Step-by-step type inference

A tool to better understand Hindley-Milner type systems through a Haskell interpreter

Luchino Allix-Lastrego

Université de Montréal

November 14th, 2026





Table of contents

- Context
- Motivation
  - codeBoot
  - Haskell
  - Haskell codeBoot
- Implementation
  - Implementation choices
  - Skipped steps
- Evaluation
- Questions

Context

Context

- Research for my master's degree
- Supervised by Stefan Monnier
- A tool to be used in the programming languages course

Motivation

Motivation

Problem

Type-related error messages are notoriously difficult to understand in type systems.

Not unique to Haskell, but in C++ or Rust as well, particularly languages that allow parametric polymorphism (= generics/templates).

Problem of intention.

```
lastElem [] = []
lastElem [a] = a
lastElem (a:b) = lastElem b

main = print (lastElem [1, 2, 3, 4, 5])
```

```
lastElem [] = []
lastElem [a] = a
lastElem (a:b) = lastElem b

main = print (lastElem [1, 2, 3, 4, 5])
```



```
exemple.hs:5:25: error: [GHC-39999]
 - No instance for 'Num [a0]' arising from the literal '1'
 - In the expression: 1
   In the first argument of 'lastElem', namely '[1, 2, 3, 4,
     ....]'
   In the first argument of 'print', namely
       '(lastElem [1, 2, 3, 4, ....])'
  |
5 | main = print (lastElem [1,2,3,4,5])
  |                        ^
```

Solution

Create a tool that explains how type inference works, step by step, just like a debugger does for evaluation.

Solution

Create a tool that explains how type inference works, step by step, just like a debugger does for evaluation.

This is one solution among many :

- Better error messages
- Visually show the type relations between elements
- Show different modifications upon a given code to correct it
- ...

codeBoot

codeBoot is a web-based IDE developed at UdeM by Marc Feeley to teach **introductory programming**.

Key features

- Python interpreter (subset)
- REPL
- Fully client-side
- Step-by-step evaluation
- Integrated file system
- Forced version

Haskell

- General-purpose
- Statically typed
- Purely functional
- Parametric polymorphism
- Lazy evaluation
- Algebraic data types

Haskell codeBoot

Like codeBoot, but for Haskell:

Haskell codeBoot

Like codeBoot, but for Haskell:

Haskell subset interpreter

Haskell without type classes, list comprehensions, `where`, monads other than `IO`, or lazy evaluation ...

Haskell codeBoot

Like codeBoot, but for Haskell:

Haskell subset interpreter

Haskell without type classes, list comprehensions, `where`, monads other than `IO`, or lazy evaluation ...

Step-by-step type checker

Like a debugger, but for type checking, showing the context and how each piece of information is learned.

Example: the type of *map*

How do you explain that the function *map*:

```
map f []       = []
map f (h : r) = (f h) : (map f r)
```

has type $\forall\, \alpha, \beta \ :\ (\alpha \to \beta) \to [\alpha] \to [\beta]$?

Demo

Skipped steps

- Many steps are not shown (more detail in the implementation)
- They don't help understand
- About 9 times less than naive approach

Implementation

Implementation choices

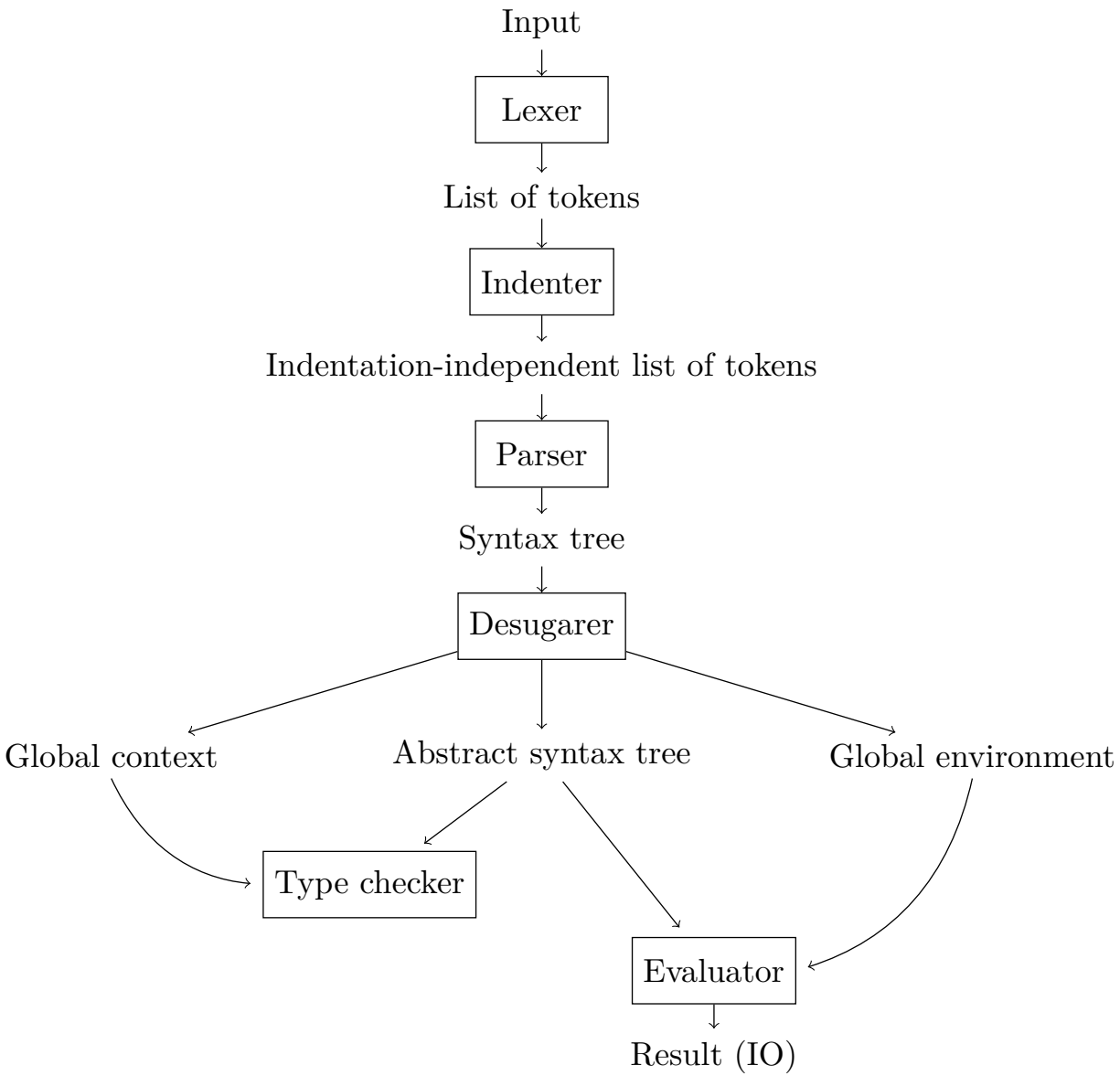
JavaScript

- Easier integration into codeBoot
- No compilation step ⇒ easier debugging (web browser)

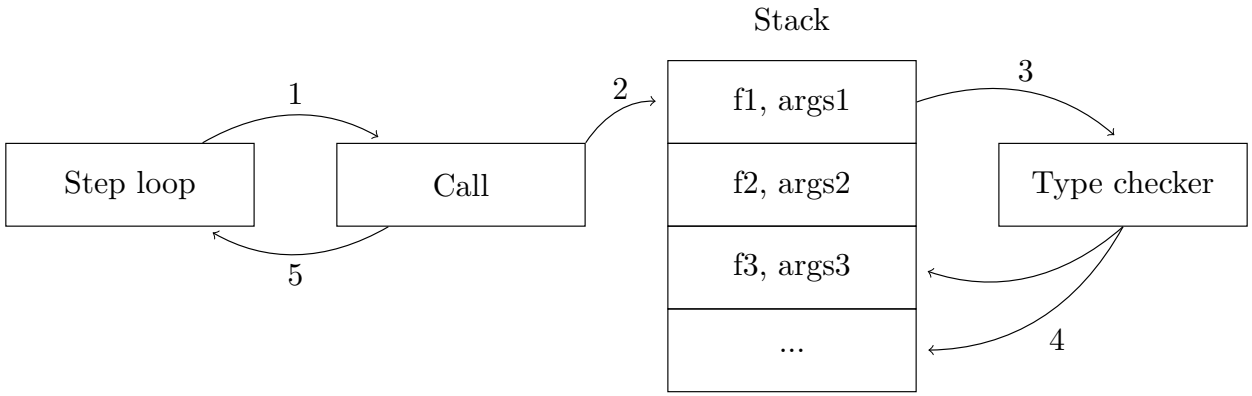
Manual implementation

- Lexer, parser, type checker and evaluator
- No external tools ⇒ easier debugging and no adaptation required

Pipeline


Input
Lexer
List of tokens
Indenter
Indentation-independent list of tokens
Parser
Syntax tree
Desugarer
Global context
Abstract syntax tree
Global environment
Type checker
Evaluator
Result (IO)

CPS-like type checker



CPS (continuation-passing style) makes it possible to mark certain steps as skippable.

Skipped steps

```
sum x = x + 1

res = sum 41
```

- Success / failure steps (the outcome is already certain)

  $\Rightarrow$ *e.g. occurs-check*

Skipped steps

```
sum x = x + 1

res = sum 41
```

- Success / failure steps (the outcome is already certain)
  $\Rightarrow$ *e.g. occurs-check*

- Redundant information
  $\Rightarrow$ *e.g. unification between identical types*

Skipped steps

```
sum x = x + 1

res = sum 41
```

- Success / failure steps (the outcome is already certain)

  $\Rightarrow$ *e.g. occurs-check*

- Redundant information

  $\Rightarrow$ *e.g. unification between identical types*

- Trivial steps (required by the type checker, but easy to understand)

  $\Rightarrow$ *e.g. specialization of immediate values*

- Superfluous steps (complex, but they don't aid understanding)
  $\Rightarrow$ *e.g. generalization of functions*

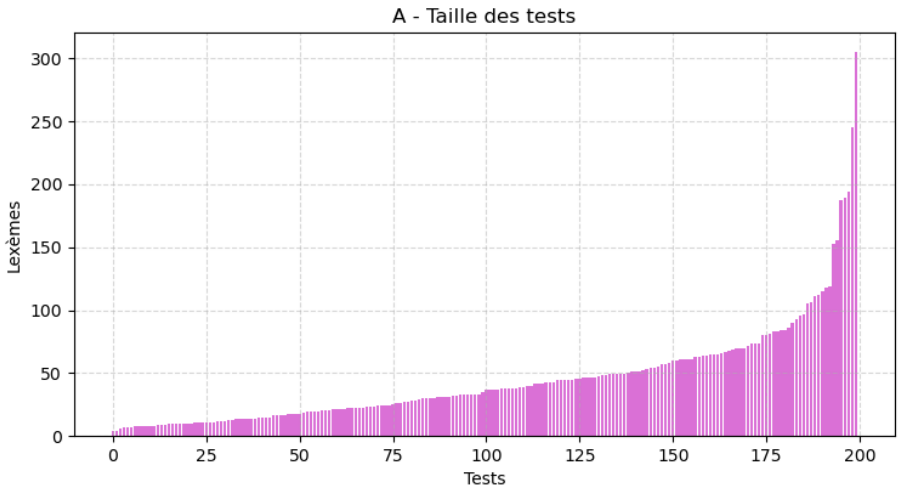
```
map f []       = []
map f (h : r) = (f h) : (map f r)
```

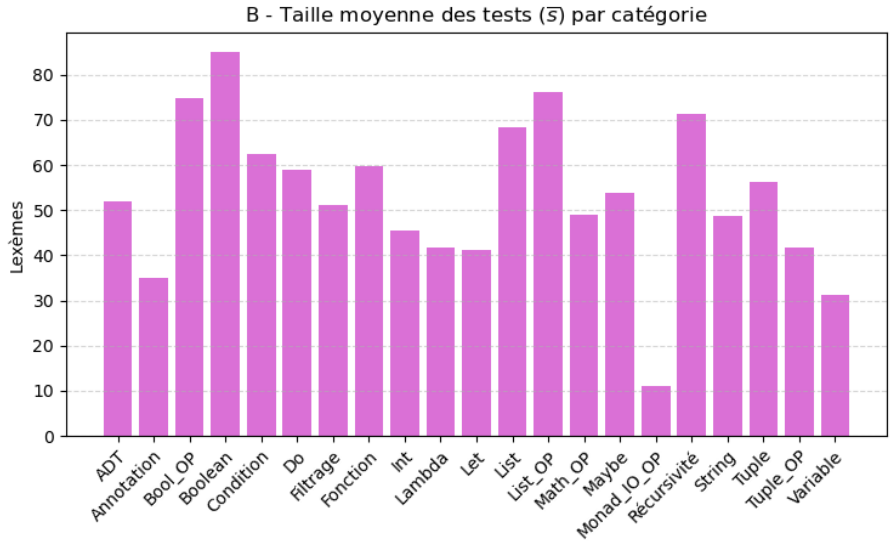
$(t1 \to t2) \to [t1] \to [t2]$

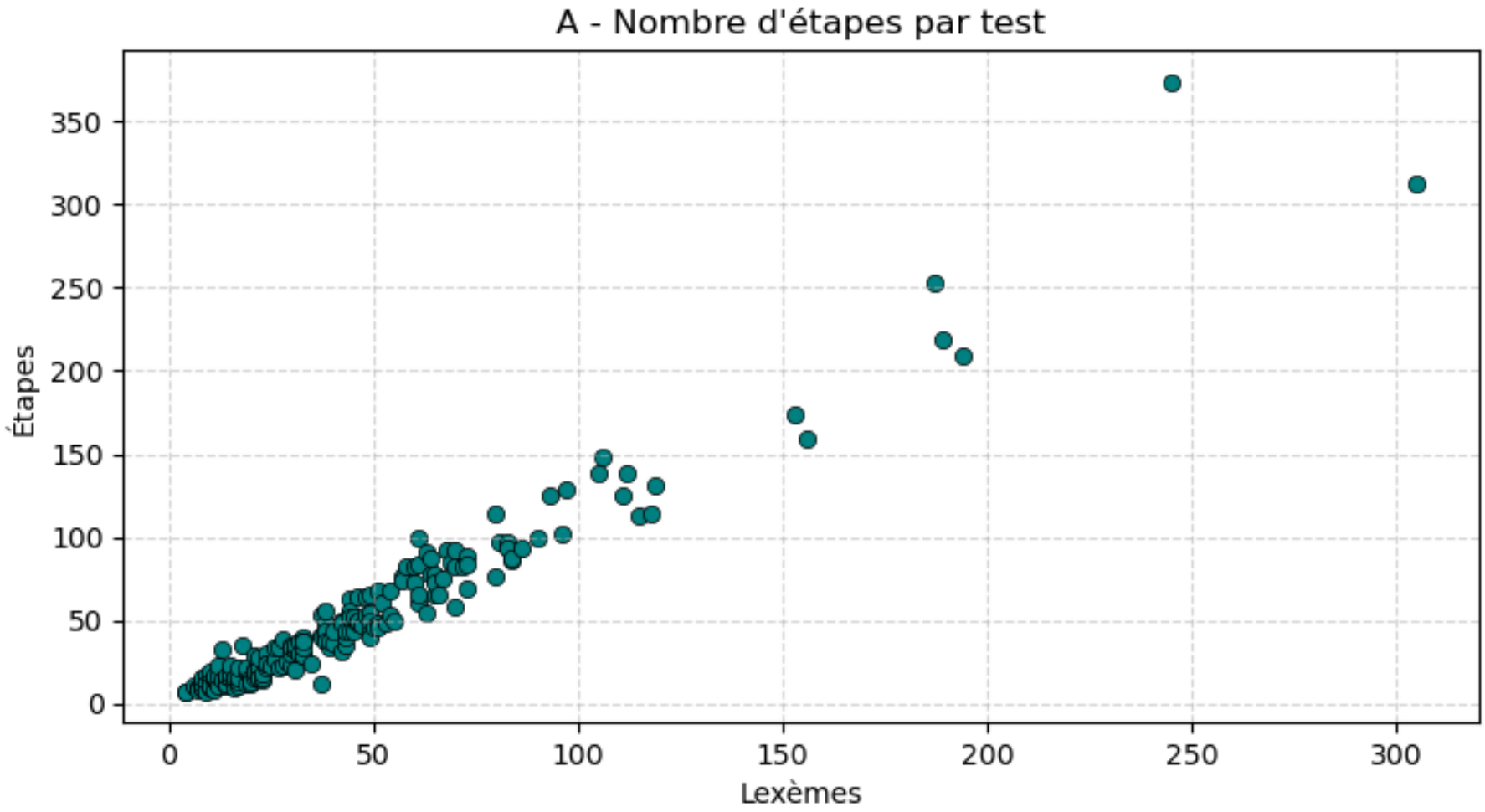
$\forall\, \alpha, \beta \;:\; (\alpha \to \beta) \to [\alpha] \to [\beta]$

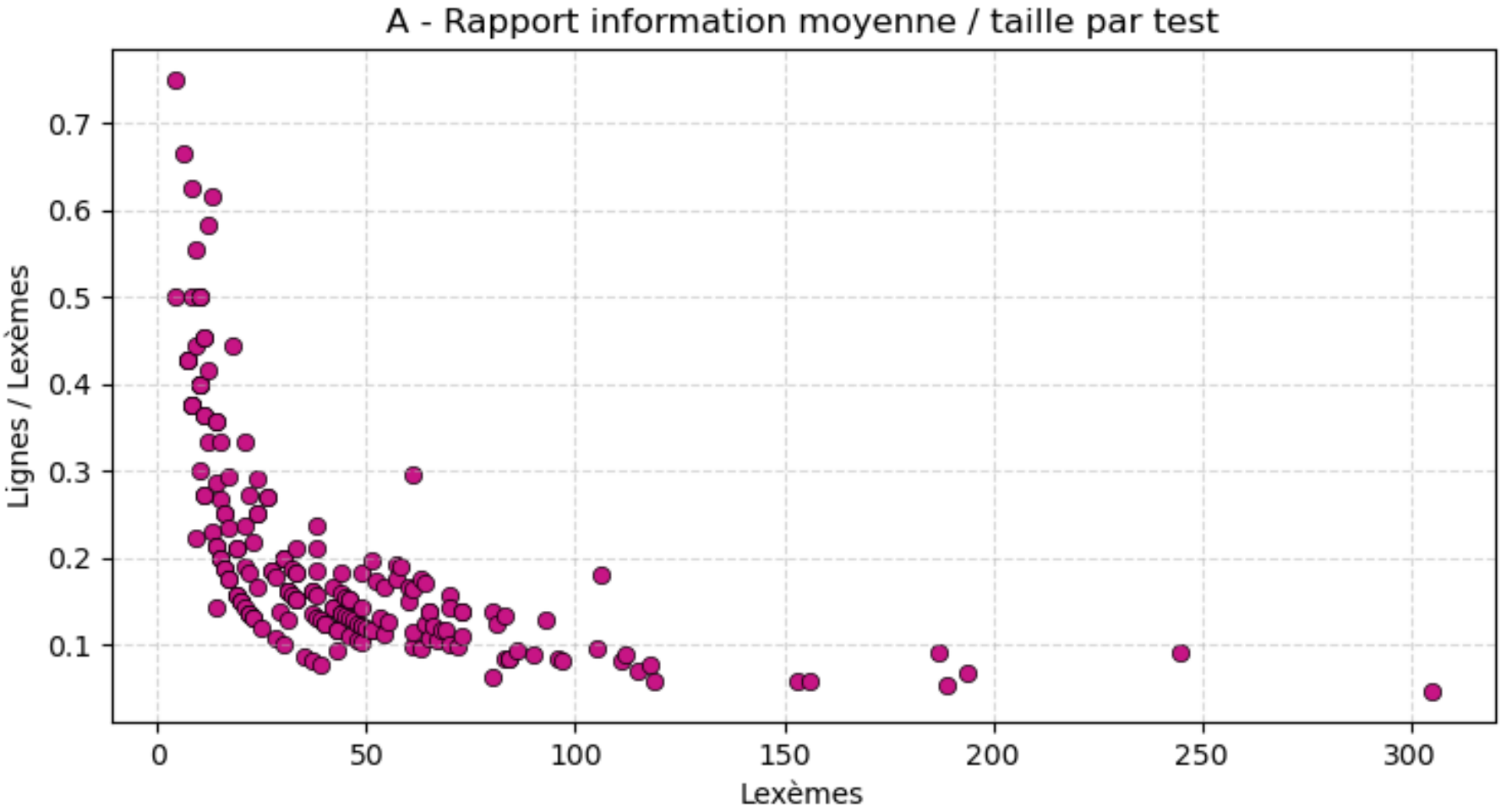
Evaluation

Evaluation: tests






A - Nombre d'étapes par test
Étapes
Lexèmes
0
50
100
150
200
250
300
350


A - Rapport information moyenne / taille par test
Lignes / Lexèmes
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
50
100
150
200
250
300
Lexèmes

Conclusion

Strengths

- Linear to program size
- Uniform amount of information per step
- No language feature requires disproportionately many steps

Conclusion

Strengths

- Linear to program size
- Uniform amount of information per step
- No language feature requires disproportionately many steps

Limitations

- Too many steps
- Too much information

Future work

- Cover a larger subset of Haskell
- Make it work like an actual debugger (showing steps that lead to failure)
- Back-stepping
- Step-by-step evaluation in Haskell codeBoot
- Choose what steps are shown
- A case study

Questions ?

Links



codeBoot
→ `https://codeboot.org`



Haskell codeBoot
→ `https://luchinoallix.github.io/`



Thesis (in French)
→ `https://umontreal.scholaris.ca/items/...`