation of a recipe and the different ways to interpret it we thought it would be beneficial to keep the combinators as simple and abstract as possible. This means that, for now, we will be describing recipes "cooking show" style where we presume everything is conveniently measured and prepared in a bowl next to the chef.

2.2 Currying - But Not What You Think

In this section I shall introduce the full set of combinators we are currently working with and proceed to build up a more complex recipe.

For the sake of conciseness I shalln't bother listing all the ingredient declarations here as they are relatively straightforward.

We could just create a complex recipe outright however, this would be rather verbose and is likely to repeat a lot of code which somewhat goes against the core principles of DSLs and Haskell. Therefore we shall begin by defining some extra combinators made from our fundamental combinators.

Marinating is a frequently used recipe so let's write a function for it:

```
marinate :: Recipe -> [Recipe] -> Recipe
marinate r' (r:rs) = r' >< marinade
    where marinade = foldr (><) r rs
```

`marinate r' rs` means marinate `r'` in a marinade composed of the list of recipes `rs`.

Due to the compositional nature of our combinators it is easy to create new ones like `marinate` thus allowing us to avoid repeating common sequences of recipes. There is no need to adjust any of our functions that work on recipes to work with `marinate` because it's just a combination of the combinators we already support.

A huge number of recipes start by heating olive oil in a pan then adding something. We define the following combinator for this:

```
preheatOil :: Temperature -> Recipe -> Recipe
preheatOil t r = oil >< r
    where oil = heat t oliveOil
```

That is `preheatOil Medium onion` means heat some olive oil on medium heat then add some onion.

I'm now going to explain the reasoning behind `wait` not taking a recipe as an argument. What this allows us to do is use `wait` in multiple ways. `wait t >< r` means perform recipe r for t amount of time whereas `r >>> wait t` means perform recipe r then wait for t. Using this we can define another combinator to allow us to heat a recipe at a given temperature but also for a certain time:

```
heatFor :: Temperature -> Recipe -> Time -> Recipe
heatFor temp r time = (heat temp r) >< (wait time)
```

So let's get to that more complex recipe I mentioned earlier, Chicken Jalfrezi!

```
spicedChicken :: Recipe
spicedChicken = marinate chicken [cumin, coriander, turmeric]

chickenAndPeppers :: Recipe
chickenAndPeppers = heatFor Medium (cookedChicken >< redPeppers) 10
    where cookedChicken = heatFor Medium spicedChicken 10

jalfreziSauce :: Recipe
jalfreziSauce = heatFor Medium (sauceBase >< spices) 10
    where
        onionMix = onion >< garlic >< greenChilli
        cookedOnions = heatFor Medium (preheatOil Medium onionMix) 5
        spices = cumin >< coriander >< turmeric >< garamMasala
        sauceBase = cookedOnions >< water >< tinnedTomatoes >< spices

chickenJalfrezi :: Recipe
chickenJalfrezi = heatFor Medium chickenJalfrezi' 5
    where chickenJalfrezi' = chickenAndPeppers >< jalfreziSauce >< cherryT
```

And there we have it. Quite a large recipe expressed formally with multiple reusable components. As the project continues, the set of combinators will continue to evolve as we run in to issues. Some of these issues have already been discovered and will be discussed later (Sections 3.3 5.2).

2.3 Conditionals

Mention wait combination.

(Heat Medium oliveOil) ¿¡ onion, are we still heating the onion!?

2.4 Moving Forward

At the time of writing this interim report, the combinators we have are decent but not perfect. There are several crucial pieces of information not captured such as measurements and, as mentioned above, conditionals and optional recipes are not yet implemented.

As the project continues we will naturally be provided with an opportunity to scrutinise the set of combinators we are working with by analysing any issues we run into.

3 Meaning of a Recipe

Our set of combinators allow us to describe a recipe, but what does a recipe actually mean? What are the different ways it can be interpreted? In this section we will discuss these possible interpretations.

3.1 Ingredient Analysis

The end result of all recipes is just the result of transforming and combining the initial ingredients in some way. Extracting a list of ingredients from one of our recipes would be really simple as `Ingredient` is it's own constructor in our recipe type. Once we have this list we can process it in a variety of ways.

One such interpretation could be to evaluate the cost of the recipe and create a shopping list of sorts. Even the cost of a recipe has multiple interpretations. For example the recipe may only use 50g of flour however, if we don't already have some flour then we would have to buy an entire bag so we may wish to use that cost instead.

If we had access to some sort of database with details about ingredients we could work out the nutritional values of a recipe, which diets it supports and the range of flavours present in the dish. This could then possibly be used to recommend susbstitutes to meet certain requirements or, going back to our shopping list idea, in the event that an item is out of stock.

3.2 Executing a Recipe

How do we actually perform a recipe? We need to translate our declarative recipe into a set of actions. Again if we can make these compositional then we will only need to change our definition in the event of a new fundamental combinator.

If we consider our description of a recipe as a tree then the actions we can choose to complete first are the leaves of the tree. This means that we have a set of actions that we can possibly perform at any given time and this set of actions becomes smaller as we move to the root of the tree i.e. as time progresses.

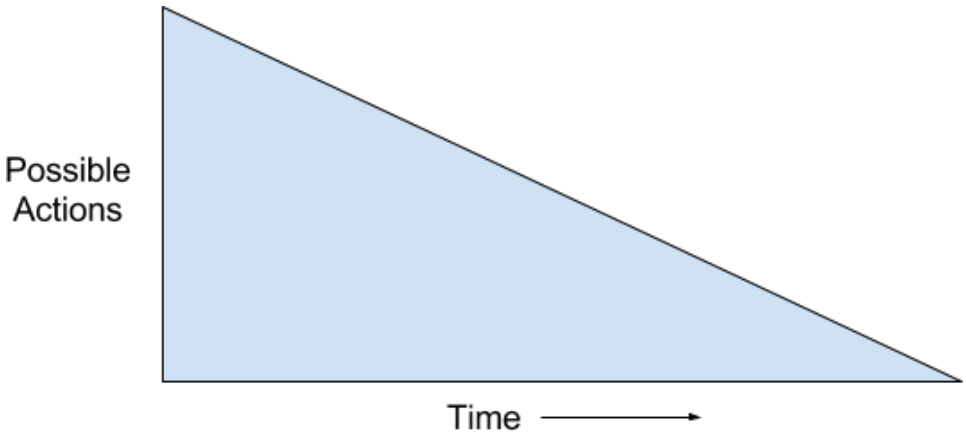

Figure 2: Number of possible actions over time.

We can model this relationship by translating our description of a recipe into a recipe process. A recipe process is a mapping of time to a recipe action `RP = Time -> RA`. A recipe action is a list of the possible actions you could be doing, `RA = [Action]`, actions being fundamental things such as getting an ingredient or stirring something. At this point a full set of actions has not been created but here are some initial thoughts:

```
data Action =
    Get Recipe
    -- Heat
    | Preheat Int
    | Refrigerate Recipe
    | PlaceInHeat Recipe
    | LeaveRoomTemp Recipe
    | Freeze Recipe
    -- Wait
    | DoNothing
    -- Combine
    | PlaceAbove Recipe Recipe
    | PlaceIn Recipe Recipe
    | PourOver Recipe Recipe
    | Mix Recipe Recipe
    deriving Show
```

4 Implementation

Now we can look at the different concrete implementations of this system,

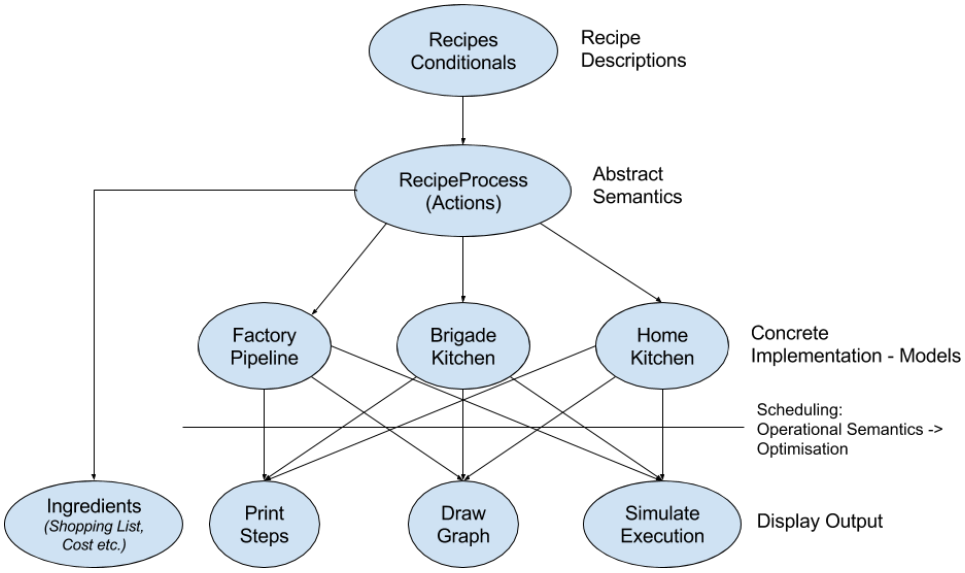

Figure 3: Components of the recipe system.

4.1 Kitchens

4.2 Optimisations, Graphs and Simulations

5 Progress

This project shall use the scrum development methodology. With the gaps between supervisor meetings acting as the sprints. The scrum methodology seems most appropriate for this project as it is naturally very flexibly and doesn't require a huge amount of detailed planning in advance. This suits the research nature of the project as it is impossible to precisely state what will be done and when. Each sprint offers an opporuntity to re-prioritise and alter tasks based upon the progress and findings of the previous sprint.

5.1 Project Management

5.2 Contributions and Reflections

6 Related Work

References

[1] The Guardian. 2015. *Future of food: how we cook*. https://www.theguardian.com/technology/2015/sep/13/future-of-food-how-we-cook

[2] Paul Hudak. Domain Specific Languages. Department of Computer Science, Yale University, December 15, 1997.

[3] Michael Snoynman. O'Reilly Webcast: Designing Domain Specific Languages with Haskell. January 4, 2013. https://www.youtube.com/watch?v=8k_SU1t50M8

[4] Simon Peyton Jones, Microsoft Research, Cambridge. Jean-Marc Eber, LexiFi Technologies, Paris. Julian Seward, University of Glasgow. Composing contracts: an adventure in financial engineering. August 17, 2000.