B522 Final Paper

Ben Boskin and Paulette Koronkevich

May 2, 2018

1 Overview of project

We first chose μKanren as our DSL for the project, but hit many barriers. Initially, we found difficulty trying to handle the termination of the unification algorithm used, which is not structurally inductive. Secondly, in trying to make an equality relation that was practical for an orderless and potentially infinite data structures. In an effort to use an element of μKanren, we decided to use its monadic computation structure to allow for fair disjunction, as the core for an interpreter for regular expressions.

In addition, in order to further demonstrate our understanding of an interesting type system, we created an abstract machine for System F.

2 Regular Expressions

Regular expressions provide a way to describe regular languages. The grammar of regular expressions that we support is:

$$E ::= \emptyset \mid \epsilon \mid x \mid E \cup E \mid E \bullet E \mid E^* \mid E^+$$

The forms $\cup$ and $\bullet$ require fair disjunction when paired with the infinite repetition that comes from * and $^+$, for example, when generating members of the language described by the RE:

$$(a^*) \bullet (b^*)$$

Depending on the implementation, results for the first 8 members of this language could vary, including possibilities such as:

1. $\{\epsilon, a, aa, aaa, aaaa, aaaaa, aaaaaa, aaaaaaa\}$
2. $\{\epsilon, a, b, ab, aa, bb, aab, abb\}$
3. $\{\epsilon, bbbbbb, aaabbb, abbb, aaab, abb, a, abbbbbb\}$

Clearly, 2 is the better ordering of these, and an ideal implementation would generate all strings of length n before generating strings of length $n+1$. This process is very similar to the evaluation of a miniKanren expression such as:

```
(defrel (in-RE o)
   (conde
      [(== o ‘())]
      [(fresh (almost)
         (in-RE almost)
         (conde
            [(== o ‘(a .  ,almost))]
            [(appendo almost (b) o)]))]))

>(run 8 q (in-RE q))
‘(() (a) (b) (a a) (a b) (a a a) (b) (a a b))
```

This parallelism is why we chose regular expressions as a simplified DSL from μKanren for our project. Our evaluation of the same expression yields the list `‘("" "a" "b" "aa" "bb" "ab" "bbb" "aaa")` which is actually more fair than the μKanren expression!

2.1 Fair Disjunction

The MonadPlus specification includes two operations in addition to the traditional `return` and `bind` operations: `mzero :: ma`, which denotes failure, and `mplus :: ma -> ma -> ma`, which denotes fair disjunction. This `mplus` operator is used in μKanren `conde` expressions, but in our implementation of RE's, we use it to implement $\cup$ and $\bullet$.

For concatenation •, we needed an operator that would mix fair disjunction as well as conjunctive operator, such as string concatenation. We added `mconj :: ma -> ma -> ma`, which is non-interleaving at top level, but uses `mplus` to fairly join the first goal with the rest, where the goal is a string. This ensures, as in the previous example, that `a`'s appear before `b`'s, but with fair ordering, so `b` appears before `ab`. In our original implementation, there were three different functions implementing this, however, by abstracting over the conjunction operator, the three functions became applications of `mconj` with different conjunctive operators.

2.2 Streams

Streams are μKanren's primary data structure, in order to represent infinite lists of goals, which may or may not be attempted. In the Racket implementation, they are either a simple `cons` pair, or a thunk, in order to allow the users to ask for a finite amount of answers when infinitely many are available. We adopted a similar implementation for our regular expressions, where these streams represent all possible strings that satisfy the RE specification.

```
data Stream :  Set -> Set where
   [] :  A : Set -> Stream A
   Cons :  A : Set -> (a :  A) -> (s :  Stream A) -> (Stream A)
   $ :  A : Set -> (⊤ -> Stream A) -> (Stream A)
```

As with μKanren, we have the users specify the amount of strings they want generated from a given RE. In μKanren, the list of answers returned is always a finite list. To ensure this property in our implementation, we defined a property of streams called `isList`, which ensures a stream holds no thunks (i.e. no occurences of `$`). Our implementation has one entry point, the function `get-chars :  ℕ -> RE -> Stream String`, which we have proven using the `isList` property always returns a finite list.

2.3 Depedently typed `printf`

Regular expressions in programming languages are often used in print statements with non-string data types. For example,

```
printf("%n is a number") 3
```

is one way to print `"3 is a number"`. It is an interesting problem to form a typed function that accepts a variable number of arguments of variable types, depending on excape characters found in the string passed to `printf`. Dependent types are a great method determining the type of such print functions!

We found a use for our traditional RE's in the context of `printf` by using the same escape character to have RE's within strings. For example,

```
print(2, "%(%s ∪ %n) is a string or number.") "cat" 3
```

would yield the strings `"cat is a string or number"` and `"3 is a string or number"`. This `print` function also takes a number in order to generate a finite amount of strings, even if some RE generates infinite strings.

2.3.1 Parsing

The first task is to parse the given string passed to `print` into a data structure representing a typed function, which we have named `Printfn`. This function, when given the right number of typed arguments in the right order, will then return an RE, which our system can then generate the specific number of strings. For the forms ∪ and •, our parser needed to do extra work to consume both arguments properly. We require whitespace between the arguments, where a mutually recursive helper functions `split` and `makeRE` will return the two arguments as RE's.

2.3.2 From `Printfn` to Regular Expressions

We have the constructors `return, concat, disj, star,` and `plus` in our `Printfn` type, which all correspond to the RE forms mentioned earlier. In addition, we have `takeStr, takeChar,` and `takeNum` which correspond to the escape characters found in the input string `%s, %c, %n` respectively. These ensure the types and ordering of expressions passed to `print`. When encountering one of these forms, the input is passed to the function captured, ensuring that the types agree.

Finally, the resulting RE is passed to our entry point `get-chars` with the number of outputs desired.

3 System F

System F is a language with polymorphic types. An abstract machine will have a more involved type system than the STLC, with types occuring in expressions. We developed an CEK-style abstract machine for System F and hopefully proved type safety. :(

3.1 Type Safety