If It Type-checks, It Works: FoolProof Types As Specifications

Brandon Wu, bjwu@andrew.cmu.edu
Carnegie Mellon University

March 27, 2021

Abstract

Proponents of functional programming languages often espouse the phrase "If it type-checks, it works" to describe their code. While a strong type system can prevent a great deal of run-time errors, this statement is often a hyperbole, and not predicated on any factual basis. In this paper, we will explore a toy functional programming language called FoolProof that utilizes a rich type system to provably ensure the program's correctness upon type-checking.

1 Introduction

"If it type-checks, it works!" *Functional Programming* enthusiasts often parrot[1] this phrase, attempting to convey the idea that a strong type system can help to guarantee correctness at runtime. While this is a point that has some basis in reality, it is not an altogether truthful one. In many programming languages, types are a fantastic way to rule out many classes of potential errors, but they fail to encode the entire specification of the program, making *False Parroters* out of our FP enthusiasts.

In this paper, we will discuss the design of a groundbreaking new language[2] that can give truthful credence to the eponymous claim.

2 Motivation

Types encode a great deal of information about a program's behavior at runtime. For instance, with certain kinds of type systems, programming language designers can rule out race conditions[3], ensure the restriction of certain kinds

[1] There appears to be an odd relationship between parrots and functional programming. Perhaps the basis for another paper.

[2] And humble, too!

[3] !!!!!

of resources, and create programs that are capable of acting generically over different varieties of data.

A stellar example of this phenomenon comes from Wadler's seminal paper[4] [1], which proves that a total function of type $\forall\tau.\tau \to \tau$ can only be the identity function, which is a shining example of how types serve as *specifications*, giving us information over how a program may behave at run-time. Put most simply, for instance, a program of type `int` computes an integer, a program of type `'a list -> 'a list` can only permute and manipulate the elements of a given list, and a program of type `'e list -> ('e list -> ('l -> 'a) -> (unit -> 'a) -> 'a) -> ('e list -> ('r -> 'a) -> (unit -> 'a) -> 'a) -> 'l * 'r -> 'a) -> (unit -> 'a) -> 'a` means you are enrolled in 15-150.

This is not a sufficient impetus to justify the claim, however. While it is true that a total function of type `'a list -> 'a list` can only return a list containing the same elements that it was given, this still describes a wide variety of programs! For instance, consider the following code:

```
(* twice : 'a list -> 'a list *) [5]
fun twice [] = []
  | twice (x::xs) = x :: x :: twice xs
(* thrice : 'a list -> 'a list *)
fun thrice [] = []
  | thrice (x::xs) = x :: x :: x :: thrice xs
```

Both of these functions have the same type signature, and yet exhibit provably different behavior! This causes our claim to be dead in the water. To *Formally Prove* this claim, we will need to try something else.

As Carnegie Mellon University School of Computer Science Professor Robert Harper champions, imposing restraints on programming languages only creates *more* freedom, because those restraints can be selectively relaxed at a later date [2]. It is clear that this failure is a result from having a type system which is too relaxed. Thus, we must accordingly search for a stronger type system.

This is not to say that the ML type system is entirely hopeless - for instance, types such as `unit` and `'a -> 'a` provably can only have the behavior of a program which returns `()`, and a program which acts as the identity function[6], respectively. There is no choice here – for these types, the behavior of the program is deterministically decided by its type. Up to extensional equivalence, we can regard both of these types as having only a *single inhabitant*. We will seek to emulate this behavior, in search of the perfect type system, by defining the language FoolProof.

[4] Discussing the concept of *Fancy Polymorphism*

[5] Not to be confused with the Korean girl group.

[6] The astute totality citer will note that a function of the aforementioned types may also loop forever. Since determining whether or not a program halts is undecidable, we will therefore assume that it does halt, and laugh while you try to prove us wrong.

3 FoolProof

As specified before, in order to seek the *Forevermore Paradise* of programming language nirvana, we must create a language which is as restricted as possible. Types supposedly serve as specifications, and yet in the ML type system, our type system is not strong enough to ensure the extensional behavior of all programs of a specified type. We will take this approach one level higher, then, and design a language where the types themselves document the exact behavior of their programs.

We will now define the language of FoolProof, with statics and dynamics as follows:

Statics:

$$\frac{}{\Gamma, x : x \vdash x : x}\ VAR$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \texttt{if}\ \tau_1\ \texttt{then}\ \tau_2\ \texttt{else}\ \tau_3}\ ITE$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{true}}\ TRUE \quad \frac{}{\Gamma \vdash \texttt{false} : \texttt{false}}\ FALSE$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{z}}\ ZERO \quad \frac{\Gamma \vdash n : \tau}{\Gamma \vdash \texttt{S}(n) : \texttt{S}(\tau)}\ SUCC \quad \frac{}{\Gamma \vdash () : ()}\ UNIT$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \tau_x \vdash e_1 : \tau_1)}{\Gamma \vdash \texttt{rec}(e; e_0; x.e_1) : \texttt{rec}(\tau; \tau_0; \tau_x.\tau_1}\ REC$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \langle \tau_1, \tau_2 \rangle}\ TUPLE$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \cdot 1 : \tau \cdot 1}\ PROJ_L \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \cdot 2 : \tau \cdot 2}\ PROJ_R$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_1 \tau_2}\ APP \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \lambda \tau_1.\tau_2}\ LAM$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash 1 \cdot e : 1 \cdot \tau}\ IN_L \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash 2 \cdot e : 2 \cdot \tau}\ IN_R$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau_x \vdash e_1 : \tau_1 \quad \Gamma, y : \tau_y \ \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ 1 \cdot x \hookrightarrow e_1 |\ 2 \cdot y \hookrightarrow e_2 : \texttt{case}\ \tau\ \texttt{of}\ 1 \cdot \tau_x \hookrightarrow \tau_1 |\ 2 \cdot \tau_y \hookrightarrow \tau_2}\ CASE$$

Dynamics:

$$\frac{}{\langle e_1, e_2\rangle \cdot 1 \longmapsto e_1}\ STEP_1$$

$$\frac{}{\langle e_1, e_2\rangle \cdot 2 \longmapsto e_2}\ STEP_2$$

The rest of the dynamics were abducted by 312 students for their homework.

4 Theory

To substantiate our claim, there are a few theorems that we can prove. In the spirit of Harper (2021) [3], we will do so in a method of *discovering* the proof, via trying a few naive methods until finally pushing the theorem through.

Theorem 1 $\boxed{Preservation}$*: If* $\Gamma \vdash e : \tau$ *and* $e \longmapsto e'$*, then* $\Gamma \vdash e' : \tau$*.*

We proceed via induction on dynamics.

- Case 1: $ZERO$
  No dynamics apply, so this case is vacuous.
- Case 2: $STEP_1$
  Uhhh...
  OK, wait, I think I might need a lemma.

Theorem 2 $\boxed{: \cong =}$*: The typing relation is the identity relation on terms.*

We proceed via induction on statics.
Base case: `refl`
Inductive case: `refl`

Theorem 3 $\boxed{Preservation}$*: If* $\Gamma \vdash e : \tau$ *and* $e \longmapsto e'$*, then* $\Gamma \vdash e' : \tau$*.*

We proceed via induction on dynamics.

- Case 1: $ZERO$
  No dynamics apply, so this case is vacuous.
- Case 2: $STEP_1$
  No, that didn't seem to help.

Theorem 4 $\boxed{\neg Preservation}$ *(alt.* False Preservation*): Preservation is not true.*

Given Theorem 2, preservation just isn't true, man.

In summary, FoolProof is a language that assigns each term a type which is equivalent to the original term. By this pleasingly restrictive type system, we ensure that the typing relation is bijective between types and terms, ensuring that, like the `unit` type, each type has only one inhabitant.

While we fail to preserve the property of preservation[7] (as the dynamics

[7] But we plunder the profits of plosive pronunciation!

regularly cause terms to step to terms with different types), we gain a great deal of predictive power. In FoolProof, types truly are specifications, as they describe the behavior of the program with extreme specificity. For instance, the behavior of only program with type $\langle$ `true, false` $\rangle \cdot 1$ is to project the first element from the tuple $\langle$ `true, false` $\rangle$, and the behavior of the only program with type `if true then true else false` is to make style graders sad.

Critics may complain that it is possible to construct programs in FoolProof which are nonsense, and do not correspond to any sensible dynamics. This is true, however the claim that we are trying to prove is "if it type-checks, it works". Clearly, any program in FoolProof type-checks, however in the same sense that no program is ever truly *wrong*, as it does what it was written to do (which may not be the programmer's intention), all programs written in FoolProof *work*. They do exactly what their type specification says that they should do. Thus, a garbage program in FoolProof such as `true false true : true false true` does exactly what its specification says it should do, namely being utter garbage.

Furthermore, via the principle of referential transparency, we obtain the following equivalences:

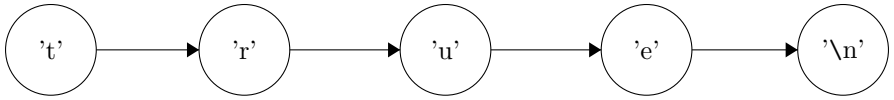
$$\begin{aligned}
\textit{types} &\cong \textit{propositions} && \text{(types-are-propositions)} \\
\textit{terms} &\cong \textit{propositions} && \text{(FoolProof types are the same as their terms)} \\
\textit{programs} &\cong \textit{propositions} && \text{(terms in FoolProof are just programs)} \\
\textit{proofs} &\cong \textit{propositions} && \text{(programs-are-proofs)}
\end{aligned}$$

Therefore, we conclude that proofs are propositions[8]. It is unclear as to what purpose this revelation serves, but *proposition-relevance* sure is a fun term.

5 Practice

Some skeptics may be concerned with the practical applications[9] of this language. We will proceed to elucidate some of these benefits.

Pedagogy Students who are first introduced to abstract syntax trees often have difficulty understanding them. To *Facilitate Pedagogy*, the author has decided on an efficient representation for FoolProof ASTs to help eager scholars learn the intricacies of this new language. We have provided a graphic of the AST representation of the term `true : true` in FoolProof via the following tree:



[8] How many props could a prop-proof prove if a prop-proof could prove props?
[9] We prefer function applications.

Compile times Type-checking in FoolProof is very fast. The author of this paper has written the following `synthtype` function in ML that takes in an AST representation of the FoolProof term, and returns its type in another efficient format. Notably, the synthtype function runs in amortized constant time[10], which is an astounding improvement over type-checking algorithms such as Hindley-Milner.

```
type AST = string
type typ = string
(* synthtype : AST -> typ *)
fun synthtype x = x
```

6 Conclusions

In conclusion, via a type system that is as restrictive as possible, we have designed a functional programming language[11] FoolProof which truly embodies the principle of "if it type-checks, it works". Functional programmers can rejoice, because with FoolProof, their unfounded claims of superiority over imperative programmers now have actual credence.

7 References

[1] Philip Wadler. 1989. Theorems for free! In Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89). Association for Computing Machinery, New York, NY, USA, 347–359. DOI:https://doi.org/10.1145/99370.99404

[2] Robert Harper, *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

[3] Robert Harper, *How to (Re)Invent Tait's Method*. Unpublished manuscript, 2021.

8 Acknowledgements

PL theorists need no acknowledgements. We need only the light of computational trinitarianism.

[10] It also runs in actual constant time.

[11] FoolProof is a somewhat functional language, and its initials are FP, so it must be a functional programming language.