Typesafe variadic functions using Continuations

By Anthony Nicola Cipriano

Motivation

Languages like Haskell use lazy evaluation and corecursive "functions" to handle infinite sequences.

Another approach on solving such a problem would be using generators. A generator is an object, that has some starting data and a step to compute more data.

In the following paper, I will expand the idea of generators into continuations, which will allow ML programmers to use seemingly dynamic features, like LISP's `apply` in a type-safe fashion.

Continuations

A continuation is a function that operates on a state and an event, where the state is the data to change under influence of the event.

A Haskell definition of continuations could look like the following:

```haskell
type Continuation state event = state -> event -> state
```

A generator, which maps a state value to another state does not have the extra `event` parameter, however this argument will be useful when implementing things, like functions with variable argument count.

Continuos Values

A continuos value is a state mapped to a continuation.

Its type declaration is the following:

```haskell
data Continuos state event = (:>) state (Continuation state event)
```

The continuos value both represents the current state of a generator and the step it would take to reach the next state under the influence of a special value, the event.

Functions on Continuos Values

The simplest definable functions on continuos values are the getters, both the one returning the state, and the one return the continuation:

```
getState :: Continuos a b -> a
getState (x :> f) = x

getContinuation :: Continuos a b -> Continuation a b
getContinuation (x :> f) = f
```

The most important function on continuos values is the application operator. It takes a continuos value and changes it under the occurence of an event.

Its type signature is the following:

```
($:) :: Continuos a b -> b -> Continuos a b
($:) (x :> f) y = (f x y) :> f
```

The following chapter (practical use) assumes, that the programming language used for the code examples has built-in support for continuos values, thus `c $: y` will always be abbreviated to `c y` using overloaded juxtaposition.

Practical Use

Using continuos values, several functions can be implemented, that would normally be inconvenient in a language, like Standard ML.

For example, one could define a sum "function" as a continuation over the initial state of 0 (identity value), where the event is of type number:

```
sum :: Continuos Int Int
sum = 0 :> (+)
```

The sum function could be used as follows:

```
sum :: Continuos Int Int {- State: 0 -}
sum 1 2 3 :: Continuos Int Int {- State: 6 -}
getState (sum 1 2 3 4 5) :: Int {- 15 -}
```

To also cover more general cases, one could define an `apply` function of the following signature:

```
apply :: Continuos a b -> [b] -> a
```

A useful definition might look like this:

```
applyContinuos :: Continuos a b -> [b] -> Continuos a b
applyContinuos c [] = c
applyContinuos c (y : ys) = applyContinuos (c y) ys

apply c ys = getState (applyContinuos c ys)
```

One could use those functions as follows:

```
applyContinuos sum [1,2,3] :: Continuos Int Int {- State: 6 -}
apply sum [1,2,3] :: Int {- 6 -}
applyContinuos sum [1,2,3] 4 5 :: Continuos Int Int {- State: 15 -}
```