An Algebraic Datatype based Language with guaranteed Termination

By Anthony Nicola Cipriano

Motivation

In the past, the vast majority of programmers used imperative languages, where "looping" took place using `GOTO` or `JUMP` statements.

Nowadays most imperative programmers use real loop constructs, like the popular `while` and `for` loops and the less common `do..while` loop.

A different branch of programming is the functional declarative route, where looping is encoded into recursive functions and inductive algebraic data-types, which make it easier to reason about programs.

All of this three different ways of writing programs have one problem in common:

> There is no general method to detect the termination of a program.

In the following sections, I will introduce a programming language, for which every syntactically correct program will come to an end in a finite amount of time.

Turing Completeness and the Halting Problem

The absence of such a general termination checker is proven by the halting problem. This proof generalizes the idea of a program into a mathematical construct, called a Turing Machine.

A programming environment reconstructing a Turing Machine, is often considered "Turing complete".

This means, that a language where termination is predictable is not Turing complete. It is important to mention, that Turing Completeness is completely overrated by most programmers. This paper will contain programs and algorithms written in such a non-Turing complete language, which show that limiting ourselves to a restricted set of functionality is not always a bad thing.

Complexity of Implementation

Every important information about program termination is encoded into the syntax of the language. This means, that it is completely impossible to write down code, which will run forever.

This means, that writing a compiler for this language is not more complex, than implementing Haskell, Elm or related languages.

It would make a lot of sense to implement this language as a subsystem inside of Haskell and related languages, so that algorithms, where termination is a goal can be implemented in such a DSL.

For the sake of familiarity, the following code examples will be shown in a Haskell like syntax, however this is not enforced; this can also be written in a LISP-like or even C-like syntax, as long as pattern matching and inductive/ algebraic data-types are part of the language and the compiler does type-checking (see SML, OCaml, ...).

The theoretical Foundation: Arrow Patterns

For this language to work, several restrictions to common programming environments have to be given:

1. Functions cannot circular depend on each other.
2. A function cannot call itself inside its body.

Those two rules basically reassemble a Caml-like language, without signatures and without `let rec`.

Arrow Patterns are the key to recursion: they move the recursive part into the pattern matching instead of the function body.

Arrow Patterns can be used in place of any sub-pattern.

To make the idea clear, here is some Haskell code for getting the length of a list:

```
length [] = 0
length (_:xs) = 1 + length xs
```

This algorithm for calculating the length of a list will always terminate, because the first argument always decreases in complexity, until it reaches the non-recursive base-case.

The equivalent algorithm written using Arrow Patterns, would look like this:

```
length [] = 0
length (_:(->n)) = 1 + n
```

Here, the recursive part (`length xs`) was encoded into the pattern match, so that not `xs`, but the result `n` will be available in the function body.

This way, the recursion on arbitrary arguments gets syntactically replaced by the recursion on a sub-pattern of the complete argument pattern.

Note, that Arrow Patterns cannot replace the entire pattern match. This means the following examples are not syntactically valid code:

```
f (->y) = y
g x@(->y) = x+y
...
```

Recursion on Algebraic Datatypes

One function, that comes in mind when talking about endless recursion is the factorial function on negative arguments, however this would raise a `No-Match Exception` using Arrow style of coding.

For the sake of demonstration, here is the factorial function on Integer numbers:

```
data Int =
    Pos Nat
  | Neg Nat

factorial (Pos O) = (Pos (S O))
factorial n@(Pos (S (->x))) = n * x
```

As soon as one passes a negative argument to it, a failure would be raised. Languages like Elm are able to detect such a missing-clause at compile time, so this probably wouldn't even compile, however on any cases it runs it would either yield a result after a finite amount of time, or it would raise an exception, again, after a finite amount of time.

A factorial implementation, that would be correctly defined on Natural numbers, could look like this:

```
data Nat =
    O
  | S Nat

factorial O = S O
factorial n@(S (->x)) = n*x
```

Also addition on natural numbers could be easily implemented using recursion on the second argument:

```
plus a O = a
plus a (S (->c)) = S c
```

The correctness of this algorithm is less obvious, written in this form, thus rewriting it using lambdas and a type signature results in the following code:

```
plus :: Nat -> (Nat -> Nat)
plus a = \b ->
  case b of
      O -> a
    | S (->c) -> S
```

The arrow pattern does recursion on `plus a` and not on `plus`, thus this program is correct.

Limitations

This method of recursion seems very limiting at first. For example, how could one implement the GCD function?

It is quite obvious, that the following definition won't work:

```
gcd a 0 = 0
gcd a b = gcd b (mod a b)
```

Instead, one could implement the GCD using the LCM, which itself would be implemented using prime factorisation:

```
factors n = filter isFactor [1..n]
  where isFactor x = n `mod` x == 0

isPrime x = length (factors x) == 2

primeFactorSet = filter isPrime . factors

howOften a n = last $ filter (\x -> n `mod` (a^x) == 0) [1..n]

-- primeFactors :: Nat -> [(Int,Nat)]
primeFactors n = map (\f -> (howOften f n, f)) $ fs
  where fs = filter isPrime $ primeFactorSet n

{-
lcm a b =
  the definition of lcm over primeFactors takes more than two lines,
  so it was left out in this snippet, even though it could easily
  be implemented in a manageable amount of time.
-}

gcd a b = (a*b) `div` lcm a b
```

A Note on Practicality

Having all programs to terminate is probably nothing you always want. Think of a REPL; should it terminate after each I/O-Action? Probably not.

However, using this style of coding for functions in the program where termination should be guaranteed, such as the lexer or parser in the REPL, can reduce a big error source.

Also one could let the `Main` module, export an `init :: a` and a `iter :: a -> Maybe a`, rather than a `main :: IO ()`, so that the iter function gets passed the argument of init, then, when returning `Just _`, applied to itself again, until it returns `Nothing`. This mechanism would move the looping part to the Runtime, rather than the programmer having to deal with it.