Type Debugging with Counter-Factual Type Error Messages Using an Existing Type-Checker

Kanae Tsushima
National Institute of Informatics
Japan
k_tsushima@nii.ac.jp

Olaf Chitil
University of Kent
UK
oc@kent.ac.uk

Joanna Sharrad
University of Kent
UK
jks31@kent.ac.uk

Example

Applies a function to each element of a list in OCaml

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

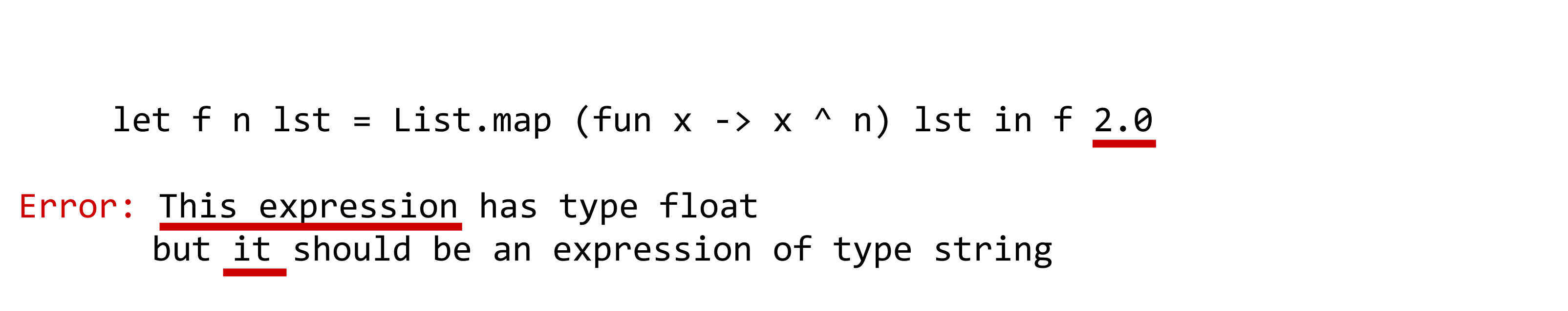
Example

This code is ill-typed:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

Example

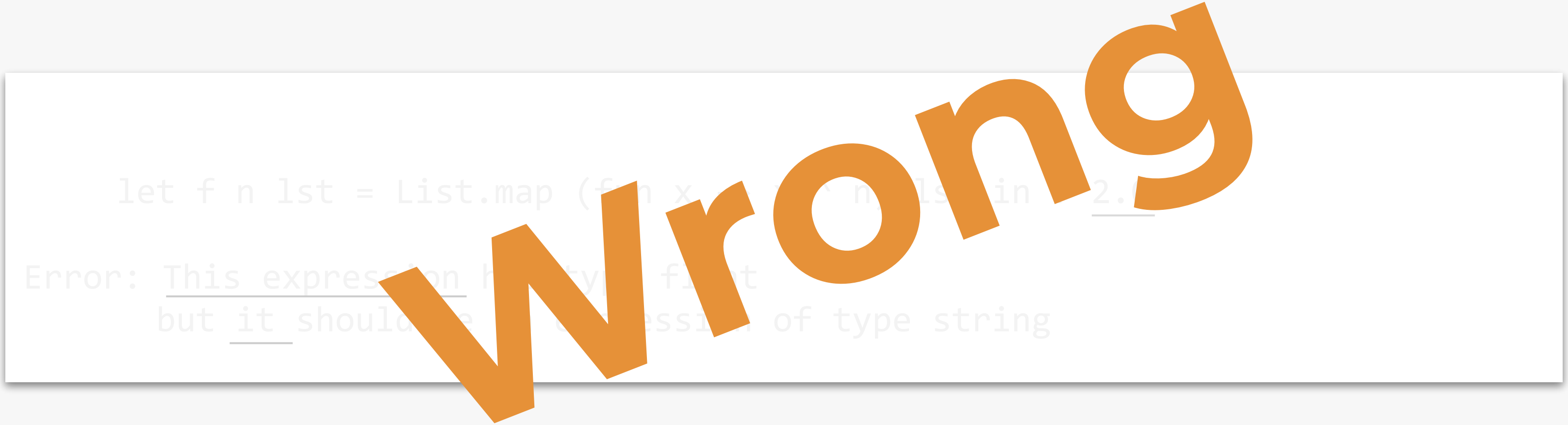
This code is ill-typed and returns a counter-factual message:



```
    let f n lst = List.map (fun x -> x ^ n) lst in f 2.0

Error: This expression has type float
       but it should be an expression of type string
```

Example



```
  let f n lst = List.map (fun x [illegible] ^ n) lst in 2.[illegible]

Error: This expression has type float
       but it should be an expression of type string
```

Example

This code is still ill-typed:

```
let f n lst = List.map (fun x -> x ^ n) lst in f 2.0
```

Example

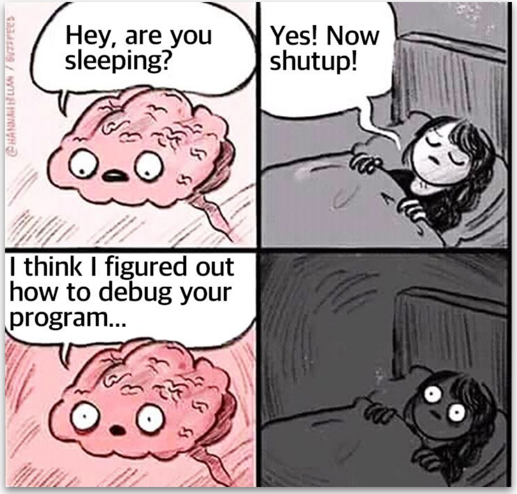
The error we need:

```
     let f n lst = List.map (fun x -> x ^ n) lst in f 2.0

Error: This expression has type string -> string -> string
       but it should be an expression of type ‘a -> float -> ‘b
```

What is the issue?

We all experience type errors, causing hours of frustration…



What is the issue?

We all experience type errors, causing hours of frustration…

...made worse as the compiler doesn’t know your intent!

BUT THE ERROR IS BECAUSE OF LINE 20, NOT 50!
DO YOU REALLY THINK THAT YOU'RE SMARTER THAN A COMPILER?

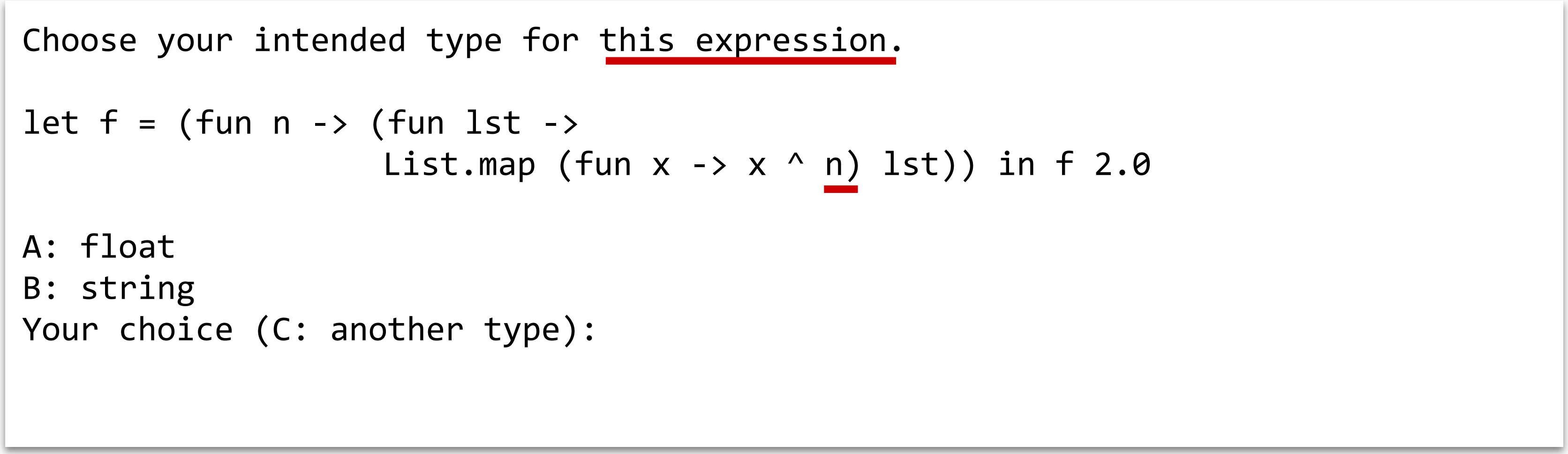
An illustration of our method

Four Potential Counter-Factual Types:

```
let f = (fun n -> (fun lst ->
                   List.map (fun x -> x ^ n) lst)) in f 2.0
```

An illustration of our method

Three choices:



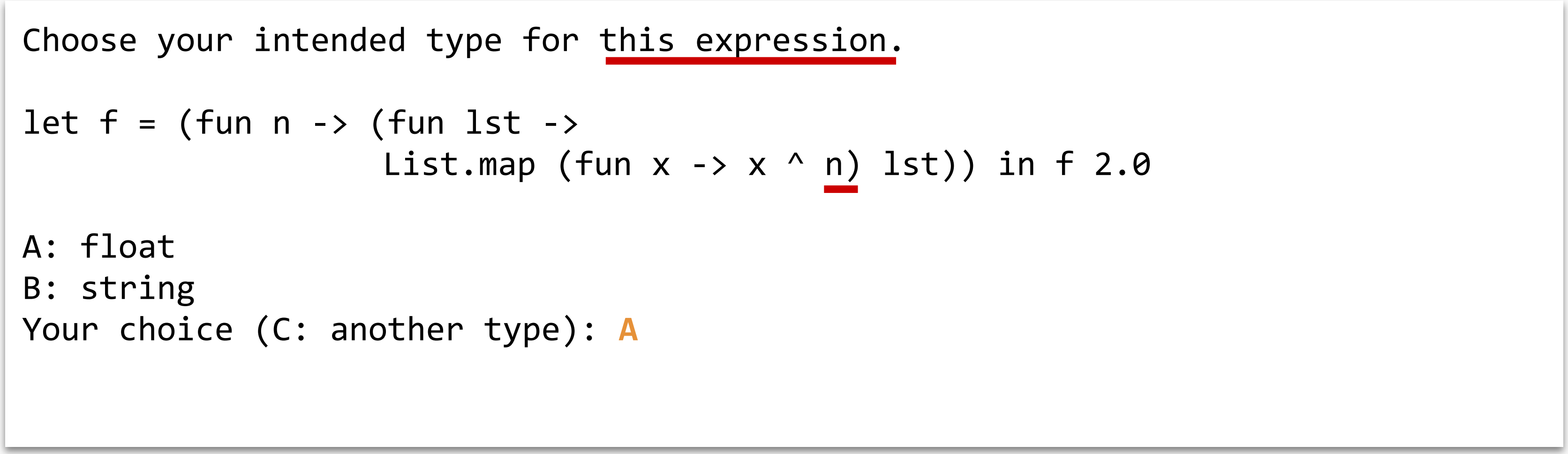
```
Choose your intended type for this expression.

let f = (fun n -> (fun lst ->
                   List.map (fun x -> x ^ n) lst)) in f 2.0

A: float
B: string
Your choice (C: another type):
```

An illustration of our method

Choice A:



```
Choose your intended type for this expression.

let f = (fun n -> (fun lst ->
                   List.map (fun x -> x ^ n) lst)) in f 2.0

A: float
B: string
Your choice (C: another type): A
```

An illustration of our method

Choice A - Program Annotation:

```
let f = (fun n -> (fun lst ->
                   List.map (fun x -> x ^ (n:float) lst))in f 2.0
```

An illustration of our method

Debugger Result:

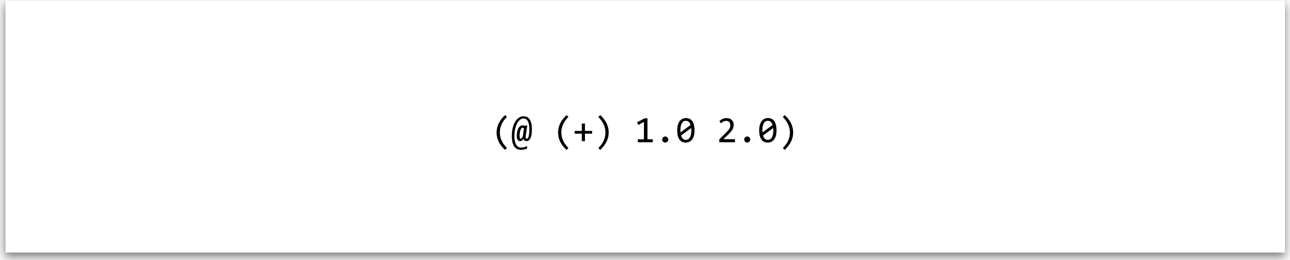
```
let f = (fun n -> (fun lst ->
                   List.map (fun x -> x ^ n) lst)) in f 2.0

Error: This expression has type  string -> string -> string
  but it should be an expression of type ‘a -> float -> ‘b
```

Blackbox Type Checker

Reuse of the existing type checker

- Only to see if a program is well-typed or ill-typed
- Well-typed we gain the type. Ill-Typed no more information
- No use of the type-checkers own error messages

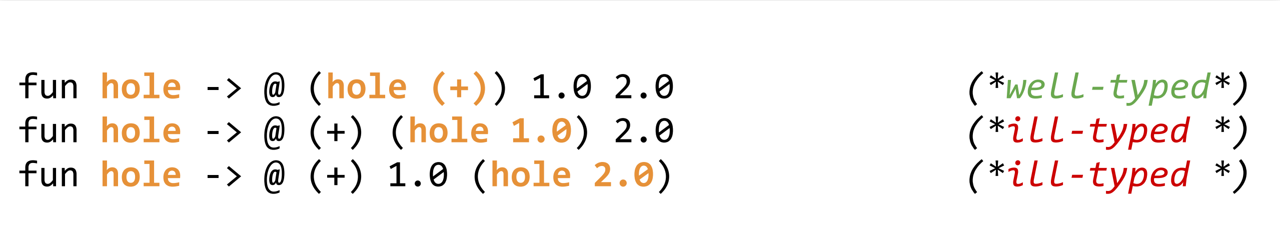


```
(@ (+) 1.0 2.0)
```

Blackbox Type Checker

Reuse of the existing type checker

- Called on variations of the program with holes
- Well-typed gives us the type:
  - (int -> int -> int)



```
fun hole -> @ (hole (+)) 1.0 2.0            (*well-typed*)
fun hole -> @ (+) (hole 1.0) 2.0            (*ill-typed *)
fun hole -> @ (+) 1.0 (hole 2.0)            (*ill-typed *)
```

Counter-Factual Types

When the expected and actual type differ.

- Replacing each leaf or leaf application with a hole
- The existing type checker is used to obtain types
- Using type error slicing for efficiency

```
1. + 2.
Error: This expression has type
     int -> int -> int                        (* expected *)
But it should be an expression of type
     float -> float -> ‘a                     (* actual   *)
```

Programmers Intent

Do what I say not what I do...

- Ask questions that relate to the code
- Make them easy to understand
- Influence the debugger’s next move

```
A: string -> string -> string
B: string -> float -> 'a
Your choice (C: another type): float -> float -> float
```

Evaluation

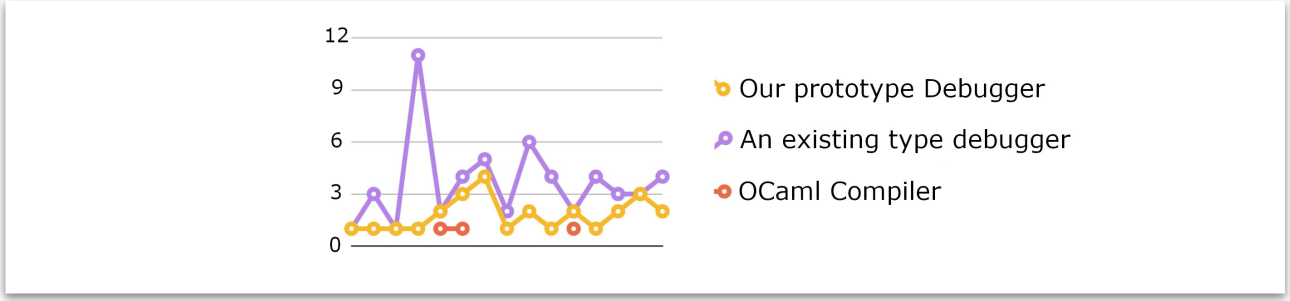
The inaccurate reporting of the location of a type error

- Compared to OCaml and an existing interactive debugger
- 15 ill-typed programs from two sources:
  - Student programs from an introductory course
  - Test code from an online demonstration
- 6 programs generated to test runtime

Comparison to the existing

How many questions to get to the correct location?

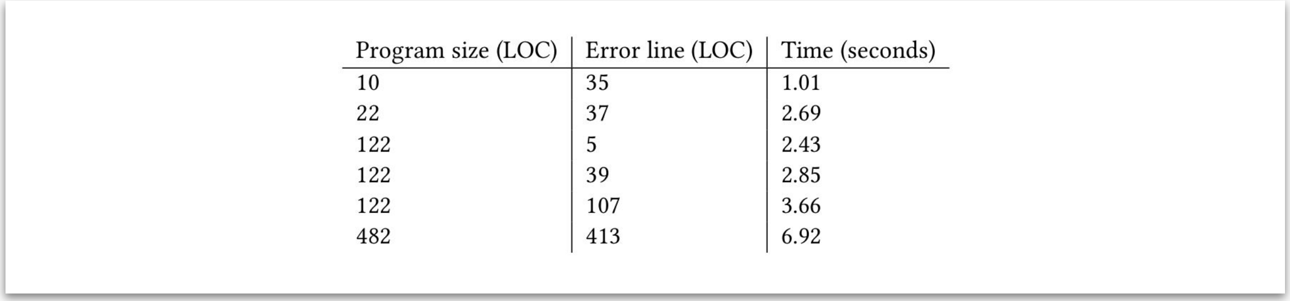
- **100%** location success over **20%** from OCaml
- **30%** fewer questions than the existing debugger



Runtime Costs

Is runtime reasonable for interactive use?

- Works best for programs up to **100** lines of code
- **3.2** seconds average for location discovery



Program size (LOC)	Error line (LOC)	Time (seconds)
10	35	1.01
22	37	2.69
122	5	2.43
122	39	2.85
122	107	3.66
482	413	6.92

Future Work

- Develop heuristics on question locations
- Evaluate on a larger set of ill-typed programs.
- Replace Slicing with Delta Debugging

Thank You

- An Interactive Type Debugger which:
  - Uses ‘holes’ with an existing type checker
  - Has Slicing to increase efficiency
  - Asks the programmers intention.
  - Provides Counter-Factual Error Messages