Foundations and Limitations of Jacinda

Vanessa McHale

May 23, 2022

Contents

0.1 Introduction

Jacinda is an expression-oriented, functional language that performs a subset of Awk's function.

In many ways it is unsatisfactory or even defective, but it is useful and so I present the implementation so that one can

0.2 Core Approach

In order to stand toe-to-toe with Awk, Jacinda has some special facilities to work with streams filtered via regular expressions.

To define a stream from a pattern: `{%/Bloom/}{‘0}`. `‘0` is a line expression; we could write

0.3 Type System

A signal defect of the type system is that it fails to distinguish expressions in the context of a line from expressions in general. In fact expressions in the context of a line form a monad but are not treated as such because explicit typing would be too burdensome and I did not know how to implement implicit coercions.

Consider:

```
[y]|> {|‘0~/^$/}
```

This evaluates whether the last line is blank.

In a more principled world, we would have `‘0 : Ctx Str` and we would force {| ... } to take an expression of type `Ctx a`. Then we could do something like {|(~/^$/)"‘0}... of course this is more prolix.

0.3.1 Implementation

The implementation uses constraint-based typing, as in my Kempe compiler [1].

Due to various defects in my implementation, polymorphic values can only be instantiated with one type;

```
let val sum := [(+)|0 x]
  in sum {|sum (let val l := splitc ‘0 ’ ’ in l:i end)} end
```

is not accepted because `sum` has type `List Int -> Int` at one site and `Stream Int -> Int` at another.

0.3.2 Static Typing

Static typing catches errors before runtime; in particular it gives feedback in one's text editor.

Types as Documentation

The manpages specify type signatures, e.g.

```
\. Binary operator: prior
        (a -> a -> b) -> Stream a -> Stream b
```

This is more concise and less ambiguous than Awk documentation.

0.3.3 Typeclasses

Type classes are used to witness implementations [2]. Type assignment information is used in the interpreter;

0.4 Rewriting Folds

Often, it is most sensible to express programs in terms of several folds, e.g.

```
let
  val tot := (+)|0.0 $1:
  val n := (+)|0.0 [:1.0"$0
in tot%n end
```

We would like to pass over the file contents only once; thus we need to

Bibliography

[1] V. E. McHale. A compiler for a stack-based language. `http://vmchale.com/original/compiler.pdf`, 2021.

[2] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery.