Rippl: Recursively Inferred Pure functional Pythonic Language

Da Hua Chen (Tester), Hollis Lehv (System Architect),
Amanda Liu (Language Guru), Hans Montero (Manager)

1 Introduction

Rippl is a functional language that leverages the safety and elegance of pure languages like Haskell with the intuitive syntax of Python. With list comprehensions, a strong static type system implementing Hindley-Milner style inference, higher-order functions, and simple syntax, Rippl is a powerful computational language with strong support for list-oriented calculations as well as support for association of items of different types in the form of tuples. Rippl will be an appropriate introduction for users without prior functional programming experience who want to effectively and safely perform complex mathematical calculations.

2 Motivation

Functional programming languages generally support powerful and concise language constructs such as higher-order functions, partial application, recursion, and immutability, but at the cost of hard-to-read syntax that is not friendly to unfamiliar users. High-level imperative languages like Python support a highly human-readable syntax but allow mutable state which can lead to bugs and a weak, dynamic typing system which pushes errors to runtime. Rippl has many of the paradigmatic features of functional programming, such as higher-order functions, recursion, and lazy evaluation. Its syntax is intended to make programs written in a functional style more readable and intuitive to users who are more accustomed to other programming paradigms.

3 Language Features

Rippl sets out to provide a clean and intuitive functional programming experience like no other language can. As such, it is inspired by some of the best features of some of the best languages out there.

3.1 Clean Code

One of Rippl's biggest influences is Python, for its simple operator syntax. Keywords like `and`, `or`, and `not` make it closer to natural language and therefore more human-readable. Rippl extends this user-positive experience to functional operations that in other languages take the form of symbols or strangely abbreviated words, which obfuscates code. Rippl instead offers descriptive yet concise list operators like `cons` and `cat` and tuple operators like `first` and `sec`. Lastly, list comprehensions receive a fresh look in Rippl, with the introduction of the `over` keyword and a more familiar mathematical notation, reminiscent of set builders.

3.2 Type Inference

Since Rippl is a statically typed language, it can take advantage of an inferenced type system. Specifically, Rippl implements the Hindley-Milner system, which is the basis for Haskell's type system. By having language-native operators and literals bootstrapped in the typing context with an explicit type, we are able to infer the type of expressions using Hindley-Milner's core syntactic type inference rules. This type system also allows for parametric polymorphism, meaning the type of an expression may be expressed in terms of type variables and will type check with any expression of a concrete type that matches the type schema. Also like Haskell, Rippl will not require type annotations, but will allow them for type-checking and documentation purposes.

3.3 First-class Functions

The use of first-class functions encompasses many of the core features of the functional programming paradigm. One such feature is higher-order functions where arguments and return values may be other functions. Currying is another feature that allows us to view functions, function compositions, partially applied functions, and return values all as arrow types. Having functions in a curried arrow-type form allows us to employ partial application tied together with closures, as is idiomatic in functional languages.

3.4 Lazy Evaluation

Rippl supports a Haskell-style, non-strict lazy evaluation in which the value of expressions isn't fully evaluated until it is needed in another expression. This is particularly useful in that Rippl also supports immense list constructions like infinite lists and cycles, which are native to Haskell. Given the infinitely self-referential definition of these structures, they are kept unevaluated until an element is needed for further computation.

4 Syntax

e ::=		*Expressions*
	c	literals
|	x	variables
|	$e\ e$	application
|	`fun` x `->` e	lambda abstraction
|	`let` x `=` e `in` e	let binding
|	`if` e `then` e `else` e	if-then-else
|	γ	list comprehension
γ ::=		*List Comprehensions*
|	`[`e`...]`	infinite list
|	`[`e`...`e`]`	ranged list
|	`[`e `|` x `over` γ`,]`	parametric list
|	`[`e `|` x `over` γ`,`e`]`	qualified list
σ ::=		*Proper Types*
|	`int`	integer
|	`float`	floating point number
|	`bool`	boolean
|	`char`	character
τ ::=		*Higher Order Types*
|	`[]`	list
|	`()`	tuple

```
// Single line comments will be marked by two forward slashes

/* Multiline comments will be delimited by slash-stars
   Similar to a C-like program */
```

5 Operators

5.1 Numeric

Our numeric or *num* type represents the typeclass that encompasses *int* and *float*. This use of this typeclass is mainly meant to aid in inference and protect the user from explicit typecasting. *int*, *float*, and *num* types may be used explicitly in user-annotated types. However, numeric literals are inferred to be of type *int* unless the literal is explicitly written with a "." decimal point and inferred to be of type *float*.

Operator	*Type*	*Function*
+	$num \rightarrow num \rightarrow num$	addition
-	$num \rightarrow num \rightarrow num$	subtraction
*	$num \rightarrow num \rightarrow num$	multiplication
/	$num \rightarrow num \rightarrow num$	division
%	$int \rightarrow int \rightarrow int$	modulus
^	$num \rightarrow num \rightarrow num$	power
>	$num \rightarrow num \rightarrow bool$	greater than
>=	$num \rightarrow num \rightarrow bool$	greater than or equal
<	$num \rightarrow num \rightarrow bool$	less than
<=	$num \rightarrow num \rightarrow bool$	less than or equal
==	$num \rightarrow num \rightarrow bool$	equal
!=	$num \rightarrow num \rightarrow bool$	not equal

5.2 Boolean

Boolean values are of type *bool* and consist of the literals `true` and `false`.

Operator	*Type*	*Function*
`and`	$bool \rightarrow bool \rightarrow bool$	and
`or`	$bool \rightarrow bool \rightarrow bool$	or
`not`	$bool \rightarrow bool$	negation

5.3 Character

Character literals are delimited by single quotes and are of type *char*. Strings are constructed and treated as lists of *char*.

Operator	*Type*	*Function*
==	$char \rightarrow char \rightarrow bool$	equal

5.4 Lists

A list is a first order type that is parametrically polymorphic in the type of its elements. All elements of a list must be the same type. Strings are handled and represented internally as lists of `chars`, but literal strings may be written using double quotes. Lists as well as list types are delimited with brackets (e.g. `[int]`, `[0,3,1]`, `"Hello"`).

Operator	*Type*	*Function*
`cons`	$a \rightarrow [a] \rightarrow [a]$	construct
`head`	$[a] \rightarrow a$	head
`tail`	$[a] \rightarrow [a]$	tail
`len`	$[a] \rightarrow int$	length
`cat`	$[a] \rightarrow [a] \rightarrow [a]$	concatenate

5.5 Tuples

A tuple is a first order type, parametrically polymorphic in the types of its elements. A tuple can contain two elements of any type. Tuples as well as their types will be delimited by parentheses (e.g. `("PLT",4118)`, `([char],int)`).

Operator	*Type*	*Function*
`first`	$(a, b) \rightarrow a$	first element of tuple
`sec`	$(a, b) \rightarrow b$	second element of tuple

6 Sample Programs

6.1 Infinite Sum

This program returns the sum of the first n positive integers using an infinite list.

```
1  inf_sum :: int -> int
2  inf_sum n = let infinity = [1...] in
3      let rec_inf_sum x list acc = if x == 0 or (len list) == 0
4          then acc
5          else rec_inf_sum (x-1) (tail list) ((head list) + acc)
6      in rec_inf_sum n infinity 0
```

6.2 Collatz Conjecture

The Collatz conjecture is a conjecture in mathematics surrounding the iterative function shown below:

$$\begin{cases} \frac{1}{2}x & x \texttt{ is even} \\ 3x+1 & x \texttt{ is odd} \end{cases}$$

Lothar Collatz proposed that this sequence will always converge to 1 starting from an arbitrary positive integer. Given a starting integer, the following program returns a list of iterations that ends in 1. The program hence uses 1 as its base case even though it hasn't been mathematically proven that all numbers will eventually reach this case, but we believe in Lothar.

```
collatz :: int -> [int]
collatz n =
    let rec_collatz n list =
        if n == 1
            then 1 cons list
        else if n % 2 == 0
            then rec_collatz (n / 2) (n cons list)
            else rec_collatz (3*n +1) (n cons list)
    in rec_collatz n []
```

6.3 Weak Prime Number Theorem

Bertrand's Postulate stated in the Weak Prime Number Theorem that there is always a prime number to be found between some n and its double $2n$. This postulate was later proven by Pafnuty Chebyshev and refined by Paul Erdös. The following program includes a function that determines the primality of a number and a function that takes a n and returns the first prime between n and $2n$.

```
is_prime :: int -> bool
is_prime n =
    let max = n / 2 in
    let range = [2...max] in
    let divisors = [x | x over range, n % x == 0] in
    len divisors == 0

prime_number_theorem :: int -> int
prime_number_theorem n =
    let range = [(n+1)...2*n] in
    let odd_range = [x | x over range, x % 2 != 0 ] in
    foldl (fun prev -> fun curr -> if is_prime then prev else curr)
        (head odd_range) odd_range
```