S. Krajči (ed.): ITAT 2018 Proceedings, pp. 18–27
CEUR Workshop Proceedings Vol. 2203, ISSN 1613-0073, © 2018 Michal Štrba and Richard Ostertág

Approaching side-effects in pure functional programming by interpreting data structures as recipes

Michal Štrba, Richard Ostertág

Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, Bratislava, Slovakia
faiface@ksp.sk, ostertag@dcs.fmph.uniba.sk

Abstract: We present a new approach to side-effects in pure functional programming. The approach is not novel in every aspect but rather in combining of multiple already known concepts into a coherent method.

The key concept in this approach is to view some data structures as describing a recipe of what should be done. A program expressed in form of this data structure is then passed to another program called 'interpreter'. An interpreter takes the recipe, examines it, and actually executes all the desired side-effects.

Keeping simplicity, clarity, and avoiding introduction of any unnecessary theory, the recipe data structures tend to roughly conform to the CPS (continuation passing style) model and tend to contain function values inside of them. This means, that expressing them in code involves a lot of anonymous functions and large "continuations" as arguments to other functions. Attempting to write such code in any of the traditional functional languages, most notably Haskell, results in a very messy, unpleasant, and hard to read code. We also think that it's the main reason why techniques described in this paper weren't developed earlier in these languages.

Therefore, we created a new language that makes such code beautiful and pleasant to read. The subtle features of the language that make this new approach to side-effects viable have far reaching implications, mostly by making it possible to write purely functional code that reads top to bottom.

In this paper, we describe the Funky programming language, how it facilitates writing vertical code, how we can write interpreters for recipe data structures, and how we can use the language to transform and combine the recipes in purely functional code and thereby attain great expressivity in writing side-effecting programs.

1 Short introduction to Funky

Funky is a simple language with a small set of orthogonal features that combine very well. The language was designed and implemented by us and was the topic of our bachelor's thesis. This paper is a shortened version of that thesis. The parts left out in this paper are mainly the thorough description of the language. This shouldn't cause any trouble to people generally familiar with functional programming, for whom this paper is intended anyway.

Now we'll briefly describe the important aspects of the language so that the following sections are easy to understand. The description is very dense, full comprehension is not required.

1.1 Names and tokens

Tokens in Funky are generally separated by whitespace, except for these special characters which are always parsed as separate tokens, whether separated by whitespace or not:

```
( ) [ ] { } , ; \ #
```

Aside from these, all tokens are separated by whitespace. Consequentially, identifiers may contain all kinds of symbols. For example, these are all valid identifiers: `fold>`, `fold<`, `empty?`, `skip-whitespace`. Dashes (`-`) are used to separate words in function names instead of underscores or camel-case. However, type names start with upper-case letters and use camel-case.

Tokens starting with a digit or a +/- sign followed by a digit are numbers (valid or invalid). Any series of characters enclosed in single quotes is a character literal (valid or invalid) and similarly, double quotes enclose a string literal (again, valid or invalid). Escaping in characters/strings works as expected.

1.2 Functions

A function definition is signified by the `func` keyword, which is followed by a function name, a colon, the type of the function, equals sign and finally the function body – an expression. Function definitions cannot be nested and expressions are only allowed as function bodies, no expressions outside functions are meaningful. For example:

```
func sqrt : Float -> Float = \x x ^ 0.5

func max : Int -> Int -> Int =
    \x \y if (x >= y) x y

func flip : (a -> b -> c) -> b -> a -> c =
    \f \x \y
    f y x
```

As we can see, function argument names are not specified before the equals sign, instead, all arguments are introduced by abstractions. Funky is very consistent with this – literally all variables in expressions are introduced by abstractions, there isn't a single case otherwise.

Abstractions are very concise – they consist only of a backslash followed by the name of the bound variable followed by the body of the abstraction (function). Multiple arguments are introduced simply by nesting abstractions. The body of an abstraction always spans until the end of the scope (for example, inside parentheses it spans until the closing parenthesis), which prevents unnecessary parentheses in many cases.

Function names consisting solely of special characters (no letters or numbers) are infix functions. All infix functions have the same precedence, which is lower than the precedence of regular prefix function application and are all right-associative. This is to free the programmer from the burden of manually specifying the precedence and associativity of infix functions. Right-associativity was chosen because it's more often useful than left-associativity.

The type system is very similar to the one in Haskell. The main difference is that Funky has no type-classes and disallows higher-kinded types (type variables with arguments), which shall not be confused with higher-order types that are supported in Funky.

Quite unusually and very importantly, Funky supports function overloading – defining multiple functions with the same name, but different types. For example:

```
# map applies a function to all elements of
# a list
func map : (a -> b) -> List a -> List b =
    \f
    fold< ((::) . f) []  # :: is cons

# map applies a function to the potential
# content of a maybe
func map : (a -> b) -> Maybe a -> Maybe b =
    \f \maybe
    switch maybe
    case nothing nothing
    case just \x just (f x)

# useless converts a list of maybe floats to
# a list of maybe ints
# ^ that's a useless comment
func useless : List (Maybe Float) ->
List (Maybe Int) =
    map (map int)
```

This comes very handy in many situations – programmer doesn't have to think about names too much and similar behavior on different types may be assigned the same function name. Also, as we'll see, records are thereby allowed to share field names, which is something that causes a lot of trouble in Haskell.

Function overloading is not arbitrary – overloaded functions with colliding types are disallowed. Colliding types are such types that can be specialized (their type variables can be substituted) into the same type. This is because it would be impossible to determine, or even easily express, which of the colliding functions should be used in many situations.

In the case of need or a desire for clarification, any expression can be type-annotated with a colon:

```
((x : Int) + (y : Int) : Int)
```

1.3 Records

Records are one of the three means of creating own types in Funky (the other two are unions and aliases). Records are similar to structs from C or records from Pascal. They are compound types, a single value containing multiple fields.

A record definition is signified by the `record` keyword, which is followed by the record name, a list of type variables required by the record (if any), and an equals sign followed by a comma-separated list of fields. All fields must be type annotated.

```
record Pair a b = first : a, second : b

record Person =
    name : String,
    age  : Int,  # trailing comma is allowed

record Vec4D =
    x : Float,
    y : Float,
    z : Float,
    w : Float,
```

Funky generates a few functions per record: the constructor and a getter and an updater for each field. For example, in case of the `Person` record, these functions get generated:

```
func Person : String -> Int -> Person

func name : Person -> String
func name : (String -> String) -> Person ->
    Person

func age : Person -> Int
func age : (Int -> Int) -> Person -> Person
```

The constructor takes the values of the record fields in order and returns an instance of the record. A getter simply takes a record value and returns the given field. An updater is more peculiar. It takes a function mapping the record field to a new value and a record value. Then it returns a copy of the original record where the given field is replaced by the result of applying the function to its original value. This approach to updaters was chosen for two reasons: first is that this is usually what we want to do: to update a field according to its previous value; the second reason is that updaters compose very well in this form (they were inspired by Lenses from Haskell).

For example, let's work with these two records:

```
record Point = x : Int, y : Int
```

```
record Segment =
    start : Point,
    end   : Point,
```

Say we have a variable `seg` which is a segment. We can compose getters to access the X coordinate of the starting point:

```
(x . start) seg
```

But we can also compose updaters to change the value of that coordinate and get an updated segment:

```
(start . x) (+ 4) seg
```

To replace the value of a field with a value independent of the previous value of the field, we use the `const` function (`const x` takes one argument and always returns `x`):

```
(start . x) (const 0) seg
```

Let's define one more record:

```
record Plan = segments : List Segment
```

The `map` function can also be used as an updater on a list, and so we can write this to update all Y coordinates of all the end points of the segments in a plan:

```
(segments . map . end . y) (* 2) plan
```

The whole main motivation for the creation of the huge and abstract Lens library in Haskell is avoided in Funky simply by providing clever updater functions.

1.4 Unions

Unions are just like data types in Haskell. Their definition is signified by the `union` keyword followed by the name of the union and a list of type variables, then an equals sign followed by a `|` separated list of alternative forms. For example (`::` is cons, the list prepend function):

```
union Bool = true | false

union Maybe a = nothing | just a

union List a = empty | a :: List a
```

Funky generates a single constructor function for each alternative, which takes all the arguments in the order and returns an instance of the union. For example, these functions get generated for the `List` union:

```
func empty : List a
func (::)  : a -> List a -> List a
```

To get values out of a union value, Funky introduces a special `switch/case` structure that looks like this:

```
func length : List a -> Int =
    \list
    switch list
    case empty
        0
    case (::) \x \xs
        1 + length xs
```

Each case is followed by the name of the alternative, which is followed by a function that accepts the arguments of the alternative and returns the final result of the `switch/case` structure. The arguments in the case body don't have to be mentioned explicitly – the above function could've been written like this:

```
func length : List a -> Int =
    \list
    switch list
    case empty 0
    case (::)  const ((1 +) . length)
```

1.5 Aliases

The simplest way of making type names is by aliasing. Alias simply defines another name for an existing type, albeit a possibly more complex one. For example, the `String` type from the standard library is defined like this:

```
alias String = List Char
```

The alias is perfectly identical to the type on right side of the equals side. They may be used interchangeably. Aliases may also have type variables.

2 Vertical code

No feature is useful unless the benefits of its employment outweigh the costs – the feature must be viable. The new approach to side-effects described in this paper is viable in Funky, but not in Haskell, or any other traditional purely functional language. What makes the difference? Surprisingly, only two subtle syntactic differences, nothing major at the first sight. Yet, these two syntactic differences make code that is otherwise ugly and messy in Haskell beautiful and readable in Funky. Of course, some code that is beautiful and readable in Haskell is in its original form not possible in Funky at all. But, we're not talking about just any code here: we're talking about the kind of code that makes the new approach to side-effects viable and that is – vertical code.

Writing code that reads top to bottom and composes vertically has always been the domain of imperative programming. "Do this, then do that, then repeat this until some other thing happens" is a very natural way of expressing programs. Programmer reads the code top to bottom, remembering the invariants as they occur, until they form a complete picture of the program in their mind.

Functional programs tend to compose differently. Instead of a series of statements, functional programs are mere expressions constructed mostly from function application and abstraction. An expression has no natural "order of execution", it's usually a function application whose arguments are yet other expressions. The arguments can be understood in any order. However, if the arguments are large expressions containing more function applications with still more large expressions as their arguments, the whole thing becomes very inconvenient to read and understand. The reason is that humans operate with only a finite (and fairly small) amount of working memory. Understanding an expression that is syntactically a wide and deep tree requires a lot of working cognitive memory, so it's hard.

Of course, designers of functional languages are well aware of this. Many syntactic features present in those languages are specially designed to tackle this issue. These features include: pattern matching, guards, let bindings, where bindings, or Haskell's do notation. However, none of these features solves the problem fully, because they don't nest very well.

The motivation for designing Funky arose from the harmony experienced while playing with the pure λ-calculus and from the frustration with existing functional languages. Experimenting with the pure λ-calculus gave us the opportunity to step back – see the bigger picture. We saw that instead of adding new syntactic features like the ones described above, all that was needed was to improve the core – make writing ordinary expressions more concise and make it possible to naturally split them into multiple lines. It turned out that this was enough. Funky has no pattern matching, guards, let binding syntax, nor anything analogous to the Haskell's do notation.

The two subtle syntactic features that make it possible are: concise trailing lambdas and the semicolon. We'll demonstrate them on concrete examples: the `if` and the `let` function in Funky.

2.1 `if` and `let`

In Funky, `if` is a simple function from the standard library (body omitted):

```
func if : Bool -> a -> a -> a
```

It takes a condition and two arguments of the same type: then and else, and returns back the correct one. It can be used as a simple conditional expression, just like the if structure in Haskell, or the ternary operator in C:

```
func min : Int -> Int -> Int =
    \x \y
    if (x < y) x y
```

But if the expressions are more complex, this becomes hard to read. Let's take the factorial function as an example (we'll call it `n!` because Funky allows this):

```
func n! : Int -> Int =
    \n
    if (n <= 0) 1 (n * n! (n - 1))
```

This particular code isn't too bad, but it's easy to conceive situations where using the `if` as a simple, one-line conditional expression would be outright unacceptable.

Funky introduces a special syntactic concept: the semicolon. All it does is it puts everything that's after it (in the scope) inside parentheses. It works just like the `$` function in Haskell. So we can rewrite this:

```
if (n <= 0) 1 (n * n! (n - 1))
```

into this:

```
if (n <= 1) 1; n * n! (n - 1)
```

and then split it into multiple lines:

```
if (n <= 1)
    1;
n * n! (n - 1)
```

A more involved example is the infamous FizzBuzz problem. Here's a function that returns the correct output for each number:

```
func fizzbuzz : Int -> String =
    \number
    if ((number % 15) == 0)
        "fizzbuzz";
    if ((number % 3) == 0)
        "fizz";
    if ((number % 5) == 0)
        "buzz";
    string number
```

Here, `if` is used to express a series of cases terminated by a catch-all case. The built-in if structure in Haskell is not suitable for such purposes – one would have to use guards instead. But as we've already said, guards don't nest so well.

In contrary, Funky's `if` function nests perfectly. Both "then" and "else" expressions may be arbitrarily large, nested, vertical expressions.

```
if condition (
    then
    ...
);
else
...
```

Similarly to `if`, let bindings (variable assignments) are not a built-in language construct in Funky. Instead, `let` is a function from the standard library. Here's its full definition (including the body):

```
func let : a -> (a -> b) -> b = \x \f f x
```

The function `let` takes a value and then takes a function to which it passes the value as an argument. Thanks to the Funky's concise function construction (lambda/backslash), this function is a perfectly viable replacement for a let binding construct built directly into the language.

```
let "me stay by your"
(\side reverse side ++ " " ++ side)
```

Here we call the `let` function with two arguments: the first one is the string `"me stay by your"` and the second one is a function. Calling `let` passes the string as the argument to the function and the whole expression evaluates to `"ruoy yb yats em me stay by your"`.

Thanks to the Funky's trailing lambda syntax (function body spans until the end of scope), we can remove the parentheses around the function:

```
let "me stay by your"
\side reverse side ++ " " ++ side
```

Furthermore, we can split the expression into two lines, yielding a very readable piece of code:

```
let "me stay by your" \side
reverse side ++ " " ++ side
```

Now, instead of thinking in terms of function applications and abstractions, we understand the code as an assignment to a variable (immutable, of course) followed by the resulting expression.

To assign multiple variables, we can just stack `let` assignments:

```
let (filter prime? (count 2))   \primes
let (take-while (< 100) primes) \small-primes
let (length small-primes)       \count
string count ++ " small primes (< 100)"
```

This expression evaluates to: `"25 small primes (< 100)"`.

2.2 Vertical functions

Some function are suitable for composing code vertically, either with the help of the semicolon or trailing lambdas. We've seen both in the previous section on the `if` and the `let` function. Other functions aren't suitable for that. Now we'll describe a small theory about these functions that allow vertical composition. We call them *vertical functions*.

There are two main kinds of vertical functions: those that utilize the semicolon and those that involve trailing lambdas. Both are characterized by their signature (type).

The first kind we call *semicolon kind vertical functions*. Their signatures have this general form (where `V` is an arbitrary type and `...` is an arbitrary sequence of `->` applications):

```
... -> V -> V
```

The last argument to a semicolon kind vertical function can be usually understood as a *continuation*, analogous to CPS (continuation passing style).

The `if` function is an example of a semicolon kind vertical function.

In some cases, the above form is violated and the return type doesn't match the last argument, while the function is still used vertically. However, the signature usually follows the mentioned form.

The second kind we call *trailing lambda kind vertical functions*. Their signatures have this general form:

```
... -> (... -> V) -> V
```

The vertical function passes some arguments to the continuation, which then takes on the same role as with the semicolon kind vertical functions. Again, some vertical functions may violate this form, but those cases are rare.

The `let` function is an example of a trailing lambda kind vertical function.

2.3 Viability

The main reason why vertical functions along with their usage haven't seen the light of the day in traditional functional languages, such as Haskell is that they're hardly viable there. The reasons are very subtle. After all, Haskell supports anonymous functions and has the `$` function analogous to the Funky's semicolon.

To demonstrate this, let's take a function for generating all permutations of a list (the code contains some functions we haven't discussed, and won't discuss, in this paper, but the understanding of what the function does is irrelevant now):

```
func permutations : List a -> List (List a) =
    \list
    if (empty? list)
        (yield []; empty);
    pick (permutations (rest list)) \tail
    pick (insert (first list) tail) \perm
    yield perm;
    empty
```

In Haskell, the above code could be rewritten roughly like this:

```
permutations list =
 if null list
  then yield [] $ empty
  else
   pick (permutations (rest list)) (\tail ->
   pick (insert (first list) tail) (\perm ->
   yield perm $
   empty))
```

In case `if` was a function in Haskell, instead of a built-in construct, we can improve the code a bit:

```
permutations list =
 if (null list)
   (yield [] $ empty) $
 pick (permutations (rest list)) (\tail ->
 pick (insert (first list) tail) (\perm ->
 yield perm $
 empty))
```

We could improve the code a little bit more by removing the parentheses around anonymous functions and inserting a few more dollars:

```
permutations list =
 if (null list)
   (yield [] $ empty) $
 pick (permutations (rest list)) $ \tail ->
 pick (insert (first list) tail) $ \perm ->
 yield perm $
 empty
```

The result is still fairly bad, though. The dollar signs all over the place stick out too much and the arrow at the end of the argument list of an anonymous function is similarly detrimental to the overall aesthetics. It's no wonder that vertical functions weren't in fact invented in Haskell. Of course, one would use pattern matching or the do notation to write a similar function in Haskell. However, neither of those features nests very well. The approach taken by Funky is more general.

3 Side-effects and interpreters

Funky's approach to side-effects is unique among functional languages, but after exploring it, this fact comes rather surprising. The approach is so obvious that it's quite curious no other language (to our knowledge) has adopted it before.

The idea is this: a program in Funky is just a value, a data structure. Then there is a special program called *interpreter*. This program interacts with the Funky's runtime (evaluator) to examine this data structure, which serves the role of a recipe. A recipe then tells the interpreter what to do. Contrary to ordinary recipes, these recipes are generated, combined, recursive, and so on, all the goods of functional programming.

The Funky programming language itself has no intrinsic concept of side-effects. In fact, the interpreters themselves add no real concept of side-effects either. Everything still remains just a value. The recipe is a value that can be transformed and manipulated just like any other value. This is where a lot of expressive power comes in as we'll see.

There isn't just one interpreter. In fact, any Funky programmer can make their own interpreters for their own special purposes. Each interpreter works with a different data structure describing the desired side-effects. One interpreter is intended for command-line applications. Another one is for web servers. And yet another is for 2D games. Each interprets a data structure specialized for the given task. In this paper, we'll examine the interpreter for command-line applications.

3.1 Interpreters

Funky's runtime is currently written in Go. As interpreters need to interact with the runtime, they too must be written in Go. This may be expanded to more languages in the future, for example, it would be useful to write Funky interpreters in Java or C++.

To write an interpreter we need to import the `"github.com/faiface/funky"` package and call the `funky.Run` function. This function does all the job regarding command-line flags, reading, parsing, and compiling source files, and returns a runtime value representing the data structure, the recipe, back to the programmer.

```
package main

import "github.com/faiface/funky"

func main() {
    program := funky.Run("main")
}
```

The `funky.Run` function takes one argument: the name of the function containing the recipe value. The return type of the `funky.Run` function is `*runtime.Value`. The `runtime` package is located at `"github.com/faiface/funky/runtime"`. The `*runtime.Value` type provides several methods we can use to interact with the value:

```
func (*runtime.Value) Char() rune
func (*runtime.Value) Int() *big.Int
func (*runtime.Value) Float() float64
func (*runtime.Value) Field(i int)
    *runtime.Value
func (*runtime.Value) Alternative() int
func (*runtime.Value) Apply(
    arg *runtime.Value) *runtime.Value

// these three are implemented using the
// above six
func (*runtime.Value) Bool() bool
func (*runtime.Value) List() []*runtime.Value
func (*runtime.Value) String() string
```

The `Char`, `Int`, and `Float` methods are used to retrieve values of Funky's built-in types. The `Field` function returns the `i`-th field of a record, or the `i`-th argument to a union constructor. The `Alternative` method returns the index of the union constructor of the value. And lastly, the `Apply` function takes another runtime value and applies it to the function in the receiving runtime value and returns the result of this application.

All the above functions crash if they're used on values of wrong types. For example, calling `Alternative` on a record value crashes, and calling `Char` on a float value likewise.

The last three functions are just for convenience because booleans, lists, and strings are very widely used types.

The interpreter sometimes needs to fabricate new runtime values not originating in the Funky program. For example, when a command-line interpreter loads a character from the input, it needs to make a character value and pass it to the program. These functions are provided in the "github.com/faiface/funky/runtime" package for this purpose:

```
func MkChar(c rune) *runtime.Value
func MkInt(i *big.Int) *runtime.Value
func MkFloat(f float64) *runtime.Value
func MkRecord(fields ...*runtime.Value)
    *runtime.Value
func MkUnion(alt int, fields ...*runtime.Value)
    *runtime.Value

// these three are, again, implemented using
// the above five
func MkBool(b bool) *runtime.Value
func MkList(elems ...*runtime.Value)
    *runtime.Value
func MkString(s string) *runtime.Value
```

Their meaning is clear, so we'll avoid explaining that.

3.2 Interactive command-line programs

The data structure serving the role of a recipe we chose for simple interactive command-line programs is strikingly simple:

```
union IO =
    done               |
    putc Char IO       |
    getc (Char -> IO) |
```

It is a kind of a linked list, with three types of nodes. One signals the end of the program: `done`. The next one – `putc` – says that a character should be printed and the program should continue in some way. The first argument to `putc` is the character to be printed. The other argument is the rest of the program – a continuation. The last node – `getc` – is requesting a character from the input. It has one argument: a function. The interpreter should read the character and pass it as an argument to this function. The function then returns the rest of the program.

Funky always generates constructor functions for a union and these are the ones generated for `IO` (bodies omitted, internal to the compiler/runtime):

```
func done : IO
func putc : Char -> IO -> IO
func getc : (Char -> IO) -> IO
```

The imporant thing to notice is that `putc` is a semicolon kind vertical function and `getc` is a trailing lambda kind vertical function. This makes it easy to compose interactive command-line programs in a natural, imperative-like style.

For example, here's a "cat" program, a program that simply copies the input to the output:

```
func main : IO =
    getc \c
    putc c;
    main
```

This program has no `done` node, it's an infinite data structure. Before we run it, though, we need an interpreter. Here it is (badly formatted, because the lack of horizontal space):

```
package main

import (
    "bufio"
    "io"
    "os"
    "github.com/faiface/funky"
    "github.com/faiface/funky/runtime"
)

func main() {
    program := funky.Run("main")
    in := bufio.NewReader(os.Stdin)
    out := bufio.NewWriter(os.Stdout)
    defer out.Flush()
loop:
    for {
        switch program.Alternative() {
        case 0: // done
            break loop
        case 1: // putc
            out.WriteRune(program.Field(0).Char())
            program = program.Field(1)
        case 2: // getc
            out.Flush()
            r, _, err := in.ReadRune()
            if err == io.EOF {
                break loop
            }
            program = program.Field(0).
                        Apply(runtime.MkChar(r))
        }
    }
}
```

The interpreter enters a loop where it checks the program node type and acts accordingly, always proceeding to the continuation until reaching the `done` node.

Now we can run the "cat" program (user input is emphasized, `funkycmd` is the name of the interpreter):

```
$ funkycmd cat.fn stdlib/*.fn stdlib/
    funkycmd/*.fn
hello, cat!
hello, cat!
do you cat?
do you cat?
you do cat!
you do cat!
^D
```

At the end, the user pressed the Ctrl+D combination to signal the end of file which caused the program to finish.

We can't do much with just `done`, `putc`, and `getc`, not at least conveniently. That's why we'll now show how to define more complex functions on top of the basic ones to get a more powerful – and sometimes surprisingly powerful – behavior.

3.3 `print`, `println`, `scanln`

The first function with a more complex behavior that we're going to tackle is `print`. The `print` function prints a whole string instead of a single character as `putc` does (it doesn't *do it*, but it instructs the interpreter to do it, and similarly, `print` instructs the interpreter to print a string).

```
func print : String -> IO -> IO =
    \s \next
    if (empty? s)
        next;
    putc (first s);
    print (rest s);
    next
```

The `print` function takes a string and a continuation – the rest of the program. Then it recursively describes how to print the string – print the first character and then continue printing the rest until we printed the whole string. Alternatively, `print` can be defined with a right fold:

```
func print : String -> IO -> IO =
    \s \next fold< putc next s
```

Now we can use `print` to create a convenience `println` function, which additionally prints a newline at the end of the string:

```
func println : String -> IO -> IO =
    print . (++ "\n")
```

The next function we're going to define is `scanln`, which scans a whole line from the input (excluding the newline character) and passes its content. While `print` and `println` prepended some `putc` nodes to the continuation, `scanln` prepends some `getc` nodes, accumulates the line and passes it to the continuation, which accepts one argument: the line string. It's a trailing lambda kind vertical function.

```
func scanln : (String -> IO) -> IO =
    \f
    "" |> fix \loop \s
        getc \c
        if (c == '\n')
            (f (reverse s));
        loop (c :: s)
```

The body makes use of the `fix` function (fix-point operator, `fix f = f (fix f)`) to insert inline recursion. This is a common pattern in Funky. It's used to avoid creating unnecessary helper functions in places where the recursion needs to remember more arguments than the original function has. The recursion in our case has to remember the accumulated string starting from an empty string. The `|>` function (`x |> f = f x`) passes the empty string as the initial value to the recursion.

With the help of `print`, `println`, and `scanln`, we can make more involved programs. Here's a number guessing game:

```
func main : IO =
    println "Think a number from 1 and 100.";
    100 |> 1 |> fix \loop \min \max
        let ((min + max) / 2) \mid
        print (string mid ++ "? ");
        scanln \response
        if (response == "less")
            (loop min (mid - 1));
        if (response == "more")
            (loop (mid + 1) max);
        if (response == "yes")
            (println "Yay!"; done);
        println "Say one of less/more/yes.";
        loop min max
```

And here's an example running of the program:

```
Think a number from 1 and 100.
50? no
Say one of less/more/yes.
50? nope
Say one of less/more/yes.
50? less
25? more
37? more
43? less
40? more
41? more
42? yes
Yay!
```

3.4 `ungetc`, `skip-whitespace`, `scan`

The `print`, `println`, and `scanln` functions only "prepend" operations to the continuation. But since `IO` is a fully transparent data structure, we can define transformations that penetrate it, twist it around, or transform it in any other way.

The first and a very useful example of such a function is `ungetc`. It is used to "push a character back on the input", so that the next `getc` call will get it. The plain `IO` data structure has no such functionality and we can't get any similar behavior just by sequencing `done`, `putc`, and `getc`. What we need is we need to write a function that takes a character and a continuation, then searches through the continuation until it finds the first `getc` node, and finally passes the character to that node. Since `IO` is just a transparent data structure, this is quite easy:

```
func ungetc : Char -> IO -> IO =
    \c \io
    switch io
    case done
        done
    case putc \d \jo
        putc d;
        ungetc c;
        jo
    case getc \f
        f c
```

The `ungetc` function examines the top node of the continuation and recursively propagates itself down the data structure until it finds a `getc` node. When it does, it passes the character to the function of the `getc` node and turns it into its result.

The `ungetc` function is particularly useful for implementing the `scan` function. We've already implemented `scanln`, which scans whole lines. The `scan` function, on the other hand, scans the next full word (a continuous sequence of characters not containing any whitespace) on the input. To do that it first needs to skip all the whitespace preceding the word, then scan the word, but avoid scanning the first whitespace character after the word. We'll see how `ungetc` comes to help with this.

First, we'll make a general function for skipping whitespace on the input:

```
func whitespace? : Char -> Bool =
    \c
    any (c ==) [' ', '\t', '\n', '\r']

func skip-whitespace : IO -> IO =
    \next
    getc \c
    if (whitespace? c) (
        skip-whitespace;
        next
    );
    ungetc c;
    next
```

The `skip-whitespace` function continuously reads characters from the input until it reaches a non-whitespace character. It wasn't supposed to read this character, but it needed to in order to determine whether to stop skipping or not. So it uses `ungetc` to put it back on the input.

With the help of `skip-whitespace`, here's `scan`:

```
func scan : (String -> IO) -> IO =
    \f
    skip-whitespace;
    "" |> fix \loop \s
        getc \c
        if (whitespace? c) (
            ungetc c;
            f (reverse s)
        );
        loop (c :: s)
```

It uses the `skip-whitespace` at the beginning, then it enters a loop where it accumulates the word until it reaches a whitespace again. This whitespace wasn't supposed to be read by `scan`, so it's put back on the input by `ungetc`.

Here's a simple calculator program demonstrating the `scan` function:

```
func main : IO =
    print "> ";
    scan \x-str
    scan \op
    scan \y-str
    # float x-str returns Maybe Float
    # hence the call to extract
    let (extract (float x-str)) \x
    let (extract (float y-str)) \y
    println (
        if (op == "*") (string (x * y));
        if (op == "/") (string (x / y));
        "invalid operation: " ++ op
    );
    main
```

And here's its running:

```
> 10 / 3
3.3333333333333335
> ^D
```

3.5 `reverse-lines`

The last example in our exploring of the possibilities of the `IO` data structure is a rather peculiar one: not practically useful, but quite showing of the potential present here.

The function is called `reverse-lines` and its job is to reverse all the lines on the output. Lines come to the output in two ways: either a sequence of `putc` nodes terminated by a `putc` of a newline, or a sequence of `putc` nodes terminated by a `getc` node – all the pending output must be shown to the user before requesting input and the user input will be entered by a newline.

```
func reverse-lines : IO -> IO =
    \io
    io |> "" |> fix \loop \s \(io : IO)
        switch io
        case done
            done
        case putc \c \jo
            if (c == '\n') (
                println s;
                loop "";
                jo
            );
            loop (c :: s);
            jo
        case getc \f
            print s;
            getc \c
            loop "";
            f c
```

The `reverse-lines` function starts a loop using inline recursion with `fix` to accumulate the line to be reversed. It removes the original `putc` nodes from the data structure and keeps accumulating until reaching one of the above-mentioned conditions for terminating a line. When one of the conditions occurs, it transforms the accumulated reversed line into a proper series of `putc` calls using `print` or `println`.

Here's the small program augmented by `reverse-lines`:

```
func main : IO =
    reverse-lines;
    print " What's your name? ";
    scan \name
    println ("Hello, " ++ name ++ "!");
    done
```

And here's its running:

```
 ?eman ruoy s'tahW
!lahciM ,olleH
```

Unimportant to this paper, this `reverse-lines` function isn't perfect. For example, it fails to reverse the lines of the "cat" program, because that one has a `getc` before each `putc` and so no full line ever gets accumulated. The solution to this problem is left as an exercise to the reader.

4 Conclusion

Combining the concept of vertical code with the new approach to side-effects in a language that makes it viable resulted in a whole new approach to structuring purely functional code. We saw that purely functional programs can be expressed in a way that is familiar to an imperative programmer. In Funky, this doesn't come from artificially implanted syntactic features, but instead stems naturally from the core, general concepts in the language itself.

At the moment, Funky is virtually unknown. We will make all the efforts to get the word out there, because we believe we've got something worthy in our hands.