A Fuelled Self-Reducer for System T via Primrose (Short Paper)

Greg Brown
University of Edinburgh
Edinburgh, UK
greg.brown01@ed.ac.uk

Abstract

Self-reducers are programs that reduce embeddings of expressions to their normal forms. They are written in the same language that they reduce. To date, there are no self-reducers for strongly-normalising languages. I have implemented a fuelled self-reducer for System T. Rather than working with Gödel encodings, I used Kiselyov's [14], and Longley and Normann's [16] encodings for structured types such as products, sums and some inductive types. Working with these types within System T produces large unreadable terms. I promote these structured types to first class features of a new metalanguage called Primrose. Primrose compiles into System T. Work on Primrose is ongoing.

Keywords: partial evaluation, primitive recursion, self-reducer

ACM Reference Format:
Greg Brown. 2024. A Fuelled Self-Reducer for System T via Primrose (Short Paper). In *Proceedings of Partial Evaluation and Program Manipuation (PEPM '25).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Self-reduction is a partial evaluation technique with a long history. A reducer `r` is a program that given an embedding of a source term, computes the normal form of its input. For a given definition of normal form, the reduction equation states that a reducer `r` when applied to the embedding of a term `e` reduces to the embedding of the normal form of `e`:

$$\mathtt{r}\ \ulcorner \mathtt{e} \urcorner \rightsquigarrow^* \ulcorner \mathbf{nf}(\mathtt{e}) \urcorner$$

A reducer is a *self-reducer* when the program `r` is written in the language that `r` reduces.

Many self-reducers have been implemented over the last fourty years. Jones et al. [13] implemented the first self-applicable self-reducer. Berarducci and Böhm [3] implemented a self-reducer for the untyped λ-calculus. Launchbury [15] implemented one for the simply-typed second-order language LML, and Naylor [19] for a subset of Haskell. These languages all feature general recursion, and include non-normalising terms.

Bauer [2] provided a lower bound on the recursive power needed to express a self reducer:

Theorem 1.1. *There are no self-reducers in System T for beta normal forms that embed terms using Gödel encodings.*

As there are no general self-reducers for System T, I investigate *fuelled* self-reducers. These are self-reducers with an additional fuel parameter, which is a natural number giving an upper bound on the amount of work necessary. Given a fuel function F calculating the amount of fuel required we can write the fuelled reducer equation:

$$\forall k \geq F(\mathtt{e}).\mathtt{r}\ k\ \ulcorner \mathtt{e} \urcorner \rightsquigarrow^* \ulcorner \mathbf{nf}(\mathtt{e}) \urcorner$$

If there exists a System T program `f` that computes an upper bound for a fuel function, then there cannot exist a fuelled self-reducer. If such a reducer `r` did exist one could construct a general self-reducer by computing $\lambda x.\mathtt{r}\ (\mathtt{f}\ x)\ x$ which contradicts Bauer's theorem. Taking the contrapositive gives the following theorem

Theorem 1.2. *If there exists a fuelled self-reducer* `r` *with fuel function F, there is no System T program to compute an upper bound of F* `e` *from the Gödel encoding of* `e`.

One of my contributions is the construction of such a fuelled self-reducer for System T, using the number of reduction steps as the amount of fuel. I initially tried to work directly with Gödel encodings of terms, but I found this difficult. Instead I used work by Kiselyov [14] and Longley and Normann [16] to encode products, sums and finitary positive inductive types within System T. As these type encodings are also complex, I used Idris 2 [5] as a meta language to help construct the term. The final term is large; a condensed version is 336 kibibytes. Compare this to the Idris program generating it, which is 75 kibibytes split over several files.

To overcome size issues and the complexity of encoding types within System T, I have promoted them to first-class features in a new meta-language called Primrose. Primrose corresponds to the simply-typed λ-calculus extended with products, sums and finitary positive inductive types. I present its type system and a translation from Primrose into System T. Work on Primrose is ongoing, with the intent to write the translation to System T as a Primrose program and write a fuelled self-reducer. Combining these two programs will allow for a new, hopefully simpler self-reduction algorithm for System T.

Outline: in section 2 I formally define self-reducers and fuelled self-reducers for strongly-typed languages. I introduce the syntax and type system of Primrose in section 3. I

PEPM '25, January 21, 2025, Denver, CO, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{nat}}{\Gamma \vdash \mathsf{suc}\ t : \mathsf{nat}} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash u : A \to A \quad \Gamma \vdash v : \mathsf{nat}}{\Gamma \vdash \mathsf{primrec}\ t\ u\ v : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B}$$

Figure 1. Definition of System T

define a translation function from Primrose into System T in section 3.1. In section 3.2 I describe my plans for the future of Primrose. I conclude with a discussion on related work in section 4.

Contributions:

- report about an ongoing implementation of the meta-language Primrose for working with System T more ergonomically.
- constructed an explicit fuelled self-reducer in System T.

2 Strongly-Typed Self-Reducers

Self-reducers are programs that manipuate embeddings of programs. The details of how they function depend on the details of the embedding. For a simply-typed language, one needs to choose both a type representing embedded values and a function converting programs to that type.

Definition 2.1. A *language* consists of:

- a set Ty of types;
- a type operation $\to$: $\mathsf{Ty} \times \mathsf{Ty} \to \mathsf{Ty}$ for function types;
- a Ty-indexed set $\mathsf{Tm}(-)$ of terms of a given type;
- a term operation $\$: \mathsf{Tm}(A \to B) \times \mathsf{Tm}(A) \to \mathsf{Tm}(B)$ for function applications;
- a Ty-indexed reduction relation $\rightsquigarrow_A \subseteq \mathsf{Tm}(A) \times \mathsf{Tm}(A)$;
- a Ty-indexed partial function $\mathbf{nf}_A : \mathsf{Tm}(A) \rightharpoonup \mathsf{Tm}(A)$ specifying the normal form of a term, when it exists.

I write $\rightsquigarrow^*$ as the transitive closure of $\rightsquigarrow$.

I will elide the type annotations on $\rightsquigarrow^*$ and $\mathbf{nf}$. I will also write an application $f\ \$\ x$ as justaposition $f\ x$. When talking about multiple languages, I will use superscripts on Ty and Tm to distinguish between the languages.

This definition of languages is compatible with untyped languages, by taking $\mathsf{Ty} = \{\star\}$ and $\star \to \star := \star$.

Example 2.2. System T is the language consisting of types of the form $\mathsf{Ty} ::= \mathsf{nat} \mid \mathsf{Ty} \to \mathsf{Ty}$. Members of $\mathsf{Tm}(A)$ are terms t such that the judgement $\vdash t : A$ from fig. 1 holds. Take $\rightsquigarrow$ to be beta-reduction, and $\mathbf{nf}$ to beta normal forms.

Languages are capable of embedding terms of another. This is commonly called 'quoting' the term [2, 15, 8].

Definition 2.3. A *quotation function* of an object language O into a meta language M consists of:

$$\begin{aligned}
\mathfrak{g}(x) &:= 2^0 \cdot 3^x \\
\mathfrak{g}(0) &:= 2^1 \\
\mathfrak{g}(\mathsf{suc}) &:= 2^2 \\
\mathfrak{g}(\mathsf{rec}\ z\ s\ n) &:= 2^3 \cdot 3^{\mathfrak{g}(z)} \cdot 5^{\mathfrak{g}(s)} \cdot 7^{\mathfrak{g}(n)} \\
\mathfrak{g}(\lambda x.t) &:= 2^4 \cdot 3^{\mathfrak{g}(t)} \\
\mathfrak{g}(f\ t) &:= 2^5 \cdot 3^{\mathfrak{g}(f)} \cdot 5^{\mathfrak{g}(t)}
\end{aligned}$$

Figure 2. A Gödel encoding of System T terms, assuming variables are using de Bruijn indices.

- a meta type $E \in \mathsf{Ty}^M$ of program codes;
- for all object types $A \in \mathsf{Ty}^O$ a function

$$\ulcorner - \urcorner_A : \mathsf{Tm}^O(A) \to \mathsf{Tm}^M(E)$$

A *self-quotation function* is a quotation function where the object and meta languages are equal.

Note that a quotation function need not be computable nor injective. Also notice that not all meta terms of type E need to represent well-typed object terms.

Example 2.4. Here are three self-quotation functions for System T, choosing E to be nat:

1. the constant function $\ulcorner \mathtt{e} \urcorner_A := 0$;
2. the Gödel encoding $\ulcorner \mathtt{e} \urcorner_A := \mathfrak{g}(\mathtt{e})$;
3. the Gödel encoding of normal forms $\ulcorner \mathtt{e} \urcorner_A := \mathfrak{g}(\mathbf{nf}(\mathtt{e}))$.

The first of these functions forgets everything about the term. The second uses the Gödel encoding, defined in fig. 2, to represent the term as an abstract syntax tree. The third forgets the original term, instead embedding its normal form. Some choices of quotation functions are more useful than others.

Definition 2.5. A *reducer* relative to quotation function $\ulcorner - \urcorner$ from O to M is an Ty^O-indexed family of meta terms $\mathtt{r}_A : \mathsf{Tm}^M(E \to E)$ such that for any object term $\mathtt{e} \in \mathsf{Tm}^O(A)$ the reduction

$$\mathtt{r}_A\ \ulcorner \mathtt{e} \urcorner_A \rightsquigarrow^* \ulcorner \mathbf{nf}(\mathtt{e}) \urcorner_A \tag{1}$$

holds whenever the normal form $\mathbf{nf}(\mathtt{e})$ exists.

A *self-reducer* is a reducer where the object language and meta language are the same.

Notice that a reducer is a family of programs. This is important for typed object languages whose normal forms are eta-long. The behaviour of a reducer is unspecified when applied to object terms without a normal form. A common design goal is to ensure that a reducer terminates for all object terms.

Example 2.6. The identity function $\lambda x.x$ is a self-reducer for quotation functions 1 and 3 of example 2.4 as $\mathbf{nf}(\mathbf{nf}(\mathtt{e})) = \mathbf{nf}(\mathtt{e})$ for all terms $\mathtt{e}$.

Bauer [2] proved theorem 1.1 regarding the non-existence of self-reducers for System T. The proof relies on being able to construct terms that compute:

- the Gödel encoding of a natural number;
- a natural number from its Gödel encoding;
- the encoding of applying two terms together.

From this one can construct a fixed point for every term of type $\mathsf{nat} \to \mathsf{nat}$. As the successor function has no fixed point, this is a contradiction.

Adding *fuel* to a reducer can prevent encoding applications, circumventing this impossibility result.

Definition 2.7. A *fuelled* reducer for quotation function $\ulcorner - \urcorner$ from O to M and a fuel functions $F_A : \mathsf{Tm}^O(A) \to \mathbb{N}$ is an Ty^O-indexed family of meta terms $\mathtt{r}_A : \mathsf{Tm}^M(\mathsf{nat} \to E \to E)$ such that for any object term $\mathtt{e} \in \mathsf{Tm}^O(A)$ the reductions

$$\forall k \geq F_A(\mathtt{e}).\mathtt{r}_A\ k\ \ulcorner \mathtt{e} \urcorner_A \rightsquigarrow^* \ulcorner \mathbf{nf}(\mathtt{e}) \urcorner_A \tag{2}$$

hold whenever the normal form $\mathbf{nf}(\mathtt{e})$ exists.

A fuelled self-reducer is a fuelled reducer where the object language and meta language are the same.

Fuel is an additional parameter given to an iterative process describing an upper bound on the number of steps to reach the final value. Adding fuel overcomes the restrictions of theorem 1.1. To adapt Bauer's impossibility proof to a fuelled reducer, one must be able to calculate the fuel required to reduce an application. As fuelled self-reducers exist for System T, it suggests the fuel needed to reduce an application cannot be computed.

The inequality in eq. (2) helps to prevent 'cheating'. Without it, one could smuggle a complete computation through the fuel function. For example, for the fuel function $F_A(\mathtt{e}) := \mathfrak{g}(\mathbf{nf}(\mathtt{e}))$, the program $\mathtt{r}\ k\ x := k$ would satisfy this weaker definition.

3 Primrose

Encoding complex types into System T results in large terms that are impossible to read. I introduce a new language, Primrose, to act as a new metalanguage for System T. By design, any Primrose term can be compiled into a well-typed System T term.

Primrose is the simply-typed λ-calculus extended with sums, products and finitary positive inductive types. The syntax is shown in fig. 6 and fig. 7 shows the typing rules. Primrose has two typing judgements:

- A finpos X checks all uses of X within A are finitary positive;
- $\Gamma \vdash t : A$ checks term t at type A under context Γ;

The finpos judgement is related to the encoding of inductive types within System T. I will provide more commentary when describing the translation of Primrose into System T later. The other typing rules are standard.

$$\begin{aligned}
\mathsf{let}\ x = e\ \mathsf{in}\ t &\rightsquigarrow t[x/e] \\
(\lambda x.t)\ u &\rightsquigarrow t[x/u] \\
\langle L_1 : t_1, \ldots, L_n : t_n \rangle.L_i &\rightsquigarrow t_i \\
\mathsf{case}\ L_i\ t\ \mathsf{of} \\
\{L_1\ x_1 \Rightarrow t_1; \ldots; L_n\ x_n \Rightarrow t_n\} &\rightsquigarrow t_i[x_i/t] \\
!(\sim t) &\rightsquigarrow t \\
\mathsf{fold}\ \sim t\ \mathsf{as}\ x\ \mathsf{by}\ u &\rightsquigarrow \\
u[x/\mathsf{unroll}\ (\mathsf{map}\ (y \mapsto \mathsf{fold}\ y\ \mathsf{as}\ x\ \mathsf{by}\ u)\ (\mathsf{roll}\ t))]
\end{aligned}$$

Figure 3. Primrose reduction rules. The map, roll, and unroll functions are described later.

The reduction relation for Primrose terms is shown in fig. 3. The most interesting case is fold. First, map is a meta-function that performs the fold recursively on all appropriate subterms of t. This value is then substituted into the accumulator u. I will provide an inductive definition for map later.

3.1 Translation into System T

By design, all Primrose types can be translated into System T types. I will use type encodings by Kiselyov [14] and Longley and Normann [16] to perform the translation.

System T has only two type formers: nat and function arrows. As nat is inhabited, all System T types are inhabited; for function types, use a constant function returning an arbitrary value. I will denote arbitrary inhabitants with the constant arb.

This allows System T to express the untagged union of any two types like the `union` construct in C. Given two types A and B, the union type $A \sqcup B$ has four operations satisfying two equations:

$$\begin{aligned}
&\mathsf{inl} : A \to A \sqcup B \qquad \mathsf{inr} : B \to A \sqcup B \\
&\mathsf{prl} : A \sqcup B \to A \qquad \mathsf{prr} : A \sqcup B \to B \\
&\mathsf{prl}\ (\mathsf{inl}\ x) =^{\beta,\eta} x \qquad \mathsf{prr}\ (\mathsf{inr}\ y) =^{\beta,\eta} y
\end{aligned}$$

I give the full definitions of $A \sqcup B$, inl and prl in fig. 4. The definition proceeds by recursion on the types A and B. inr and prr are defined symmetrically.

Example 3.1. The union of types A_0 and B_0

$$\begin{aligned}
A_0 &:= (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat} \\
B_0 &= \mathsf{nat} \to \mathsf{nat}
\end{aligned}$$

is the type $A_0 \sqcup B_0 = (\mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}$. The left injection and projection are both identity functions. A function f of type B_0 is right-injected to the function $\lambda g, x.f\ (g\ \mathsf{arb}_{\mathsf{nat}})$. An encoding h of type $A_0 \sqcup B_0$ is right-projected into the function $\lambda x.h\ (\lambda y.x)\ \mathsf{arb}_{\mathsf{nat}}$.

Union types are essential for defining a translation of Primrose into System T. Another useful type are lists. A list of

$$\begin{aligned}
\mathsf{nat} \sqcup \mathsf{nat} &:= \mathsf{nat} \\
(A \to B) \sqcup \mathsf{nat} &:= A \to B \sqcup \mathsf{nat} \\
\mathsf{nat} \sqcup (A \to B) &:= A \to \mathsf{nat} \sqcup B \\
(A \to B) \sqcup (C \to D) &:= A \sqcup C \to B \sqcup D
\end{aligned}$$

$$\begin{aligned}
\mathsf{inl}_{\mathsf{nat},\mathsf{nat}} &:= \lambda n.n \\
\mathsf{inl}_{A\to B,\mathsf{nat}} &:= \lambda f, x.\mathsf{inl}_{B,\mathsf{nat}}\ (f\ x) \\
\mathsf{inl}_{\mathsf{nat},A\to B} &:= \lambda n, x.\mathsf{inl}_{\mathsf{nat},B}\ n \\
\mathsf{inl}_{A\to B,C\to D} &:= \lambda f, x.\mathsf{inl}_{B,D}\ (f\ (\mathsf{prl}_{A,C}\ x))
\end{aligned}$$

$$\begin{aligned}
\mathsf{prl}_{\mathsf{nat},\mathsf{nat}} &:= \lambda n.n \\
\mathsf{prl}_{A\to B,\mathsf{nat}} &:= \lambda f, x.\mathsf{prl}_{B,\mathsf{nat}}\ (f\ x) \\
\mathsf{prl}_{\mathsf{nat},A\to B} &:= \lambda f.\mathsf{prl}_{\mathsf{nat},B}\ \mathsf{arb}_A \\
\mathsf{prl}_{A\to B,C\to D} &:= \lambda f, x.\mathsf{prl}_{B,D}\ (f\ (\mathsf{inl}_{A,C}\ x))
\end{aligned}$$

Figure 4. Encoding of union types in System T.

type A is represented by a function from positions to values: $A^* := \mathsf{nat} \to A$. Item lookup is performed by calling this function with the index to search. Constructing the list is more involved, with the constructors nil and cons defined as follows.

$$\begin{aligned}
\mathsf{nil} &:= \lambda x.\mathsf{arb} \\
\mathsf{cons} &:= \lambda v, f, i.\mathsf{primrec}\ v\ (\lambda x.f\ (\mathsf{pred}\ i))\ i
\end{aligned}$$

The empty list contains arbitrary data at all positions. The cons operation uses primrec to test whether to return the head of the list or a value from the tail.

I will write statements of the form $[x; y; z]$ as syntactic sugar for lists.

For the remainder of this section, I will construct a pair of functions translating closed types and open terms from Primrose to System T, both called $[\![-]\!]$.

I use Kiselyov's [14] encoding of product types to translate Primrose products into System T types.

$$\begin{aligned}
[\![\langle L_1 : A_1, \ldots, L_n : A_n \rangle]\!] &:= \left(\bigsqcup_{i=1}^{n} [\![A_i]\!]\right)^* \\
[\![\langle L_1 : t_1, \ldots, L_n : t_n \rangle]\!] &:= [\mathsf{in}_1\ [\![t_1]\!], \ldots, \mathsf{in}_n\ [\![t_n]\!]] \\
[\![e.L_i]\!] &:= \mathsf{pr}_i\ ([\![e]\!]\ i)
\end{aligned}$$

Products are represented as a heterogeneous list. I use union types so that each element is represented in a uniform way. The tupling operation stores the components of the product in a list, after injecting each component into the union. The projection operation looks up the component from the list, and then projects it to the correct type.

I will write $A \times B$ for the product of two arbitrary System T types, with tupling operation $\langle -, - \rangle$ and projections fst and snd, defined similarly to above. I will also allow pattern-matching expressions within System T, such as let $\langle x, y \rangle = p$ in t.

Kiselyov [14] also gives an encoding of sum types.

$$\begin{aligned}
[\![\{L_1 : A_1, \ldots, L_n : A_n\}]\!] &:= \mathsf{nat} \times \bigsqcup_{i=1}^{n} [\![A_i]\!] \\
[\![L_i\ t]\!] &:= \langle i, \mathsf{in}_i\ [\![t]\!] \rangle \\
\left[\!\!\left[\begin{array}{l}\mathsf{case}\ e\ \mathsf{of} \\ \{L_1\ x_1 \Rightarrow t_1; \ldots L_n\ x_n \Rightarrow t_n\}\end{array}\right]\!\!\right] &:= \begin{array}{l}\mathsf{let}\ \langle i, v \rangle = [\![e]\!]\ \mathsf{in} \\ [\ \mathsf{let}\ x_1 = \mathsf{pr}_1\ v\ \mathsf{in} [\![t_1]\!] \\ ;\ \ldots \\ ;\ \mathsf{let}\ x_n = \mathsf{pr}_n\ v\ \mathsf{in} [\![t_n]\!] \\ ]\ (\mathsf{pred}\ i)\end{array}
\end{aligned}$$

Sums are represented as tagged unions. The first component of the pair identifies which case of the union the value occupies. The second stores the value itself. The case expression uses a list to compute the value of each branch. The tag then selects the appropriate branch.

The final type constructor to translate into System T are inductive types. Longley and Normann [16, prop 4.2.12] describes a general strategy for encoding finitary positive types within System T. This is exactly the set of types that satisfy finpos.

Definition 3.2. A finitary positive *container* type is a list of pairs made of a Primrose type and a natural number.

Containers are a normal form for the inductive types that Primrose can operate on. A container has a number of constructors, each consisting of an arbitrary type for data and a finite number of children. One can *fill* a container type $F = (A_i, n_i)_{1 \le i \le k}$ with a type B. We define this filling

$$F\ B := \sum_{i=1}^{k} A_i \times B^{n_i}$$

Containers are endofunctors on types. I define the map operation on a container as follows, where $f^k : A^k \to B^k$ is the action of the $(-)^k$ functor.

$$\begin{aligned}
&\mathsf{map} \quad : (A \to B) \to F\ A \to F\ B \\
&\mathsf{map}\ f\ x := \mathsf{case}\ x\ \mathsf{of}\ \{L_i\ p \Rightarrow \langle \mathsf{fst}\ p, f^{n_i}\ (\mathsf{snd}\ p) \rangle\}
\end{aligned}$$

I define a second utility function called getChildren. Given a filled container, this empties the filling into a list, replacing each filled value with its index into the list. The value n_i is the number of filled values for case i, and iota_k is the tuple $0, \ldots, k-1$.

$$\begin{aligned}
&\mathsf{getChildren} \quad : F\ A \to \mathsf{nat} \times A^* \times F\ \mathsf{nat} \\
&\mathsf{getChildren}\ x := \mathsf{case}\ x\ \mathsf{of}\ \{L_i\ p \Rightarrow \langle n_i, \mathsf{snd}\ p, \langle \mathsf{fst}\ p, \mathsf{iota}_{n_i} \rangle \rangle\}
\end{aligned}$$

Definition 3.3. Two Primrose types A and B are *isomorphic* when there are Primrose terms $\vdash f : A \to B$ and $\vdash f^{-1} : B \to A$ such that

$$(\lambda x.f^{-1}\ (f\ x)) =^{\beta\eta} (\lambda x.x)$$
$$(\lambda y.f\ (f^{-1}\ y)) =^{\beta\eta} (\lambda y.y)$$

Type isomorphisms describe when two types contain the same amount of information. I use isomorphisms to convert finite positive types into containers.

Lemma 3.4. *Whenever A* finpos *X there exists a container $\bar{A}$ such that $\bar{A}\ X$ is isomorphic to A. Denote the forward direction from $\bar{A}\ X$ to A as* unroll *and the reverse as* roll.

This lemma is what defines the structure of finpos. The type variable X cannot appear on either side of a function arrow, because there is no guarantee the type will be isomorphic to a container. Similarly X cannot be used within other inductive types, as the inductive type could mean that a constructor can use X a variable number of times.

Example 3.5. Consider the type of binary trees, where all leaves and internal nodes are tagged with a natural.

$$T := \mu X.\langle \mathtt{D}\colon \mathsf{nat}, \mathtt{S}\colon \{\mathtt{L}\colon \langle\rangle, \mathtt{B}\colon \langle \mathtt{L}\colon X, \mathtt{R}\colon X\rangle\}\rangle$$

This type has the corresponding container $\bar{T} := [(\mathsf{nat}, 0), (\mathsf{nat}, 2)]$. The first case corresponds to leaves, and the second case corresponds to branches.

Container types provide enough structure to use Longley and Normann's [16] encoding for finitary positive inductive types.

$$[\![\mu X.A]\!] := \mathsf{nat} \times \mathsf{nat} \times [\![\bar{A}\ \mathsf{nat}]\!]^*$$

$$\begin{aligned}
[\![\sim t]\!] := \ & \mathsf{let}\ \langle k, cs, t'\rangle = \mathsf{getChildren}\ (\mathsf{roll}\ [\![t]\!])\ \mathsf{in} \\
& \mathsf{let}\ \mathsf{offset} = \lambda i, x.1 + i + k * x\ \mathsf{in} \\
& \langle\ 1 + \mathsf{fst}\ (\mathsf{primrec} \\
& \quad \langle 0, 0\rangle \\
& \quad (\lambda\langle x, i\rangle.\langle \mathsf{max}\ x\ (\mathsf{depth}\ (cs\ i)), 1 + i\rangle) \\
& \quad k) \\
& , 0 \\
& , \mathsf{cons}\ (\mathsf{map}\ (\lambda i.\mathsf{offset}\ i\ (\mathsf{root}\ (cs\ i)))\ t') \\
& \quad (\lambda n. \\
& \quad\quad \mathsf{let}\ \langle q, r\rangle = \mathsf{divmod}\ n\ k\ \mathsf{in} \\
& \quad\quad \mathsf{map}\ (\mathsf{offset}\ r)\ (\mathsf{heap}\ (cs\ r)\ q)) \\
& \rangle \\
[\![!t]\!] := \ & \mathsf{let}\ \langle d, r, h\rangle = [\![t]\!]\ \mathsf{in} \\
& \mathsf{unroll}\ (\mathsf{map}\ (\lambda i.\langle d, i, h\rangle)\ (h\ r)) \\
[\![\mathsf{fold}\ e\ \mathsf{as}\ x\ \mathsf{by}\ t]\!] := \ & \mathsf{let}\ \langle d, r, h\rangle = [\![e]\!]\ \mathsf{in} \\
& \mathsf{primrec} \\
& \quad (\lambda i.\mathsf{arb}) \\
& \quad (\lambda f, i. \\
& \quad\quad \mathsf{let}\ x = \mathsf{unroll}\ (\mathsf{map}\ f\ (h\ i))\ \mathsf{in} \\
& \quad\quad [\![t]\!]) \\
& \quad d\ r
\end{aligned}$$

Inductive types are encoded as a triple. The last component of the triple acts as a heap. Every entry stores a constructor, a value of that constructor's data, and a pointer for each of its children. By construction, the heap cannot be cyclic, thus it stores a tree. The first component is an upper bound on the depth of the tree. The second indicates the root node of this tree.

The roll operation is most complex. The operation merges the heaps of all the recursive terms together, and then adds an extra entry for the new root. The heaps are merged together by striping; assuming k children, the first child heap occupies indices $1 + k * i$, the second $2 + k * i$ and so on.

In detail, it finds the recursive components of the input value using getChildren. It then defines a helper function offset which calculates the new heap indices for child i. The depth of the returned tree is the maximum depth of all children, plus one for the additional level of indirection. The new root is at index 0 which is the front of the heap. The root of the new heap updates the result of getChildren so each pointer refers to the root of the child tree. The rest of the heap uses divmod to calculate which child and what heap entry to access—the remainder is the child and the quotient the index. Applying offset to all pointers in that value updates the pointers to access the new heap.

The unroll operation is straightforward thanks to storing the root of the tree. For every pointer it takes a copy of the initial tree and replaces the root with that pointer.

The fold operation works across the whole heap in parallel. First an arbitrary map of values is created. Then for each layer in the tree it replaces all the pointers into the heap with values from the last layer's value map, and then it applies the accumulator expression. Iterating on the depth stored with the heap ensures that every layer has been reduced.

Example 3.6. Continuing example 3.5, we calculate

$$[\![T]\!] = \mathsf{nat} \times \mathsf{nat} \times (\mathsf{nat} \to (\mathsf{nat} \times \langle\rangle) + (\mathsf{nat} \times \langle \mathsf{nat}, \mathsf{nat}\rangle))$$

Figure 5 contains both the diagram of a tree and a box-and-pointer representation of its encoded value.

Trees stored with this encoding are restricted to a constant number of children for each type of node. I believe it possible to allow trees with nodes that can have a variable but finite number of children. This would require changing the definitions for container, map, getChildren, roll and unroll. The details of this encoding have not been fully worked out. In the Primrose type system, allowing these branching structures would amount to modifying the finpos condition for inductive types. Instead of rejecting occurrences of the variable inside of an inductive type, it would require occurrences to be finpos too.

3.2 Future Work

Work on Primrose is ongoing, and there are many important theoretical and practical issues to address. One problem is

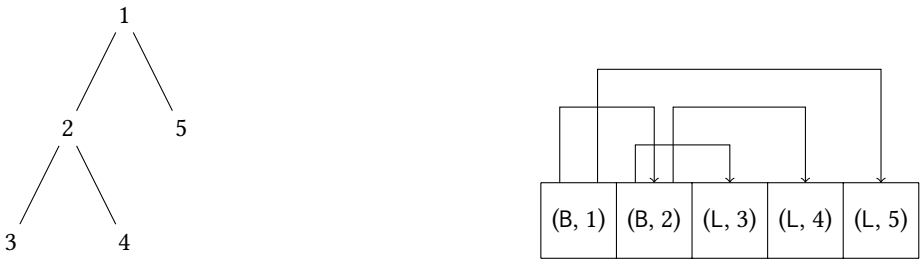

Figure 5. Representation of a tree (left) and its System T encoding (right)

proving that the translation into System T preserves reduction. Longley and Normann [16] only give a sketch of why their encoding of inductive types works, whilt the reduction rules for unions only hold when considering System T with eta conversion for functions.

Another challenge is to convert my existing System T self-reducer into a Primrose program. As a dependently typed language, Idris guarantees that my existing term is well-typed. However, Idris also has polymorphism, allowing me to reuse several definitions. As Primrose is only simply typed, this could lead to a lot of duplicated code. Alternatively, I could investigate extending Primrose with prenex polymorphism as in Standard ML.

A practical issue whilst working with Primrose is that there is currently no interpreter, making testing programs impossible. I have written a type checker for Primrose in Idris, but inductive types interact poorly with Idris's requirement that all programs terminate. One potential workaround I have is to compile Primrose into Scheme.

As a final test of the practicality of Primrose, I intend to write a fuelled self-reducer. As Primrose embeds System T, there is a Primrose program converting System T terms into Primrose terms. The translation function also acts via structural induction, so in theory it may be possible to write this in Primrose too. Composing these three programs, I can take a System T term, convert it to Primrose, reduce it, and then translate it back to System T. Thus I could have a second self-reducer for System T. This could be compared with my existing self-reducer.

4 Related Work

Self-reducers. Table 1 lists self-reducers for a number of systems. They have different type systems and recursive power. Whilst they use different quotation functions, each embeds terms as some form of abstract syntax tree.

The results in the 80's and early 90's focus on establishing self-reducers for practical languages [13, 10, 4, 15]. In parallel, Mogensen [18], and Berarducci and Böhm [3] worked on self-reducers for the λ-calculus; the first self-reducers for languages with first-class functions.

There has recently been a series of papers on interpreters and reducers for more minimal strongly-typed languages by Brown and Palsberg [8, 6, 9, 7]. They use a calculus dubbed $\mathrm{F}^{\mu i}_{\omega}$, consisting of System F with type-level functions, iso-recursive types, and intensional type functions. The language is Turing complete.

Typed Representation. My definition of quotation function (definition 2.3) uses an untyped representation of terms. All terms of the object language are embedded into the same meta type E. It also allows for values representing ill-typed terms, and possibly even values representing no term at all.

An alternative is *typed representation*. Instead of a single meta type for program codes, there is a Ty^O-indexed-family $E(-) : \mathsf{Ty}^O \to \mathsf{Ty}^M$ of types for program codes. Using multiple types, one can enforce that all values of type $E(A)$ represent object terms in $\mathsf{Tm}^O(A)$. This approach is used by Brown and Palsberg [8, 6, 9, 7] to define their self-reducers and self-evaluators.

Self-Evaluators. Both self-reducers and self-evaluators are colloquially known as self-interpreters. Consider the definition of a self-evaluator:

Definition 4.1. A *self-evaluator* relative to a quotation function $\ulcorner - \urcorner$ is a type-indexed family of programs $\mathsf{u}_A : \mathsf{Tm}(E \to A)$ such that for all terms $\mathsf{e} \in \mathsf{Tm}(A)$

$$\mathsf{u}_A \ulcorner \mathsf{e} \urcorner \rightsquigarrow^* \mathbf{nf}(\mathsf{e})$$

A self-evaluator (u for unquote) takes embeddings of terms and behaves the same as the original term. Many self-reducers implement self-evaluators [15, 18, 12, 7].

Many programming languages also contain self-evaluators as primitive operations. This includes JavaScript, Scheme and Python among others.

Conversion for dependent type theories. Current work on dependent type theory asks whether conversion checking is computable within a dependently-typed language [1]. Conversion checking often involves reducing terms to head-normal form; this can be extended to reduce terms to beta-normal form. Brown and Palsberg [8] constructed a self-reducer for System U, an inconsistent dependently-typed calculus. It is an open problem if a consistent dependently-typed language has a self-reducer.

References

[1] A. Abel, J. Öhman, and A. Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2, POPL, 23:1–23:29. DOI: 10.1145/3158111.

[2] A. Bauer. On self-interpreters for System T and other typed λ-calculi. Draft, (Jan. 2016). Retrieved Sept. 25, 2024 from https://math.andrej.com/wp-content/uploads/2016/01/self-interpreter-for-T.pdf.

[3] A. Berarducci and C. Böhm. 1992. A self-interpreter of lambda calculus having a normal form. In *Computer Science Logic, 6th Workshop, CSL '92, San Miniato, Italy, September 28 - October 2, 1992, Selected Papers* (Lecture Notes in Computer Science). E. Börger, G. Jäger, H. K. Büning, S. Martini, and M. M. Richter, (Eds.) Vol. 702. Springer, 85–99. DOI: 10.1007/3-540-56992-8_7.

[4] A. Bondorf. 1989. A self-applicable partial evaluator for term rewriting systems. In *TAPSOFT'89: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages (CCIPL)* (Lecture Notes in Computer Science). J. Díaz and F. Orejas, (Eds.) Vol. 352. Springer, 81–95. DOI: 10.1007/3-540-50940-2_29.

[5] E. C. Brady. 2021. Idris 2: quantitative type theory in practice. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)* (LIPIcs). A. Møller and M. Sridharan, (Eds.) Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:26. DOI: 10.4230/LIPICS.ECOOP.2021.9.

[6] M. Brown and J. 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*. R. Bodík and R. Majumdar, (Eds.) ACM, 5–17. DOI: 10.1145/2837614.2837623.

[7] M. Brown and J. Palsberg. 2018. Jones-optimal partial evaluation by specialization-safe normalization. *Proc. ACM Program. Lang.*, 2, POPL, 14:1–14:28. DOI: 10.1145/3158102.

[8] M. Brown and J. Palsberg. 2015. Self-representation in Girard's System U. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. S. K. Rajamani and D. Walker, (Eds.) ACM, 471–484. DOI: 10.1145/2676726.2676988.

[9] M. Brown and J. 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*. G. Castagna and A. D. Gordon, (Eds.) ACM, 415–428. DOI: 10.1145/3009837.3009853.

[10] D. A. Fuller and S. Abramsky. 1988. Mixed computation of Prolog programs. *New Gener. Comput.*, 6, 2&3, 119–141. DOI: 10.1007/BF03037134.

[11] C. B. Jay and J. Palsberg. 2011. Typed self-interpretation by pattern matching. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*. M. M. T. Chakravarty, Z. Hu, and O. Danvy, (Eds.) ACM, 247–258. DOI: 10.1145/2034773.2034808.

[12] N. D. Jones, C. K. Gomard, and P. Sestoft. 1993. *Partial evaluation and automatic program generation. Prentice Hall international series in computer science*. Prentice Hall. ISBN: 978-0-13-020249-9.

[13] N. D. Jones, P. Sestoft, and H. Søndergaard. 1985. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. Tech. rep. 1. University of Copenhagen.

[14] O. Kiselyov. 2022. Simply-typed encodings: PCF considered as unexpectedly expressive programming language. Retrieved Sept. 16, 2024 from https://www.okmij.org/ftp/Computation/simple-encodings.html.

[15] J. Launchbury. 1991. A strongly-typed self-applicable partial evaluator. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings* (Lecture Notes in Computer Science). J. Hughes, (Ed.) Vol. 523. Springer, 145–164. DOI: 10.1007/3540543961_8.

[16] J. Longley and D. Normann. 2015. *Higher-Order Computability. Theory and Applications of Computability*. Springer. ISBN: 978-3-662-47991-9. DOI: 10.1007/978-3-662-47992-6.

[17] H. Makholm. 2000. On Jones-optimal specialization for strongly typed languages. In *Semantics, Applications, and Implementation of Program Generation, International Workshop SAIG 2000, Montreal, Canada, September 20, 2000, Proceedings* (Lecture Notes in Computer Science). W. Taha, (Ed.) Vol. 1924. Springer, 129–148. DOI: 10.1007/3-540-45350-4_11.

[18] T. Æ. Mogensen. 1992. Efficient self-interpretations in lambda calculus. *JFP*, 2, 3, 345–363. DOI: 10.1017/S0956796800000423.

[19] M. Naylor. 2008. Evaluating Haskell in Haskell. *The Monad.Reader*, 10, 25–33. Retrieved Oct. 1, 2024 from http://www.haskell.org/wikiupload/0/0a/TMR-Issue10.pdf.

A Additional Figures

$A, B ::=$	types
$\quad X$	variables
$\mid A \to B$	functions
$\mid \langle L : A, \ldots, L : A \rangle$	products
$\mid \{L : A, \ldots, L : A\}$	sums
$\mid \mu X.A$	inductive types
$e, f, t, u ::=$	terms
$\quad x$	variables
$\mid \mathsf{let}\ x = e\ \mathsf{in}\ t$	let bindings
$\mid \lambda x.t$	abstraction
$\mid f\ t$	application
$\mid \langle L : t, \ldots, L : t \rangle$	tuples
$\mid e.L$	projection
$\mid L\ t$	injection
$\mid \mathsf{case}\ e\ \mathsf{of}\ \{L\ x \Rightarrow t; \ldots; L\ x \Rightarrow t\}$	case splitting
$\mid \sim t$	rolling
$\mid\ !e$	unrolling
$\mid \mathsf{fold}\ e\ \mathsf{as}\ x\ \mathsf{by}\ t$	folding

Figure 6. Syntax of Primrose

$$\frac{}{X \text{ finpos } X} \qquad \frac{X \notin \mathsf{FV}(A) \qquad X \notin \mathsf{FV}(B)}{A \to B \text{ finpos } X} \qquad \frac{A_i \text{ finpos } X}{\langle L_1 : A_1, \ldots, L_n : A_n \rangle \text{ finpos } X} \qquad \frac{A_i \text{ finpos } X}{\{L_1 : A_1, \ldots, L_n : A_n\} \text{ finpos } X} \qquad \frac{X \notin \mathsf{FV}(A)}{\mu Y.A \text{ finpos } X}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\ t : B}$$

$$\frac{\Gamma \vdash e : A \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ t : B} \qquad \frac{\Gamma \vdash e : \langle L_1 : A_1, \ldots, L_n : A_n \rangle}{\Gamma \vdash e.L_i : A_i} \qquad \frac{\Gamma \vdash t_i : A_i}{\Gamma \vdash \langle L_1 : t_1, \ldots, L_n : t_n \rangle : \langle L_1 : A_1, \ldots, L_n : A_n \rangle}$$

$$\frac{\Gamma \vdash t : A_i}{\Gamma \vdash L_i\ t : \{L_1 : A_1, \ldots, L_n : A_n\}} \qquad \frac{\Gamma \vdash e : \{L_1 : A_1, \ldots, L_n : A_n\} \qquad \Gamma, x_i : A_i \vdash t_i : B}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{L_1\ x_1 \Rightarrow t_1; \ldots; L_n\ x_n \Rightarrow t_n\} : B}$$

$$\frac{\Gamma \vdash t : A[X/\mu X.A] \qquad A \text{ finpos } X}{\Gamma \vdash {\sim}t : \mu X.A} \qquad \frac{\Gamma \vdash e : \mu X.A \qquad A \text{ finpos } X}{\Gamma \vdash !e : A[X/\mu X.A]} \qquad \frac{\Gamma \vdash e : \mu X.A \qquad \Gamma, x : A[X/B] \vdash t : B \qquad A \text{ finpos } X}{\Gamma \vdash \mathsf{fold}\ e\ \mathsf{as}\ x\ \mathsf{by}\ t : B}$$

Figure 7. Typing rules of Primrose

Table 1. A taxonomy for self-reducers

System	Year	Type Complexity	Functions	Recursive Power	Fuelled
Jones et al. [13]	1985	Untyped	Second Class	General	No
Fuller and Abramsky [10]	1988	Untyped	Second Class	General	No
Bondorf [4]	1989	Untyped	Second Class	General	No
Launchbury [15]	1991	Simple	Second Class	General	No
Mogensen [18]	1992	Untyped	First Class	General	No
Berarducci and Böhm [3]	1992	Untyped	First Class	General	No
Makholm [17]	2000	Simple	Second Class	General	No
Naylor [19]	2008	Polymorphic	First Class	General	No
Jay and Palsberg [11]	2011	Polymorphic	First Class	General	No
Brown and Palsberg [9]	2017	Polymorphic	First Class	General	No
Brown and Palsberg [7]	2018	Polymorphic	First Class	General	No
This work	2024	Simple	First Class	Bounded	Yes