Massachusetts Institute of Technology

Free Food:
Context-Aware Substitution Engine & Recipe-Oriented Domain Specific Language

Jackson Kearl, Sanchit Bhattacharjee, Cesar Guerrero
github.com/JacksonKearl/FreeFood
Prof. Gerald Sussman
May 17, 2017

1 Project Overview & Inspiration

In the digital age, it has become increasingly easy for anyone to access any recipe in existence. With the countless allergies, pantry deficiencies and specific ingredient aversions across the human population. This reality, combined with the fact that recipes can only exist after some chef, or equally knowledgeable person, creates them, means that countless people are presented with recipes that they cannot create without some significant modifications every day.

As we cannot expect recipe creators to take the time to create every possible modification imaginable, for any possible constraint imaginable, we instead present a language for describing recipes in a easily extensible and parseable format, coupled with a substitution system to intelligently pair with a library of known to be reasonable substitutions, which together will be able to take arbitrary recipes and arbitrary constraints, and intelligently produce the reformed recipe which best fits a users individual constraints.

In more concrete terms, we first present a LISP-like domain specific language for representing recipes, particularly for food, but extensible to any set of instructions composed of combining elements into a whole in a recursive structure. We then present a system which is able to break down a recipe in this form into a more human-readable, step-by-step representation of the recipe, extract its ingredients, and output modified versions of the recipe based on things like dietary restrictions.

In summary, we here provide a system which will allow unto users what we like to call the *Four Food Freedoms*[1]:

0. Freedom to customize food as you wish, for any purpose.
1. Freedom to study the components of a food object, and change it to fit your desires. Access to the recipe is a precondition for this.
2. Freedom to redistribute copies of a food production so you can feed your neighbor.
3. Freedom to distribute copies of your modified recipes to others. By doing this you can give the whole foodie community a chance to benefit from your changes. Access to the recipe is a precondition for this.

[1]Text based, with modifications, on the Four Freedoms as presented on the Wikipedia article, *The Free Software Definition*, as opposed to the Four Freedoms as presented on gnu.org. The former uses a CC BY-SA 3.0, whereas the latter uses a NonDerivitive variant. Interesting that the very text saying that all software should be able to be modified, cannot itself be modified.

2 The Domain Specific Language

2.1 Overview & Inspiration

The domain specific language logically consists of a list-based tree structure, where nodes with children define actions (which can be thought of as functions in a functional context) that are applied to all of the children of that node, and leaf nodes correspond to raw ingredients. Raw ingredients are expressed in the form (name amount unit). For example, baking 2 cups of flour and 1 cup of sugar together would be denoted by

```
((bake ((temp: 350 F) (time: 40 min)))
  (flour ((amount: 2 cups)))
  (sugar ((amount: 1 cup))))
```

This structure can be made increasingly complex as actions are performed on the results of other actions. In the context of the example above, if the flour had to be mixed with 1 cup of water beforehand, the resulting tree representing this would look like:

```
((bake ((temp: 350 F) (time: 40 min)))
  ((mix ())
    (flour ((amount: 2 cups)))
    (water ((amount: 1 cup))))
  (sugar ((amount: 1 cup))))
```

Such a structure isn't exactly the most readable form of a recipe, and as such, code must be written to output recipes in a conventional form that can be used by humans, in terms of an ingredients list and with step by step instructions. However, such a structure has implications on the extensibility of the system due to the fact that the logic makes sense in a functional context and as a result can be thought of as being evaluated, an extension that will be discussed further in the extensibility section of this report.

2.2 Ingredient Extraction

One important feature to make such a system usable to humans is to extract a detailed list of ingredients from the domain-specific language. Such an extraction could be particularly useful not only because the process of gathering ingredients tends to be a separate practice than actually making the recipe, but because the full set of ingredients is not quite apparent or easy to see in the defined recipe language.

In order to implement this, it is important to note the distinction between the two types of objects in this tree: actions (or functions) and ingredients. It is of course the latter that we are concerned about in this system, so one has to be able to check whether an object is an ingredient or an action. There are a couple of ways to do this:

- Check if the second element in any list within the nested list 'tree' recipe is a number or not (with the latter being either a string or a list itself)
- Note that an action can never be an 'argument' to an ingredient, but at some level, every ingredient must be an 'argument' to an action, meaning that ingredients correspond exactly to leaf nodes

The first option is simpler, but reliant on the fact that a number will never be passed to an action in this recipe structure, and thus may be less extensible, so the second option is probably better.

Once it is possible to identify the ingredients that can be found in our recipe language, these ingredients must be compiled. This can be stored as a list of pairs, where each pair corresponds to the form (ingredient amount). To match ingredients, one can just check if the name is the same as a previously processed ingredient and if so, combine it with the previous ingredient. It is on this combining and accumulation of amount that the task becomes harder due to the possibility of differing units. One way to handle this case is to essentially store sums of different units as is, such that the sum of a tablespoon and a cup of sugar would simply equate to "1 tablespoon and 1 cup of sugar." However, potentially if a generic adder is built on top of this, it should be possible to get some more robust additions between units of the same ingredient for a more refined ingredient list.:

2.3 Instruction Extraction

Another important feature of our system is to be able to extract the order in which instructions are executed. Because certain instructions require other instructions to be executed first, it is important that a correct ordering is established.

The process of extracting instructions is kept separate from the process of extracting ingredients. In order to do this, ingredient representation is replaced with a pointer to the ingredient found within the ingredient extraction list. This way, when a procedure is called that requires the instructions list, the ingredient pointer will be evaluated to whatever it points to. This way, if an ingredient substitution is made, and then a call to the instructions is made, the instructions will reflect any changes to the ingredients made by the substitution, without affecting functionality.

Instruction extraction is done by first doing a depth-first search for ingredients on the recipe tree. An ordering list is kept of all steps performed. The depth-first search is done by iterating through the nodes of the recipe tree. Each node is first checked to see what its children are. If a child is an ingredient, a pointer is created that points to said ingredient in the ingredient list. This pointer is then set to be part of an arguments list for the action that will be added to the ordering list. If a child is an action, it is added to an on-going stack of actions, and "the results of <action>” is added to the arguments list. Once all the children of an action are added to the arguments list, the action, along with its arguments are added to the ordering list. The next top-most action is popped from the action stack, and is subsequently analyzed. Once there are no more actions in the stack, the depth-first search is done.

The ordering list is then reversed, thus creating a topological sort of the actions. This new reverse-ordered list is what is kept and analyzed when calls to the instructions list are made. An example of a case when such a call might be made is a procedure that creates a human-readable recipe.

3 Substitution Engine

3.1 Overview & Inspiration

It is often the case that for whatever reason, a user might prefer not to execute a recipe directly as given, and instead would rather make some substitutions, for instance ingredient substitutions in order to remove allergens from the product or make the product more healthy, or process substitutions in order to reduce the cook time of a product or to make up for tools or equipment which are not readily available.

The system we are developing for representing these recipes should handle these preferences and constraints intelligently, performing the required substitutions to fit a user's preference. To do this we provide a library of viable substitutions, along with several methods for specifying appropriate substitutions. The user will be able to input arbitrary constraints, taking forms ranging from a list of allergens/ingredients that must not be present, to a provided list of ingredients which are available, and more.

At the heart of this system is an operation which is best described as a "fan-reduce" substitution process. In this process, we provide the engine with two separate functions for use in determine what substitution to make. For the 'fan' stage, we define a procedure which will be passed an item to be replaced. The procedure will take this item, and return all possible items which could replace it. This output is then passed directly to the 'reduce' procedure, which will be passed first the environment to create a filter specific to the given environment. This filter is expected to reduce the passed list of all possible items from the fan stage down to one item, which it will then return to the caller. We can see that by including various extra detail in the fan output, such as some cost function or similar, we are able to use the reduce function to perform a wide variety of optimization methods. More possible implementations of this fan/reduce setup, along with possible limitations, are described in sections following.

3.2 Implementation (Selected Portions)

The user begins an exchange with the substitution engine with a recipe in the DSL format above specified. In this this example we walk through the falafel example, as present in the codebase under `sub-engine/main.scm`. We start out by defining a starting recipe and the limitations we will be enforcing:

```
(define pantry '(() (deep-fry lemon-juice parsley)))

(define falafel
  '((deep-fry ((temp: 350 F) (time: 4 min)))
    ((let-rest ((time: 2 hour)))
      ((food-process ((time: 5 min)
                      (description: "scrape occasionally")))
        ((soak ((time: 12 hour)))
          (garbonzo-bean ((amount: 1/2 cup)
                          (description: "cannot be canned!"))))
        ((dice ((description: "coarse")))
          (yellow-onion ((amount: 1/4 unit))))
        ((chop ((description: "coarse")))
          (parsley ((amount: 1/4 cup))))
        (garlic ((amount: 2 unit)))
        (lemon-juice ((amount: 2 tablespoons)))))))
```

We next make a call to the main interface of the substitution engine, `customize-recipe`, which accepts as parameters a recipe and a `(things to use, things not to use)` pair. We wrap this in `amb-possibility-list` so as to receive all possible legal substitutions:

```
(amb-possibility-list (print-recipe
  (customize-recipe falafel pantry)))
```

This calls the `customize-recipe` procedure, as defined in `sub-engine/substitution-engine.scm`. This procedure walks the tree from the root downwards, recursively determining if each component is legal given the passed environment:

```
(define (customize-recipe recipe environment)
  (let ((acceptable (acceptable-given-environment environment)))
    (if (base-component? recipe)
      (if (not (acceptable recipe))
          (sub-base-component recipe environment substitutions)
          recipe)
      (if (not  (acceptable (car recipe)))
          (cons (sub-means-of-combination
                     (car recipe) environment substitutions)
                (map (lambda (component)
                        (reform-recipe
                            component environment substitutions))
                     (cdr recipe)))
          (cons (car recipe)
                (map (lambda (component)
                        (reform-recipe
                            component environment substitutions))
                     (cdr recipe)))))))
```

We see that those components decided to be invalid with respect to the environment are replaced via calls to either `sub-means-of-combination` or `sub-base-component`, depending on their type. These two procedures are effectively the same, but separated for possible future improvements. Sample code shown below:

```
(define (sub-means-of-combination
             means-of-combination environment substitutions)
  ((find-in-dict-list
       (car means-of-combination)
       substitutions)
    environment means-of-combination))
```

In this substitutions procedure, we finally look in the substitutions dictionary for an appropriate substitution. The substitution library is defined in a format such as below (from `sub-engine/demo.scm`:

```
(define substitutions `(

    ,(create-substitution
      'deep-fry
      iter-amb
      (lambda (params)
        (let ((temp (find-in-dict-list-or-default 'temp: params 350))
              (params (del-from-dict-list 'temp: params)))

          `(,(cond ((< temp 250) `(pan-fry ,(cons '(temp: low) params)))
                   ((> temp 400) `(pan-fry ,(cons '(temp: high) params)))
                   (else         `(pan-fry ,(cons '(temp: med) params))))))))

    ,(create-substitution
      'parsley
      iter-amb
      (lambda (params)
        `((italian-parsley ,params))))

    ,(create-substitution
      'lemon-juice
      iter-amb
      (lambda (params)
        `(((combine ((name: diluted-lime-juice))
            (lime-juice ,(scale params '((amount: 1/2))))
            (water      ,(scale params '((amount: 1/2)))))
          ((boil ((time: 5 min) (name: concentrated-orange-juice))
            (orange-juice ,(scale params '((amount: 2))))))))))
```

We see that the fan steps are specific to the individual component, specifying things such as the item or procedure to replace a given item or procedure with, along with various transformations which should be applied to that item or procedures parameters when creating the substitution. However, while the fan steps are all unique, there is a single common reduce step shared across all these substitutions, specifically the `iter-amb` procedure, which when given an environment returns a amb-style filter function specific to that environment. We define the `iter-amb` procedure as follows:

```
; return a function which, when given a list of possible substitutions,
; returns an amb which may be any of the valid solutions w.r.t the given
; environment existing among that list.
(define (iter-amb environment)
  (let ((acceptable?
          (lambda (item)
              ((acceptable-given-environment-recursive environment)
                  item))))
    (lambda (possibilities)
      (list-amb (filter acceptable? possibilities)))))
```

Additionally, we let `create-substitution` be a procedure which converts this into a dict-list of lambdas which, when given an item and an environment, will perform the fan-map procedure outlined above, returning a possible substitution:

```
; return (symbol, lambda) pair such that the lambda, when
; passed an environment and a (sym (args ...)) list
; will return a (sym ((args... )) (amb) list-item which is
; a valid substitution for the provided (sym (args ...)) value,
; subject to the restrictions put in place by the environment.
;
; possibilities-generator should be a function specific to this
; symbol, which, when given an arg list, returns an object which,
; when passed to sub-generator-system (called with the environment),
; yields some item which may replace this symbol.
(define (create-substitution sym sub-generator-system
                                possibilities-generator)
  (list
    sym
    (lambda (environment item)
      (if (null? (cdr item))
        ((sub-generator-system environment)
            (possibilities-generator '()))
        ((sub-generator-system environment)
             (possibilities-generator (cadr item)))))))
```

Finally, given this stack, we can see how the `sub-means-of-combination` is able to call upon the fan-reduce substitution procedure implemented in `create-substitution` via functions such as `iter-amb` and the provided `substitutions` dictionary. Upon receiving it's substitution, `sub-means-of-combination` is able to pass that back up to `customize-recipe`, and finally, after the rest of the substitutions have been made, we receive an output such as below:

```
((pan-fry ((temp: med) (time: 4 min)))
 ((let-rest ((time: 2 hour)))
  ((food-process ((time: 5 min) (description: "scrape occasionally")))
   ((soak ((time: 12 hour)))
    (garbonzo-bean ((amount: 1/2 cup) (description: "cannot be canned!"))))
   ((dice ((description: "coarse"))) (yellow-onion ((amount: 1/4 unit))))
   ((chop ((description: "coarse"))) (italian-parsley ((amount: 1/4 cup))))
   (garlic ((amount: 2 unit)))
   ((combine ((name: diluted-lime-juice)))
    (lime-juice ((amount: 1 tablespoons)))
    (water ((amount: 1 tablespoons)))))))

((pan-fry ((temp: med) (time: 4 min)))
 ((let-rest ((time: 2 hour)))
  ((food-process ((time: 5 min) (description: "scrape occasionally")))
   ((soak ((time: 12 hour)))
    (garbonzo-bean ((amount: 1/2 cup) (description: "cannot be canned!"))))
   ((dice ((description: "coarse"))) (yellow-onion ((amount: 1/4 unit))))
   ((chop ((description: "coarse"))) (italian-parsley ((amount: 1/4 cup))))
   (garlic ((amount: 2 unit)))
   ((boil ((time: 5 min) (name: concentrated-orange-juice)))
    (orange-juice ((amount: 4 tablespoons)))))))
```

We see that we have replaced the original missing ingredients with their properly scaled solutions, and we have found and returned every possible substituted recipe, as desired.

4 Extensibility & Future Work

4.1 Extensibility

At its heart, this project serves the purpose of an intelligent `amb` operator, which is able to understand not just that a complaint occurred, but also why it occurred, and what options are available for resolving the issue, along with the relative costs of those resolutions, computed with a client-supplied cost function. This is a powerful construct, and it can be applied to a variety of different constrained optimization problems, including things like circuit construction, where one could pass a list of components they have available, ranging from IC's to resistors and capacitors, and an intelligent system would be able to determine a means of combination of these which mimics a passes circuit design.

As a short example of this system's extensibility, we have built a short demonstration program for determining the best set of resistors to choose from E24 (or any other collection, as desired), which will, when configured in some simple network, mimic a resistor of a given value:

```
; fan stage
(define some-resistance (list-amb E24))
(lambda (params)
        (amb
          `((resistor ((measure: ,some-resistance)))
            ,(find-in-dict-list 'measure: params))

          `(((series ()) (resistor ((measure: ,some-resistance)))
                         (resistor ((measure: ,some-resistance))))
            ,(find-in-dict-list 'measure: params))

          `(((parallel ()) (resistor ((measure: ,some-resistance)))
                           (resistor ((measure: ,some-resistance))))
            ,(find-in-dict-list 'measure: params))
          `(((series ())
               (resistor ((measure: ,(list-amb E24))))
               ((parallel ()) (resistor ((measure: ,some-resistance)))
                              (resistor ((measure: ,some-resistance)))))
            ,(find-in-dict-list 'measure: params))))))
```

Note: In this step we end up breaking the abstraction barrier a bit: we have used what could be considered the "environment" in the fan stage, whereas before we passed only the item being substituted to the fan stage. In the future, it might be a good idea to pass both the environment and the item being substituted to both the fan and reduce stages, as it allows for a simpler interaction with the backend. More about this below.

```
; reduce stage
(define (target-value-amb environment)
  (lambda (possibility)
    (let ((substitution (car possibility))
          (target (cadr possibility)))
      (if (roughly-equal? (resistance (car possibility))
                          target (car environment))
        (car possibility)
        (amb)))))
```

Here, `resistance` *computes the equivalent resistance of a network, and* `roughly-equal?` *is a predicate which returns true if the first and second arguments are equal to the precision specified by the third.*

In this system, `environment` *is a pair consisting of (desired precision, list of resistors available).*

```
; acceptability predicate
(define (acceptable-given-environment environment)
    (lambda (item)
      (if (eqv? (car item) 'resistor)
        (bool (memv (find-in-dict-list 'measure: (cadr item))
                    (cdr environment)))
        #t)))
```

As we can see, with these three simple changes from the existing codebase, we are able to get the program to behave in a very different way. With a bit more work, it would not be difficult to, for instance, return the best solution as determined by some cost function of number of resistors versus exactness, and return that, rather than just a solution within appropriate boundaries, using the power of our system to go from this relatively abstracted `amb` procedure to a more intelligent and useful cost optimization program.

Aside: One issue that kept arising while developing the platform was where to draw the line between allowing everything and restricting to a small set of procedures. Restriction keeps the program simple, for both the creator and the user, but in this case we may have gone too far. Our system expects the world to use our "magic bullet" fan-reduce API for everything, when sometimes it just isn't possible. In this example we have needed to break the very abstraction barrier that we set up, and although the design of the platform made it easy to do so, perhaps it shouldn't have been required in the first place.

The full code for the circuit system, with documentation and example usage, is available with the rest of the code on github, at `sub-engine/circuit-demo.scm`.

The domain-specific language also helps with extensibility here as one could easily add functionality to each of the actions, such as returning the time that action (such as baking) should generally take. The tree-like structure of this language exactly allows such functionality to be recursive. Such an idea could help output other useful information such as estimating the amount of time a recipe would take by summing up the individual output time of each action, for instance.

4.2 Future Work

There are many different features and improvements which could be added to this system in the future. A few that we were able to com up with, in approximately increasing order of difficulty, include:

- Increase human readability of instruction output.
- Provide a means to go from generated human readable recipe to DSL tree.
- Provide a means for operating upon and accessing the items being operated upon in a procedure in the fan-reduce stage.
- Allow for substitutions to propagate changes up the tree rather than strictly down
- Incorporate food chemistry principles into substitution library
- Automatically generate recipes from natural language representations.

5 Conclusion & Final Thoughts

Blah....
Blah....
Blah....
Blah....
Blah....
Blah....
Blah....