ON SELF-INTERPRETERS FOR SYSTEM T AND OTHER TYPED λ-CALCULI

ANDREJ BAUER

ABSTRACT. Matt Brown and Jens Palsberg constructed a self-interpreter for System F_ω. They require a self-interpreter to have an injective quoting function and that the source codes be β-normal. I show that under the same constraints there is a self-interpreter for Gödel's System T, and in fact for any strongly normalizing typed λ-calculus expressive enough to encode natural numbers and pairs. The proof is a swindle which makes one think that the constraints by Brown and Palsberg are not good enough, but I show that they are essentially the best we could hope for in terms of type complexity. What we need is a better definition of a self-interpreter that takes into account structural properties of self-interpreters.

1. INTRODUCTION

An interpreter is a program which reads source code and executes it. A self-interpreter is an interpreter implemented in the language that it interprets. The first one was Turing's universal machine, but it was Steve Russell's implementation of `eval` in Lisp that made self-interpreters a popular programming exercise.

In defiance of the received wisdom that a total programming language cannot have a self-interpreter, Matt Brown and Jens Palsberg [3] implemented a self-interpreter for System F_ω, which is a strongly normalizing typed λ-calculus and thus certainly a total language. In order to avoid the trivial self-interpreter they imposed certain constraints on what a self-interpreter is. I show that under the same conditions already Gödel's System T has a self-interpreter (Theorem 3.2). The construction is trivial, which makes one wonder whether something is at fault with the notion of self-interpreter used by Brown and Palsberg. However, I show that there cannot be a significant improvement (Corollary 3.7) because the type of a code must be at least as complex as the type of the program it encodes.

2. UNTYPED AND TYPED SELF-INTERPRETERS

2.1. **Untyped self-interpreters.** We work first in the untyped λ-calculus [2]. A self-interpreter is a program (closed term) $\mathtt{u}$ which takes as an argument the *(source) code* $\ulcorner e \urcorner$ of a program e and outputs a term which is equivalent to e,

$$\mathtt{u}\ \ulcorner e \urcorner \equiv_\beta e.$$

In order to avoid the trivial case $\mathtt{u} = \lambda x\,.\,x$ and $\ulcorner e \urcorner = e$ we should specify what counts as source code. A programmer would certainly expect codes to be strings of characters or abstract syntax trees, and a logician would point out that such data can be coded by

This material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, Air Force Materiel Command, USAF..

numbers. Assuming that all variables are of the form x_0, x_1, x_2, ..., we define the *Gödel encoding* $\mathfrak{g}$ of terms as numbers by

$$\begin{aligned}\mathfrak{g}(x_i) &= 2^0 \cdot 3^i,\\ \mathfrak{g}(e_1\ e_2) &= 2^1 \cdot 3^{\mathfrak{g}(e_1)} \cdot 5^{\mathfrak{g}(e_2)},\\ \mathfrak{g}(\lambda x_i\,.\,e) &= 2^2 \cdot 3^i \cdot 5^{\mathfrak{g}(e)}\end{aligned}$$

and let the quoting function be $\ulcorner e \urcorner = \underline{\mathfrak{g}(e)}$, where $\underline{n}$ is the *Church numeral* representing number n. Implementing the corresponding interpreter is an edifying programming exercise which we leave to the enthusiasts. We note that $\mathtt{u}$ may be implemented using only *primitive* recursion because $\mathtt{u}\ \ulcorner e \urcorner$ calculates its result by structural recursion on e, and primitive recursion suffices to discern the syntax of e from its Gödel code $\mathfrak{g}(e)$.

2.2. **Typed self-interpreters.** Let us now move to the typed setting. As in the untyped case, we call closed terms *programs* and write $\mathsf{Prg}(\tau)$ for the set of programs of type τ.

We consider extensions of the *simply typed λ-calculus* [2, §A.1], which we refer to simply as *calculi*. One is *Gödel's System T*, see [2, §A.2] and [1], which is the simply typed λ-calculus extended with binary products $\sigma \times \tau$, a ground type of natural numbers nat and *primitive* recursion at each type. It is expressive enough to manipulate syntax through Gödel encoding. System T is strongly normalizing [1, §4.3], and so every well-typed term e has a unique β-normal form $\mathfrak{n}(e)$.

Another extension is *PCF* [8], which is the simply typed λ-calculus extended with the natural numbers nat and *general* recursion: for every type τ it has a *fixed-point operator* $\mathsf{fix}_\tau : (\tau \to \tau) \to \tau$ governed by the β-rule $\mathsf{fix}_\tau\ f \to^\beta f\ (\mathsf{fix}_\tau\ f)$. The fixed-point operators make the calculus non-normalizing because $\mathsf{fix}_\tau(\lambda x : \tau\,.\,x)$ is a program of type τ with an infinite sequence of β-reductions.

Definition 2.1. A *typed self-interpreter* is given by a type ν of *(source) codes*, and for each type τ a *quoting function* $\ulcorner\text{-}\urcorner_\tau : \mathsf{Prg}(\tau) \to \mathsf{Prg}(\nu)$ and an *interpreter* $\mathtt{u}_\tau \in \mathsf{Prg}(\nu \to \tau)$ such that $\mathtt{u}_\tau\ \ulcorner e \urcorner_\tau \equiv_\beta e$ for all $e \in \mathsf{Prg}(\tau)$.

Note that the quoting functions need not be λ-definable, i.e., there may be no programs $\mathtt{q}_\tau$ such that $\ulcorner e \urcorner_\tau \equiv_\beta \mathtt{q}_\tau\ e$. The following theorem is the justification for the popular opinion that total languages do not have self-interpreters.

Theorem 2.2. *If a calculus has a self-interpreter then it has fixed-point operators at all types.*

Proof. The proof proceeds by diagonalization much like in Lawvere's fixed point theorem [4]. Let us first show that for every type τ every $f \in \mathsf{Prg}(\tau \to \tau)$ has a fixed point. Define $g : \nu \to \tau$ by $g = \lambda x : \nu\,.\,f\ (\mathtt{u}_{\nu\to\tau}\ x\ x)$, and let $n = \ulcorner g \urcorner_{\nu\to\tau}$ be its code. Then $\mathtt{u}_{\nu\to\tau}\ n\ n$ is a fixed point of f because $\mathtt{u}_{\nu\to\tau}\ n\ n\ \equiv_\beta\ g\ n\ \equiv_\beta\ f\ (\mathtt{u}_{\nu\to\tau}\ n\ n)$. Now we obtain the fixed-point operator $\mathsf{fix}_\tau : (\tau \to \tau) \to \tau$ as the fixed point of the program

$$\lambda y : (\tau \to \tau) \to \tau\,.\,\lambda f : \tau \to \tau\,.\,f\ (y\ f).$$

□

Corollary 2.3. *System T does not have a self-interpreter.*

Proof. In System T successor $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$ has no fixed points. □

The corollary holds for other kinds of calculi, as long as they posses endomaps without fixed points, which is typical of strongly normalizing calculi such as System F_ω.

We can ask whether Theorem 2.2 can be inverted: does a simply typed λ-calculus with natural numbers and fixed-point operators have a self-interpreter? As Alex Simpson

pointed out to me, the question was answered affirmatively by John Longley and Gordon Plotkin who described a rather intricate self-interpreter for PCF [6, Prop. 6].

2.3. **Self-reducers.** Our interpreters convert source code to actual code. An alternative is to have them operate exclusively with source code.

Definition 2.4. A *typed self-reducer* is given by a type ν of *(source) codes*, and for each type τ a *quoting function* $\ulcorner\text{-}\urcorner_\tau : \mathsf{Prg}(\tau) \to \mathsf{Prg}(\nu)$ and a *reducer* $\mathtt{r}_\tau \in \mathsf{Prg}(\nu \to \nu)$ such that $\mathtt{r}_\tau \ulcorner e \urcorner_\tau \equiv_\beta \ulcorner \mathtt{n}(e) \urcorner_\tau$ for all $e \in \mathsf{Prg}(\tau)$. A self-reducer is *acceptable* when for each type τ there is $\mathsf{g}_\tau : \nu \to \mathsf{nat}$ such that $\mathsf{g}_\tau \ulcorner e \urcorner_\tau = \underline{\mathtt{g}(e)}$ for all $e \in \mathsf{Prg}(\tau)$.

The acceptability condition says that the syntax of a program may actually be discerned from its code. We need this condition to avoid the trivial self-reducer

$$\nu = \mathsf{nat}, \qquad \ulcorner e \urcorner_\tau = 0 \qquad \text{and} \qquad \mathtt{r}_\tau = \lambda x : \nu \,.\, x.$$

Theorem 2.5. *The type of source codes of an acceptable self-reducer has a fixed-point operator.*

Proof. The proof is quite similar to the proof of Theorem 2.2. Consider any $f \in \mathsf{Prg}(\nu \to \nu)$ and define

$$g = \lambda x : \nu \,.\, f\ (\mathtt{r}_\nu\ (\mathsf{app}_{\nu,\nu}(x, x)))$$

Let $y = \ulcorner g \urcorner_{\nu\to\nu}$ and compute:

$$\begin{aligned} g\ y &= f\ (\mathtt{r}_\nu\ (\mathsf{app}_{\nu,\nu}(y, y))) \\ &= f\ (\end{aligned}$$

□

3. Brown-Palsberg self-interpreters

To obtain a self-interpreter for System T we need a notion of source code with varying type. The following definition is fashioned after Brown and Palsberg [3].

Definition 3.1. A *weak Brown-Palsberg self-interpreter* is given by, for each type τ, a type of codes $\Box\tau$, a *quoting function* $\ulcorner\text{-}\urcorner_\tau : \mathsf{Prg}(\tau) \to \mathsf{Prg}(\Box\tau)$, and an *interpreter* $\mathtt{u}_\tau : \mathsf{Prg}(\Box\tau \to \tau)$ such that $\mathtt{u}_\tau \ulcorner e \urcorner_\tau \equiv_\beta e$ for all $e \in \mathsf{Prg}(\tau)$. Such an interpreter is *strong* when for every type τ, the quoting function $\ulcorner\text{-}\urcorner_\tau$ is

(1) *normal*: $\ulcorner e \urcorner_\tau$ is β-normal for all $e \in \mathsf{Prg}(\tau)$, and
(2) *acceptable*: there is $\mathsf{g}_\tau : \Box\tau \to \mathsf{nat}$ such that $\mathsf{g}_\tau \ulcorner e \urcorner_\tau = \underline{\mathtt{g}(e)}$ for all $e \in \mathsf{Prg}(\tau)$.

The normality condition expresses the idea that codes should be values, as opposed to programs that still need to be evaluated, while acceptability says that the syntax of an expression is discernible from its code. Brown and Palsberg also require that the quoting function be injective, which follows from our definition because $\mathtt{g}$ is injective. They do not explicitly postulate acceptability, although they provide programs that extract the syntax of an expression from its code.

A Brown-Palsberg self-interpreter cannot have a trivial quoting function $\ulcorner a \urcorner_\tau = a$ because codes must be β-normal, while injectivity of the quoting function prevents coding by β-normal forms $\ulcorner a \urcorner_\tau = \mathtt{n}(a)$.

Theorem 3.2. *System T has a Brown-Palsberg self-interpreter.*

Proof. Define

$$\Box\tau = \mathsf{nat} \times \tau, \qquad \ulcorner e \urcorner_\tau = \langle \underline{\mathtt{g}(e)}, \mathtt{n}(e) \rangle,$$
$$\mathtt{u}_\tau = \lambda x : \mathsf{nat} \times \tau \,.\, \mathsf{snd}\ x, \qquad \mathtt{g}_\tau = \lambda x : \mathsf{nat} \times \tau \,.\, \mathsf{fst}\ x.$$

Clearly we have $\mathtt{u}_\tau \ \ulcorner e \urcorner_\tau \equiv_\beta \mathtt{n}(e) \equiv_\beta e$, $\langle \underline{\mathtt{g}(e)}, \mathtt{n}(e) \rangle$ is β-normal because numerals are β-normal and so are pairs of β-normal terms, and the quoting function is acceptable by fiat. □

It is clear that the same proof applies to any calculus that has binary products, natural numbers, and any notion of normal form. System F_ω is an example, so we could replace the elegant self-interpreter by Brown and Palsberg with our technologically primitive solution.

The proof of Theorem 3.2 abuses the fact that Brown-Palsberg interpreters allow codes to be as complex as the programs they encode. Theorem 2.2 prevents us from using a fixed type of codes, but perhaps $\Box\tau$ can at least be less complex than τ? We shall show that this is not possible, but first we need a precise measure of complexity.

Definition 3.3. The *level* of a type τ is defined inductively as follows:

$$\mathsf{lev}(\mathsf{nat}) = 0$$
$$\mathsf{lev}(\sigma \times \tau) = \max(\mathsf{lev}(\sigma), \mathsf{lev}(\tau))$$
$$\mathsf{lev}(\sigma \to \tau) = \max(1 + \mathsf{lev}(\sigma), \mathsf{lev}(\tau)).$$

The level of τ is the deepest nesting of $\to$ to the left. The *pure types* $\nu_0, \nu_1, \ldots$ defined by

$$\nu_0 = \mathsf{nat} \qquad \text{and} \qquad \nu_{k+1} = \nu_k \to \mathsf{nat}$$

have arbitrarily high levels because $\mathsf{lev}(\nu_k) = k$.

Definition 3.4. A type σ is a *retract* of τ, written $\sigma \lhd \tau$, if there are programs $s : \sigma \to \tau$ and $r : \tau \to \sigma$, respectively called a *section* and a *retraction*, such that

$$\lambda x : \sigma \,.\, r\ (s\ x) \equiv_{\beta\eta} \lambda x : \sigma \,.\, x.$$

If additionally $\lambda y : \tau \,.\, s\ (r\ y) \equiv_{\beta\eta} \lambda y : \tau \,.\, y$ then we say that σ and τ are *isomorphisms* and that σ and τ are *isomorphic*, written $\sigma \cong \tau$.

Some basic observations about retracts are:

(1) every type is a retract of itself, $\tau \lhd \tau$,
(2) being a retract is transitive: if $\rho \lhd \sigma$ and $\sigma \lhd \tau$ then $\rho \lhd \tau$,
(3) if $\sigma \lhd \sigma'$ and $\tau \lhd \tau'$ then $\sigma \times \tau \lhd \sigma' \times \tau'$ and $\sigma' \to \tau \lhd \sigma \to \tau'$,
(4) $\sigma \lhd \sigma \times \tau$ and $\tau \lhd \sigma \times \tau$.

The last observation relies on the fact that every type in System T is inhabited. The following lemma has been known for a long time, e.g. [9, §1.8].

Lemma 3.5. *If* $\mathsf{lev}(\sigma) \leq \mathsf{lev}(\tau)$ *then* σ *is a retract of* τ.

Proof. We first verify that $\nu_k \lhd \nu_{k+1}$ for all $k \in \mathbb{N}$. Define programs $s_k : \nu_k \to \nu_{k+1}$ and $r_k : \nu_{k+1} \to \nu_k$ by

$$s_0 = \lambda n : \mathsf{nat} \,.\, \lambda m : \mathsf{nat} \,.\, n,$$
$$r_0 = \lambda f : \mathsf{nat} \to \mathsf{nat} \,.\, f\ \mathsf{zero},$$

and

$$s_{k+1} = \lambda f : \nu_{k+1} \,.\, \lambda g : \nu_{k+1} \,.\, f\ (r_k\ g),$$
$$r_{k+1} = \lambda f : \nu_{k+2} \,.\, \lambda g : \nu_k \,.\, f\ (s_k\ g).$$

We check by induction on k that s_k and r_k form a section-retraction pair. The base case is confirmed by

$$\lambda x : \mathsf{nat} \,.\, r_0 \; (s_0 \; x) \;\equiv_\beta\; \lambda x : \mathsf{nat} \,.\, (\lambda m : \mathsf{nat} \,.\, x) \; \mathsf{zero} \;\equiv_\beta\; \lambda x : \mathsf{nat} \,.\, x.$$

and the induction step by

$$\begin{aligned}
\lambda f : \nu_{k+1} \,.\, r_{k+1} \; (s_{k+1} \; f) \;&\equiv_\eta\; \lambda f : \nu_{k+1} \,.\, \lambda g : \nu_k \,.\, r_{k+1} \; (s_{k+1} \; f) \; g \\
&\equiv_\beta\; \lambda f : \nu_{k+1} \,.\, \lambda g : \nu_k \,.\, (s_{k+1} \; f) \; (s_k \; g) \\
&\equiv_\beta\; \lambda f : \nu_{k+1} \,.\, \lambda g : \nu_k \,.\, f \; (r_k \; (s_k \; g)) \\
&\equiv_{\beta\eta}\; \lambda f : \nu_{k+1} \,.\, \lambda g : \nu_k \,.\, f \; g \\
&\equiv_\eta\; \lambda f : \nu_{k+1} \,.\, f.
\end{aligned}$$

We used the induction step to pass from the third to the fourth line.

Next, let $p : \mathsf{nat} \to \mathsf{nat} \times \mathsf{nat}$ and $q : \mathsf{nat} \to \mathsf{nat} \times \mathsf{nat}$ be isomorphisms witnessing $\mathsf{nat} \cong \mathsf{nat} \times \mathsf{nat}$. Then $p_k : \nu_k \to \nu_k \times \nu_k$ and $q_k : \nu_k \times \nu_k \to \nu_k$ defined by $p_0 = p$, $q_0 = q$ and

$$\begin{aligned}
p_{k+1} &= \lambda f : \nu_{k+1} \,.\, \langle (\lambda g : \nu_k \,.\, \mathsf{fst} \; (q \; (f \; g))), (\lambda g : \nu_k \,.\, \mathsf{snd} \; (q \; (f \; g))) \rangle \\
q_{k+1} &= \lambda f : \nu_{k+1} \times \nu_{k+1} \,.\, \lambda g : \nu_k \,.\, p \; \langle \mathsf{fst} \; f \; g, \mathsf{snd} \; f \; g \rangle
\end{aligned}$$

are isomorphism between ν_k and $\nu_k \times \nu_k$.

To prove the lemma it suffices to show that the types τ and $\nu_{\mathsf{lev}(\tau)}$ are retracts of each other for any type τ. We proceed by induction on the structure of τ.

The base case nat is trivial.

For a product type $\sigma \times \tau$ we have a chain of retractions

$$\sigma \times \tau \;\lhd\; \nu_{\mathsf{lev}(\sigma)} \times \nu_{\mathsf{lev}(\tau)} \;\lhd\; \nu_{\mathsf{lev}(\sigma \times \tau)} \times \nu_{\mathsf{lev}(\sigma \times \tau)} \;\cong\; \nu_{\mathsf{lev}(\sigma \times \tau)}.$$

In the opposite direction, if $\mathsf{lev}(\sigma) \leq \mathsf{lev}(\tau)$ then

$$\nu_{\mathsf{lev}(\sigma \times \tau)} \;=\; \nu_{\mathsf{lev}(\tau)} \;\lhd\; \tau \;\lhd\; \sigma \times \tau.$$

The case $\mathsf{lev}(\tau) \leq \mathsf{lev}(\sigma)$ is symmetric.

Finally consider a function type $\sigma \to \tau$. If $\mathsf{lev}(\tau) = 0$ then $\tau = \mathsf{nat}$, hence

$$\sigma \to \tau \;\lhd\; \nu_{\mathsf{lev}(\sigma)} \to \mathsf{nat} \;=\; \nu_{\mathsf{lev}(\sigma)+1} \;=\; \nu_{\mathsf{lev}(\sigma \to \tau)}$$

and

$$\nu_{\mathsf{lev}(\sigma \to \tau)} \;=\; \nu_{\mathsf{lev}(\sigma)+1} \;=\; \nu_{\mathsf{lev}(\sigma)} \to \mathsf{nat} \;\lhd\; \sigma \to \mathsf{nat} \;=\; \sigma \to \tau.$$

If $\mathsf{lev}(\tau) = n + 1$ then

$$\begin{aligned}
\sigma \to \tau \;\lhd\; \nu_{\mathsf{lev}(\sigma)} \to \nu_{n+1} \;\cong\; \nu_{\mathsf{lev}(\sigma)} \times \nu_n \to \mathsf{nat} \;\lhd\; \\
\nu_{\max(\mathsf{lev}(\sigma),n)} \to \mathsf{nat} \;=\; \nu_{1+\max(\mathsf{lev}(\sigma),n)} \;=\; \nu_{\mathsf{lev}(\sigma \to \tau)}.
\end{aligned}$$

and

$$\begin{aligned}
\nu_{\mathsf{lev}(\sigma \to \tau)} \;=\; \nu_{1+\max(\mathsf{lev}(\sigma),n)} \;=\; \nu_{\max(\mathsf{lev}(\sigma),n)} \to \mathsf{nat} \;\lhd\; \\
\nu_{\mathsf{lev}(\sigma)} \times \nu_n \to \mathsf{nat} \;\cong\; \nu_{\mathsf{lev}(\sigma)} \to \nu_{n+1} \;\lhd\; \sigma \to \tau. \quad \square
\end{aligned}$$

With the lemma in hand we can prove an obstruction theorem for self-interpreters.

Theorem 3.6. *For any type τ, if a weak Brown-Palsberg self-interpreter satisfies $\mathsf{lev}(\Box\tau) < \mathsf{lev}(\tau)$ then every $f \in \mathsf{Prg}(\tau \to \tau)$ has a fixed point with respect to $\beta\eta$-equivalence.*

Proof. If $\mathsf{lev}(\Box\tau) < \mathsf{lev}(\tau)$ then $\mathsf{lev}(\Box\tau \to \tau) \leq \mathsf{lev}(\tau)$, hence Lemma 3.5 gives us a section and a retraction

$$s : (\Box\tau \to \tau) \to \tau \qquad \text{and} \qquad r : \tau \to (\Box\tau \to \tau).$$

Define $g : \Box\tau \to \tau$ by $g = \lambda x{:}\Box\tau\,.\, f\ (r\ (\mathtt{u}_\tau\ x)\ x)$ and let $n = \ulcorner s\ g \urcorner_\tau$. Then

$$r\ (\mathtt{u}_\tau\ n)\ n\ \equiv_\beta\ r\ (s\ g)\ n\ \equiv_{\beta\eta}\ g\ n\ \equiv_\beta\ f\ (r\ (\mathtt{u}_\tau\ n)\ n). \qquad \Box$$

The following corollary squashes any hope of a significant improvement of the notion of Brown-Palsberg interpreters, as far as complexity of the types of codes is concerned.

Corollary 3.7. *A weak Brown-Palsberg self-interpreter for System T satisfies* $\mathsf{lev}(\Box\tau) \geq \mathsf{lev}(\tau)$ *for every type τ.*

Proof. It suffices to show that every type τ has a program $f : \tau \to \tau$ without a $\beta\eta$-fixed point. By Lemma 3.5 there is a section $s : \mathsf{nat} \to \tau$ and a corresponding retraction $r : \tau \to \mathsf{nat}$. Define

$$f = \lambda x{:}\tau\,.\, s\ (\mathsf{succ}(r\ x)).$$

To see that f does not have a fixed point, suppose $f\ e \equiv_{\beta\eta} e$ for some $e \in \mathsf{Prg}(\tau)$. Then

$$r\ e \equiv_{\beta\eta} r\ (f\ e) \equiv_\beta r\ (s\ (\mathsf{succ}\ (r\ e))) \equiv_{\beta\eta} \mathsf{succ}\ (r\ e).$$

Let n be the unique number such that $\mathfrak{n}(r\ e) \equiv_\beta \underline{n}$. Then

$$\underline{n+1} = \mathsf{succ}\ \underline{n} \equiv_\beta \mathsf{succ}\ (r\ e) \equiv_{\beta\eta} (r\ e) \equiv_\beta \underline{n},$$

which is a contradiction because it implies $n + 1 = n$. To see this, we could presumably find a reference to a suitable theorem about $\beta\eta$-normal forms for System T, but here is a semantic proof. The set-theoretic model of System T validates both β- and η-equality, thus $\underline{k} \equiv_{\beta\eta} \underline{m}$ implies that in the set-theoretic model the interpretations of $\underline{k}$ and $\underline{m}$ are equal, but they are of course k and m, respectively. $\Box$

4. The moral

The self-interpreter for F_ω given by Brown and Palsberg has important structural properties that Definition 3.1 fails to capture. For instance, their encoding of types commutes with substitution [3, Thm. 5.2] and is a congruence with respect to type equality [3, Thm. 5.3]. In the original work [7] on meta-circularity Frank Pfenning and Peter Lee called such phenomena *reflexivity*. Unfortunately they spoke of it at an informal level and did not provide a definition.

In order to shed further light on self-interpreters for total languages we need a definition of self-interpreters which takes into account structural properties of self-interpreters that distinguishes the interpreter by Brown and Palsberg from the one given in Theorem 3.2. However, it will not do to simply require that $\tau \mapsto \Box\tau$ be a congruence which commutes with substitution, as that is not a general enough idea. I would instead expect to see a definition of a structure-preserving *homomorphism* between calculi, like in algebra. A good starting point might be John Longley's notion of computability structures and simulations [5].

REFERENCES

[1] Jeremy Avigad and Solomon Feferman. Chapter V: Gödel's Functional ("Dialectica") Interpretation. In Samuel R. Buss, editor, *Handbook of Proof Theory*, volume 137 of *Studies in Logic and the Foundations of Mathematics*, pages 337–405. Elsevier, 1998.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. College Publications, 1984.

[3] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F-omega. In *Principles of Programming Languages (POPL)*, January 2016.

[4] F. William Lawvere. Diagonal arguments and cartesian closed categories. *Lecture Notes in Mathematics*, 92:134–145, 1969. Republished in: Reprints in Theory and Applications of Categories, No. 15 (2006), 1–13.

[5] John Longley. Computability structures, simulations and realizability. *Mathematical Structures in Computer Science*, 24(2), 2014.

[6] John Longley and Gordon Plotkin. Logical full abstraction and pcf. In *Tbilisi Symposium on Language, Logic and Computation. SiLLI/CSLI*, pages 333–352. SiLLI/CSLI, 1996.

[7] Frank Pfenning and Peter Lee. Metacircularity in the polymorphic λ-calculus. *Theoretical Computer Science*, 89(1):137 – 159, 1991.

[8] G.D. Plotkin. Lcf considered as a programming language. *Theoretical Computer Science*, 5(3):223 – 255, 1977.

[9] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Number 344 in Lecture Notes in Mathematics. Springer-Verlag, 1973.

ANDREJ BAUER, UNIVERSITY OF LJUBLJANA, SLOVENIA
E-mail address: Andrej.Bauer@andrej.com