Pipes

Language Design and Example Programs

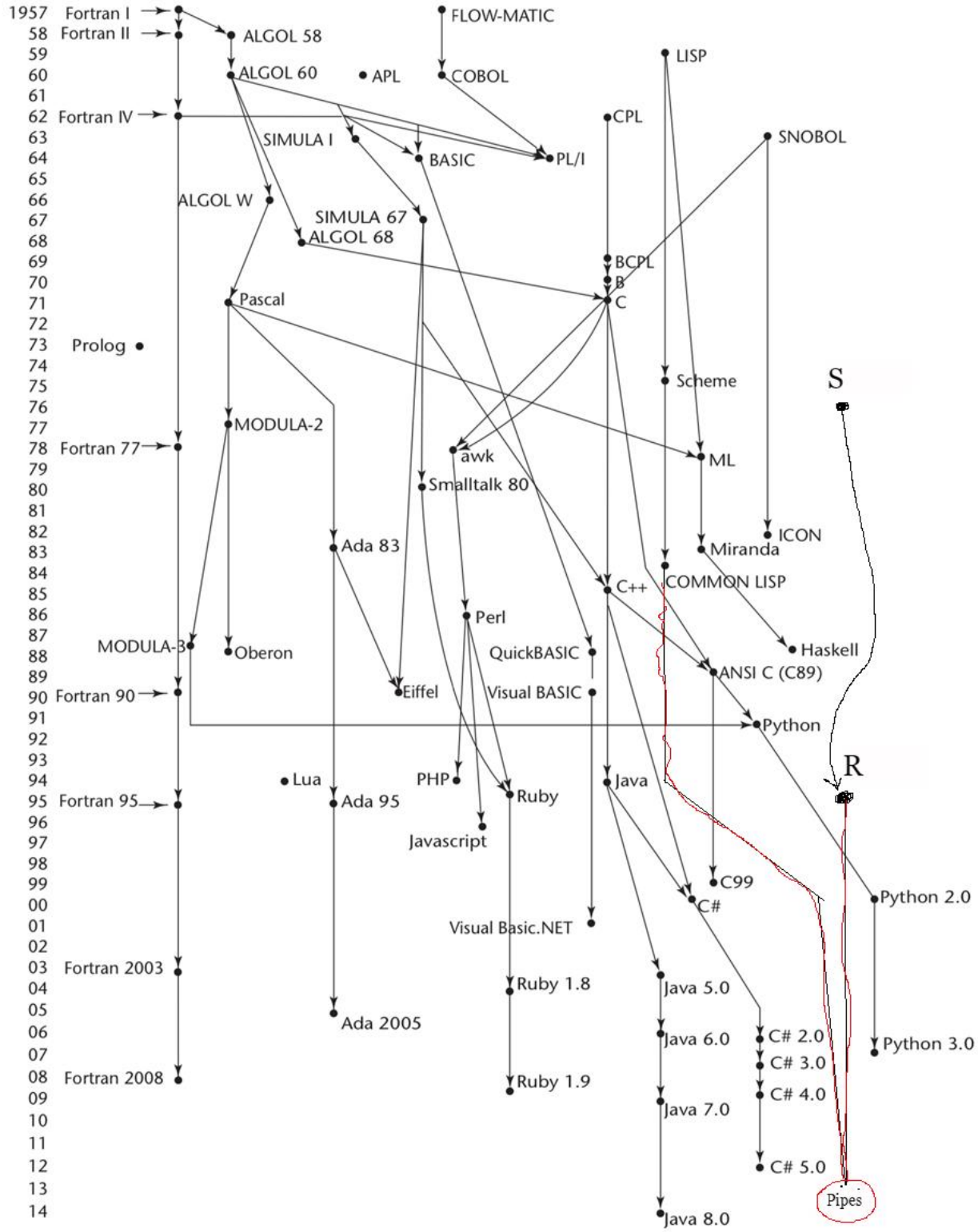
Version 0.10.1

1.Introduction

Pipes (pronounced as you would expect) is a simple, modern, and functional programming language. Based on Lisp and R, but differing in the following ways:

1. The main approach to writing expressions and statements are through the use of pipes, creating pipelines for data flow
2. Provides many helper functions out-of-the-box to assist in rapid data refinement
3. Contain first-class graphing and charting functionality, with an extensive API for Open-Source developers to contribute to the condu.it package manager. Developers will be able to manage third party libraries and tools using the “condu-cli”

1.1.Genealogy



1.2.Hello world

"Hello World" | out

1.3.Program structure

The key organizational concepts in Pipes are as follows:

1. Pipes: The program is a comprised of pipes
2. Refiners: Takes a single element in the pipeline in at a time and returns a single element
3. Reducers: Takes a set of elements in the pipeline and returns a single element back
4. Forks: Takes a single element and returns multiple elements back
5. Exits: Anything that terminates a pipeline, stdout, graphs, charts, assignment, etc.

This example:

```
// The first parameter of a refiner is always passed as the element that needs to be refined
ref doubleAndAddX(num, x) {
	num | * 2 | + x | return
}

seq(0,5) |
doubleAndAddX(5) |
fork { // A fork will return the first element to call return, if there are multiple returns
	out
	doubleAndAddX(-7) | return
} |
out
```

declares a refiner named doubleAndAddX, and begins the pipeline on a sequence of numbers from 0 to 5, exclusive. A refiner takes each of the numbers in the pipeline and "refines" it, doubling it and adding 5 in the first call. After that, the program forks, effectively creating two pipelines, which run concurrently. One pipeline terminates immediately in an out statement, and the other calls doubleAndAddX with the argument -7, doubling the result of the previous pipe and subtracting 7. This is then returned to the fork, which it then terminated in yet another out. The output of this program is

5, 7, 9, 11, 13

3, 7, 11, 15, 19

1.4.Types and Variables

All data in Pipes are treated as reference types. This is because of the way programs were intended to be created. Data is modified in place as it flows through the pipeline. This is both an optimization, and an opinionated way of enforcing a common style among developers in the Pipes ecosystem.

1.5.Statements Differing from Lisp and R

Statement	**Example**

Expression statement	`[1,2,3,4,5] | $data`
	
	`data |`
	`+ 2 |`
	`* 8 |`
	`out`

`if` statement	`[1,2,3,4,5] | $data`
	
	`data |`
	`% 2 |`
	`fork { //notice the discontinuation of the pipe`
	`  == 1 | return "odd"`
	`  return "even"`
	`} |`
	`out`

Assignment operator	//Use $ to assign a new variable to allocate memory for
	
	`[1,2,3,4,5] | $data`
	
	`Data | $x`
	
	`x | out`
	
	`//prints contents of x which are the same as data`

2.Lexical structure

2.1.Programs

A Pipes ***flow*** consists of one or more ***source files***. A source file is an ordered sequence of (*definitely* Unicode) characters.

Conceptually speaking, a program is interpreted using three steps:

1. First, allocates memory for any refiners
2. Runs each flow on the outermost level in parallel
3. Each flow awaits the completion of the previous pipeline step.

2.2.Grammars

This specification presents the syntax of the Pipes programming language where it differs from Lisp and R.

2.2.1.Lexical grammar (tokens) where different from Lisp and R

<literal> = '.' or \d**

<fork> = { . }*

*<variable> = \D.**

2.2.2.Syntactic (parse") grammar where different from Lisp and R

Flow ::== <data> | <pipelines> | <exit>

Pipelines ::== <pipeline>

::==<pipelines> | <pipeline>

Pipeline::==<statement>

::==<fork>

Statement::== <operator> <literal>

Data ::== <literal>

::==<variable>

Exit::==out

::== $<variable>

Fork::== { <pipelines> }

2.3.Lexical analysis

2.3.1.Comments

Two forms of comments are supported: single-line comments and delimited comments. ***Single-line comments*** start with the characters `//` and extend to the end of the source line. ***Delimited comments*** start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

2.4.Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens where needed.

tokens:
identifier
keyword
number-literal
string-literal

array-literal
operator-or-punctuator

2.4.1.Keywords different from Lisp or R

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier

Keywords:

```
fork    ref    red    out    and    or    return
```

3.Type System

Pipes uses a **strongly dynamic** type system. **Strongly** typed means it will not convert between non-like data types for you, i.e. “123” to 123. **Dynamic** typing means that typing is not determined until runtime, similarly to JS however, this can be inferred by the IDE to provide helpful type hints and utilizes late binding type checks.

3.1.Type Rules

The type rules for Pipes are as follows:

You have to specify type rules regardless of whether you are using a strong or weak type system. In fact, your type rules should explicitly reflect this choice. Write type inference rules in the style found in the Scope and Type” slide deck on our web site, an example of which is given below. Write your own type rules; do not just include these verbatim. Be sure your type rules match the operators you chose for your language.

S ⊢ e1 : T1	S ⊢ e1 : T	S ⊢ e1 : Number	S ⊢ e1 : String
S ⊢ e2 : T2	S ⊢ e2 : T	S ⊢ e2 : Number	S ⊢ e2: String
---	---	---	---
S ⊢ e1 | $e2 : T2	S ⊢ e1==e2: Boolean	S ⊢ e1+e2 : Number	S ⊢ e1+e2 : String
	S ⊢ e1!=e2: Boolean	S ⊢ e1-e2 : Number	
	S ⊢ e1<e2: Boolean	S ⊢ e1*e2 : Number	
	S ⊢ e1>e2: Boolean	S ⊢ e1/e2 : Number	

Pipes types are all reference types to encourage a uniform code style. If otherwise needed you can use the copy() in a pipeline.

4.Example Programs

4.1 Caesar Encrypt and Decrypt

```
ref shift(char, amount){
	char |
	upper |
	ord |
	fork {
		>=65 and <= 90 |
		- 65 |
		+ amount |
		% 26 |
		+ 65
		chr |
		return
		// else
		chr | return
	}
}

ref encrypt(word, amount) {
	word |
	explode | // explode is an example of a fork
	shift(amount)  |
	implode | // implode is an example of a reducer
	return
}

ref decrypt(word, amount) {
	encrypt(word, 26 – amount) |
	return
}

“Hello World” | $plaintext
3 | $shiftAmt

encrypt(plaintext, shiftAmt) |
out
```

4.2 Factorial

```
ref factorial(num) {
	num  |
	fork {
		<= 1 | 1 | return
		factorial(num - 1) |
		* num |
		return
	}
}

seq(0, 10) |
factorial |
out

//Prints [1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880]
```

4.3 Bubble Sort

```
ref swap(i, j) {
	I | $tmp
	j | i
	tmp | j
}

red bubblesort(arr) {
	arr |
	indexes |
	fork {
		[arr[.], [arr[.+1]] | swap(., .+1) | return
		return
	} |
	return arr
}

[4,2,6,7,1] |
wrap | //wrap the array to pass it as a single element to the reducer function
bubblesort |
unwrap |
out
```

4.4 Text Adventure Game

```
In("What's your name?") |
"Hello " + . |
fork {
        Out
        In("You are in a dark room. What do you do?") | lowercase | return
} |
fork {
        contains ("flashlight") | fork {
                "Good idea" | out
                in("you see a door, now what?")| lowercase | return
        } |
        fork {
                contains("turn knob") | fork {
                        "Very nice" | out
                        In("You open the door and a stranger approaches") | lowercase |
return
                }
                contains("kick") | fork {
                                        "Very nice" | out
                                        In("You kick the door and the stranger is angry") |
                lowercase | return
                }
                contains("turn back") | "Are you scared? You lose quitter…" | out
        }
        contains("torch") | fork {
                "The torch burnt out" | out
                in("Now what?")| lowercase | return
        } |
        contains("wander") | fork {
                "You come across a door" | out
                in("")| lowercase | return
        }
        contains("wander") | fork {
                "You come across a door" | out
                in("")| lowercase | return
        }
}
```

4.5 Fibonacci

```
ref fib(num) {
	num  |
	fork {
		< 2  | num | return
		fib(num – 1) |
		+ fib(num – 2) |
		return
	}
}

8 |
fib |
out

//Prints 21
```