# The Normalization Barrier Revisited

Shuo Ding
Georgia Institute of Technology
Atlanta, USA
sding@gatech.edu

Qirun Zhang
Georgia Institute of Technology
Atlanta, USA
qrzhang@gatech.edu

## Abstract

In a POPL 2016 paper, Brown and Palsberg presented a breakthrough result on "the normalization barrier." The normalization barrier, according to conventional wisdom, originates from a theorem in computability theory, which says that a total universal function for all total computable functions is impossible. Therefore, it was widely believed that strongly normalizing lambda calculi do not have self-interpreters either. However, Brown and Palsberg constructed a self-interpreter for F-omega, which is a strongly normalizing lambda calculus. One of the key insights behind the Brown-Palsberg breakthrough is due to the fact that "static type checking in F-omega can exclude the (computability) proof's diagonalization gadget, leaving open the possibility for a self-interpreter," according to Brown and Palsberg [2016].

In this paper, we revisit this phenomenon. In particular, in the Brown-Palsberg result, terms in F-omega were encoded as typed representations, and an external type checker was assumed to do type checking. In our work, we consider a type checker assumed to be built into the interpreter, which reports type errors on ill-typed inputs. We believe this is closer to real interpreters. Consequently, our representation is untyped, and ill-typed inputs are specifically handled. Under this setting, we show that the original computability theory result still holds. Our result does not contradict the Brown-Palsberg result. Rather, it shows that computability theory results are still applicable to F-omega from a different angle, thus "rebuilding" the normalization barrier.

***CCS Concepts:*** • **Software and its engineering** → **Interpreters**; • **Theory of computation** → **Type structures**; **Computability**.

***Keywords:*** Partial Evaluation, Program Semantics, Computability

*JENSFEST '24, October 22, 2024, Pasadena, CA, USA*
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1257-9/24/10
https://doi.org/10.1145/3694848.3694851

**ACM Reference Format:**
Shuo Ding and Qirun Zhang. 2024. The Normalization Barrier Revisited. In *Proceedings of the Workshop Dedicated to Jens Palsberg on the Occasion of His 60th Birthday (JENSFEST '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3694848.3694851

## 1 Introduction

It is well-known in computability theory that there is no computable, total, universal function for all computable total functions. The fundamental reason is that such a universal function would permit a new computable total function to be defined, which is different from every computable total function on at least one input, contradicting the assumption that the universal function can already simulate all computable total functions. As a result, conventional wisdom [9] tended to believe this phenomenon holds on lambda calculi as well, meaning that strongly normalizing lambda calculi cannot interpret themselves. This is called the "normalization barrier" [3].

In the seminal work of Brown and Palsberg [3], it was shown that F-omega, which is a strongly normalizing lambda calculus within the lambda cube [2], can indeed write self-interpreters on its typed representations. Consequently, the Brown-Palsberg result breaks the normalization barrier, showing that the computability theory result is less far-reaching than previously thought. The inherent reason for the non-applicability of the computability result, described by Brown and Palsberg, is "static type checking in F-omega can exclude the (computability) proof's diagonalization gadget, leaving open the possibility for a self-interpreter."

In this paper, we revisit this phenomenon and show that whether the computability result applies depends on the exact definition of "self-interpreters." In particular, our key insight is based on the observation that real-world typed language interpreters like Haskell GHCi [7] does type-checking inside their interpreters, emitting type errors (which are essentially special values printed) on ill-typed input programs. Since programs are eventually represented as strings, type checkers built inside interpreters are the common practice. We show that if we assume this self-interpreter definition, which does type-checking and emits special values on type error, the computability theory result still applies, and thus such self-interpreters are impossible.

The importance of the definitions of self-interpreters is also discussed in Brown and Palsberg's POPL 2017 work [4], where the authors present two classes of self-interpreters:

typed self-recognizers and typed self-evaluators. According to this classification, our setting can be called untyped self-evaluators.

Our result does not contradict the normalization breakthrough. Instead, we "rebuild" the normalization barrier by considering a different self-interpreter definition, showing whether the computability-style result applies depends on the exact formulation of the problem.

The rest of the paper is organized as follows. Section 2 details our rebuilding process, with Section 2.1 reviewing the computability result, Section 2.2 reviewing Brown and Palsberg's setting, and Section 2.3 presenting our setting. Section 3 discusses possible extensions of our setting. Section 4 concludes the paper.

## 2 Revisiting the Normalization Barrier

We first review the computability result (Section 2.1), and then discuss Brown and Palsberg's setting (Section 2.2) where "self-interpreters" are possible for F-omega, and finally discuss our setting (Section 2.3) where "self-interpreters" are impossible for F-omega.

We assume the standard definition of F-omega as formulated in Brown and Palsberg's original work [3]. In particular, we use the same notations.

### 2.1 Computability Theory

In the literature, two common ways to formulate computability theory are formal languages [10] and sets of natural numbers [1, 5], and the two approaches are equivalent. We follow the second approach: studying subsets of $\mathbb{N}$ and partial functions from $\mathbb{N}$ to $\mathbb{N}$. This is also the formulation used in Brown and Palsberg's paper [3].

**Definition 2.1.** Let $\mathbb{N}$ be the set of natural numbers. A *computable partial function* $f$ is a partial function in $\mathbb{N} \rightharpoonup \mathbb{N}$ that can be computed by a Turing machine. If $f$ is defined on every input, it is called a *computable total function*. For every computable partial function $f$, let $\overline{f}$ be a natural number encoding *one* Turing machine corresponding to $f$.[1] A function $u \in (\mathbb{N} \times \mathbb{N}) \rightharpoonup \mathbb{N}$ is called a *universal function* for all computable total functions if and only if for every computable total function $f$, we have $\forall x, u(\overline{f}, x) = f(x)$.

A universal function is an interpreter in computability theory sense. The set of natural numbers encoding Turing machines is the "programming language." In many cases, we can extend this set to the entire $\mathbb{N}$, where natural numbers not encoding Turing machines can be given a default interpretation. Under this setting, a universal function defines a programming language.

[1]We can choose any Turing machine corresponding to $f$, and $\bar{\cdot}$ itself does not need to be computable.

**Theorem 2.2.** *There does not exist a computable total function serving as the universal function for all computable total functions.*

*Proof.* Suppose we have such a computable total universal function $u$. Thus $u(\overline{f}, x) = f(x)$ for every computable total function $f$. Then $p(x) = u(x, x) + 1$ is also a computable total function, so $p$ has code $\overline{p}$. A contradiction results from $p(\overline{p}) = u(\overline{p}, \overline{p}) + 1 = p(\overline{p}) + 1$. □

Theorem 2.2 can be generalized to broader settings: any subclass of computable total functions with the ability to formulate computations like "+1" does not have universal functions in that class. The proof is essentially the same.

### 2.2 Brown and Palsberg's Setting (Typed, Self-Recognizer)

According to the classification of Brown and Palsberg [4], the normalization breakthrough work [3] constructed a typed self-recognizer, which recovers a program from its representation. Specifically for F-omega, we assume a quoter mapping each term $e$ to a typed representation $\overline{e}$, and the self-interpreter is defined as unquote satisfying the following condition for every well-typed F-omega term $e$.

$$\mathsf{unquote}\ \overline{e} \longrightarrow^* e$$

The quoter must satisfy two conditions [3]: (1) the produced representation must be in $\beta$-normal forms, and (2) the quoter itself must be injective. The first condition rules out using the term itself as the representation, and the second condition rules out using the term's normal form as the representation. There is no assumption on the computability of the quoter, however.

Under this setting, constructions similar to $p(x) = u(x, x) + 1$ are not guaranteed to be feasible. Brown and Palsberg used the term $\lambda x.\lambda y.((\mathsf{unquote}\ x)\ x)$ to demonstrate the non-feasibility, where $\lambda y$ plays the rule of "+1". This term itself is not guaranteed to be typable, let alone the subsequent construction $(p\ \overline{p})$. Thus the type system prevents the diagonalization construction. Note that in this case, the type checking occurs outside the calculus. Nevertheless, the door is open for constructing such an unquote. Indeed, Brown and Palsberg gave such an interpreter [3].

There is an important detail in Brown and Palsberg's term $\lambda x.\lambda y.((\mathsf{unquote}\ x)\ x)$. In the body part $((\mathsf{unquote}\ x)\ x)$, the first occurrence of $x$ serves as the representation of some term $e$, while the second occurrence of $x$, which is still the representation of $e$, serves as the "input" to $e$. This detail plays an important role in our construction in the next subsection.

### 2.3 Our Setting (Untyped, Self-Evaluator)

In practical typed programming languages, programs are still initially represented as strings. As a result, the interpreter's type is essentially $\mathsf{string} \longrightarrow \mathsf{string}$, with the return value as either the evaluation result of the input program, or an error message when there are type errors in the input program. In this case, runtime errors can be classified as part of the evaluation result, and we only distinguish static error messages (type errors) detected before the evaluation.

Under this setting, the type checking occurs inside the interpreter. This setting is similar to computability theory where everything is represented as a natural number. We show that the computability theory trick is applicable here and thus re-establishing the normalization barrier. According to the classification in [4], our setting can be called an untyped self-evaluator.

To represent a program as data, we still need a quoter satisfying the previously discussed two conditions: (1) producing $\beta$-normal forms, and (2) being injective. Here, the essential characteristic is that this quoter should map every F-omega term to the same type, which is similar to real programming languages where every program is represented as a string. This opens the door to handling ill-typed possibilities. We still denote this quoter as $\overline{\cdot}$.

However, pure F-omega (as defined in [3]) does not have direct string types. But we can use Church numerals to serve as natural numbers, which can encode strings using Gödel's encoding or similar techniques [6, 8]. Church numerals can be assigned the following type, denoted as $T$.[2] Conceptually, this encoding process can be understood as encoding F-omega terms as strings and then converting the strings to natural numbers.

$$T = \forall \alpha : *.(\alpha \to \alpha) \to \alpha \to \alpha$$

The quoter $\overline{\cdot}$, in our case, maps both well-typed F-omega terms and ill-typed F-omega terms to $T$. Again, we do not assume whether the quoter is computable or not. Our interpreter interp, on the other hand, has the type $T \to T$, where the result is either the quoted normal form (nf) of the quoted term or a special value denoting that there are type errors (we choose $\Lambda\alpha : *.\lambda x : \alpha.x$). Note that we can stipulate that the quoted value of well-typed terms is always different from this special value. Eventually, we get the following definition. Note that the = here means the two sides have the same normal form.

$$\mathsf{interp}\ \overline{e} = \begin{cases} \overline{\mathsf{nf}(e)} & \text{if } e \text{ is well-typed;} \\ \overline{\Lambda\alpha : *.\lambda x : \alpha.x} & \text{otherwise.} \end{cases}$$

Under this setting, the computability-style construction can be formulated as

$$p = \lambda x : T.\lambda y : \tau.(\mathsf{interp}\ (\mathsf{apply}\ x\ x)))$$

[2]Theoretically, the type parameter $\alpha$ is not important here because we only care about the number of nestings inside the Church numerals. Thus, we can also choose any concrete base type as $\alpha$, if available. Those base types may not include natural numbers or strings, however.

where $\tau$ is an arbitrary type (which could be one of the base types in F-omega, if available) and apply : $(T \times T) \to T$ is a meta-level function satisfying the following.

$$\forall e, x, (\mathsf{apply}\ \overline{e}\ x) = \overline{(e\ x)}$$

Namely, apply takes the code of a term $e$ and a term $x$ of type $T$, and returns the code of the (possibly ill-typed) term $(e\ x)$. In practical programming languages, this roughly corresponds to simply concatenating two pieces of source code, so we regard apply as a natural accompanying feature of our interpreter.

We argue that $p$ is not formalizable inside F-omega. Indeed, if it is formalizable, then it has code $\overline{p}$ of type $T$. Following the computability style argument, we get the following contradiction.

$$\begin{aligned} (p\ \overline{p}) &= \lambda y : \tau.(\mathsf{interp}\ (\mathsf{apply}\ \overline{p}\ \overline{p})) \\ &= \lambda y : \tau.(\mathsf{interp}\ \overline{(p\ \overline{p})}) \\ &= \lambda y : \tau.\mathsf{nf}((p\ \overline{p})) \end{aligned}$$

Note that every part of the above equality chain is well-typed: they are valid terms inside F-omega, assuming the existence of interp and apply. In particular, $(p\ \overline{p})$ is well-typed because $\overline{p}$ always has type $T$, and thus, the interpreter produces the normal form of a well-typed term. In general, because we assumes interp already handles type checking internally, the ill-typed possibility can only occur inside interp, which does not affect the typability of interp itself and terms constructed using interp.

## 3 Discussion

We discussed an untyped self-evaluator where the normalization barrier from computability theory is rebuilt. There are several future directions that could be explored further. First, is it possible to define a typed self-evaluator and rebuild the normalization barrier? Second, the interp term is assumed to come with another meta-level function apply, which is used to deal with the function currying issue inside F-omega. Is it possible to come up with a cleaner definition where we only assume one term? We leave these questions as future work.

## 4 Conclusion

Traditional computability theory concludes that a total universal function for all total computable functions is impossible, which resulted in a conventional wisdom called "the normalization barrier," saying that self-interpreters for strongly normalizing calculi are also impossible. The POPL 2016 work due to Brown and Palsberg broke this barrier and presented a self-interpreter for F-omega, which is a strongly normalizing lambda calculus. That result showed computability theorems were less far-reaching than previously thought. In our paper, we rebuilt the normalization barrier by changing the definition of self-interpreters to include type checking as an internal feature, showing that computability theorems are still applicable from this angle.

It is important to note that all three results (computability theory, Brown and Palsberg, ours) do not contradict each other in any sense. They are applicable in different scenarios, which demonstrates the importance of clarifying the settings and definitions before applying any theorems.

**Acknowledgements.** We thank Govind Gnanakumar for discussions on self interpreters. This work was supported, in part, by the United States National Science Foundation (NSF) under grants No. 2114627 and No. 2237440; and by the Defense Advanced Research Projects Agency (DARPA) under grant N66001-21-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the above sponsoring entities.

## References

[1] Andrea Asperti. 2008. The intensional content of Rice's theorem. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008.* ACM, 113–119. https://doi.org/10.1145/1328438.1328455

[2] Henk Barendregt. 1991. Introduction to Generalized Type Systems. *J. Funct. Program.* 1, 2 (1991), 125–154. https://doi.org/10.1017/S0956796800020025

[3] Matt Brown and Jens Palsberg. 2016. Breaking through the normalization barrier: a self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016.* ACM, 5–17. https://doi.org/10.1145/2837614.2837623

[4] Matt Brown and Jens Palsberg. 2017. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* ACM, 415–428. https://doi.org/10.1145/3009837.3009853

[5] N. Cutland. 1980. *Computability: An Introduction to Recursive Function Theory.* Cambridge University Press. https://books.google.com/books?id=wAstOUE36kcC

[6] Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Vol. 38. Springer, 173–198. https://doi.org/10.1007/BF01700692

[7] HaskellWiki. Accessed in May 2024. GHC/GHCi. https://wiki.haskell.org/GHC/GHCi.

[8] Shuying Liang, Weibin Sun, and Matthew Might. 2014. Fast Flow Analysis with Godel Hashes. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, September 28-29, 2014.* IEEE Computer Society, 225–234. https://doi.org/10.1109/SCAM.2014.40

[9] Frank Pfenning and Peter Lee. 1991. Metacircularity in the Polymorphic lambda-Calculus. *Theor. Comput. Sci.* 89, 1 (1991), 137–159. https://doi.org/10.1016/0304-3975(90)90109-U

[10] M. Sipser. 2012. Introduction to the Theory of Computation. (2012). https://books.google.com/books?id=H94JzgEACAAJ