Typed Self-Interpretation by Pattern Matching

Barry Jay
University of Technology, Sydney
Barry.Jay@uts.edu.au

Jens Palsberg
University of California, Los Angeles
palsberg@ucla.edu

Abstract
Self-interpreters can be roughly divided into two sorts: self-recognisers that recover the input program from a canonical representation, and self-enactors that execute the input program. Major progress for statically-typed languages was achieved in 2009 by Rendel, Ostermann, and Hofer who presented the first typed self-recogniser that allows representations of different terms to have different types. A key feature of their type system is a type:type rule that renders the kind system of their language inconsistent.

In this paper we present the first statically-typed language that not only allows representations of different terms to have different types, and supports a self-recogniser, but also supports a self-enactor. Our language is a factorisation calculus in the style of Jay and Given-Wilson, a combinatory calculus with a factorisation operator that is powerful enough to support the pattern-matching functions necessary for a self-interpreter. This allows us to avoid a type:type rule. Indeed, the types of System F are sufficient. We have implemented our approach and our experiments support the theory.

Categories and Subject Descriptors D.3.4 [Processors]: Interpreters; D.2.4 [Program Verification]: Correctness proofs, formal methods; F.3.2 [Semantics of Programming Languages]: Operational semantics

General Terms Languages, Theory

Keywords self-interpretation, pattern matching

1. Introduction
An interpreter implements a programming language, and a self-interpreter is an interpreter in the language that it implements. Self-interpreters are popular and available for Standard ML [34], Haskell [28], Scheme [1], JavaScript [12], Python [33], Ruby [41], λ-calculus [2, 4, 5, 21, 25, 26, 32, 35], and many other languages [23, 38, 42]. A self-interpreter enables programmers to easily modify, extend, and grow a language [31], do other forms of meta-programming [8], and even derive an algorithm for normalisation by evaluation [6].

These self-interpreters can be roughly divided into two sorts: self-recognisers that recover the input program from a canonical representation, and self-enactors that execute the input program. While we will review the rich literature on self-interpretation in sections 2 and 9, two highlights are papers by Mogensen [25], and by Berarducci and Bohm [5], that each defined a single program representation for an untyped language that supports both a self-recogniser and a self-enactor, with proofs of correctness.

Now consider statically typed languages. Most previous work on self-interpreters for these give all program representations the same, universal type. The use of a universal type ignores the type of the input program and thereby misses an important opportunity for static type checking of self-interpreters. Major progress was achieved in 2009 when Rendel, Ostermann, and Hofer [31] presented the first self-recogniser for a statically-typed language in which representations of different terms can have different types.

The challenge Rendel, Ostermann, and Hofer left open the problem of typing a self-enactor. Additionally, their type system has a type:type rule that renders the kind system of their language inconsistent.

Our results In this paper we present the first statically-typed language in which representations of different terms can have different types and in which we can program and statically type both a self-recogniser and a self-enactor. Our language uses System F types and so has no rule asserting type:type. Our approach differs from previous work by adopting a pattern-matching perspective that we summarise next.

The representation of a term is a data structure so it is natural to consider an interpreter as a pattern-matching function in which each evaluation rule left → right of the source language can be represented by a case, with a pattern derived from left and a body derived from right.

So the fundamental question becomes how to represent pattern-matching. Berarducci and Bohm achieved this by considering how to solve equations inside lambda calculus but now there is a more direct and powerful method, in the pattern calculus of Jay and Kesner [16, 18, 19]. Although it may be possible to achieve our main goals in pure pattern calculus (or even static pattern calculus), this paper adopts a simpler and more direct approach.

Recent work on factorisation calculus by Jay and Given-Wilson [17] supports combinatory calculi that are more expressive than traditional combinatory calculi (based on S and K), in being able to analyse the internal structure of any normal form, e.g. to recover X from SKX. More generally, they can define pattern-matching functions that are powerful enough to interpret themselves. This compares well with previous approaches in which functions at one level are analyzed by functions at the next higher level, as in the higher-order polymorphic λ-calculus and even which adds the rule type:type.

The pure factorisation calculus [17] is a combinator calculus with just two operators, S and F, where S is known from SK-combinators, and F is a factorization operator that is able to decompose compounds (e.g. closed normal forms) into their components. In this paper we add a constructor to block evaluation.
and together $S$, $F$, and $B$ are sufficient to represent programs in an untyped manner and to define an untyped self-recogniser and an untyped self-enactor. An additional benefit of this approach is that there is a term that decides equality of representations of combinators (closed terms). By comparison, such a term is not known to exist for higher-order abstract syntax of untyped $\lambda$-calculus when the meta-language is the untyped $\lambda$-calculus itself.

We meet the goals of static type checking by adding three more operators, the traditional operator $K$, a fixpoint operator $Y$ and an operator $E$ that tests for equality of operators. Although the types are relatively simple, being those of System $F$ they are used in two unusual ways. First, the $F$ operator cannot be typed with Hindley-Milner types alone. Rather, it takes an argument of polymorphic type, since the types of components are not determined by the type of their compound. Second, the operator $E$ doesn’t have a principal type scheme, for reasons that can be traced back to the typing of pattern-matching functions in pattern calculus. This creates difficulties when interpreting $E$ itself, which are overcome by replacing explicit references to $E$ in patterns by binding symbols which are shown to match with $E$ only.

We have implemented the entire approach and performed experiments that support our theory.

The rest of this paper. We will discuss the nature of self-interpretation and closely related work (Section 2), and we will define our language (Section 3), syntactic sugar (Section 4), self-recogniser (Section 5), self-enactor (Section 6), type system (Section 7), including proofs that our self-interpreters type check, and experimental results (Section 8). We also discuss additional related work (Section 9).

2. The Nature of Self-Interpretation

This section fixes the terminology for the various sorts of self-interpreters to be considered. Definitions have been chosen so that they apply as widely as possible, i.e. both to $\lambda$-calculi and other rewriting systems, and to programming languages with their evaluation strategies. Since the calculi emphasise static interpreters, while the programming language community emphasise dynamic ones, we propose new names for the various special cases.

Self-interpretation involves two steps. The first is a process of quotation that transforms the syntax of a term $t$ into a value or normal form $\text{quote}(t)$ ready for interpretation. The second is the application of a self-interpretor to $\text{quote}(t)$ to produce something that has the same meaning as $t$. Researchers have identified two sorts of meaning here: a static approach that focuses on program structure, and a dynamic approach that focuses on program behaviour. Let us examine each of them in turn.

A self-recogniser is a self-interpreter unquote that can recognise a term from its quotation, by reversing the quotation process:

**Self-Recogniser:** \[ \text{unquote}(t) \approx t \] (1)

where $\approx$ denotes *behavioural equivalence*. For example, quotation may add tags that block evaluation, which are then removed by unquote. The first self-recogniser is due to Kleene [21] who introduced a notion of quotation and unquote for pure $\lambda$-calculus, and established (1) as a consequence of $\beta$-equality. Barendregt [2] cited Kleene’s paper and used the term *self-interpretor* for any $\lambda$-term unquote that satisfies $\text{unquote}(t) =_\beta t$. Mogensen [25, 26], Berarducci and Böhm [5], and Bel [4] all presented $\lambda$-terms unquote that satisfy Equation (1). The name *self-recognisers* seems apt because the interpretation can recover, or recognise, (something equivalent to) the original term.

The process of quotation tends to cause code expansion, with $\text{unquote}(t)$ being a much larger program than $t$. This problem is mitigated if the self-recogniser is strong in the sense that:

**Strong Self-Recogniser:** \[ \text{unquote}(t) \longrightarrow^* t \]

where $\longrightarrow^*$ denotes reduction in a calculus, or a small-step operational semantics of a programming language. Among the examples above, those of Mogensen [25, 26] and Berarducci and Böhm [5] are strong.

A self-enactor is a self-interpreter enact that mimics evaluation:

**Self-Enactor:** \[ t \Rightarrow^* v \implies \text{enact}(t) \approx 'v' \] (2)

That is, if $t$ evaluates to a value $v$ then enact($t$) is behaviourally equivalent to $'v$'. Note that this account requires knowledge of the values $v$ and the evaluation process $\Rightarrow^*$. Also, a self-enactor cannot be a self-recogniser unless $'v' \approx v$ which usually fails. That is, preservation of meaning by a self-enactor is different to that of a self-recogniser. That said, a self-enactor enact can be combined with a self-recogniser unquote to produce a self-interpreter that maps $t$ to unquote(enact($t$)).

The first use of self-enactors is due to Mogensen [25] who proved the existence of such a term in pure $\lambda$-calculus. He used the term self-reducer for any $\lambda$-term enact that satisfies Equation (2), Mogensen [25], Berarducci and Böhm [5] (who preferred the term reductor), and Song, Xu, and Qian [35] presented $\lambda$-terms enact that satisfy Equation (2). The name self-enactor seems apt because Equation (2) implies that enact must do work to turn a quotation into action. The technical details of Mogensen’s paper [25] strongly suggest that a self-enactor is more complex than a self-recogniser. This makes sense since, unlike a self-recogniser, a self-enactor must do actual evaluation.

A strong self-enactor satisfies the following stronger requirement:

**Strong Self-Enactor:** \[ t \Rightarrow^* v \implies \text{enact}(t) \Rightarrow^* 'v' \]

Among the examples already mentioned, only that of Berarducci and Böhm is a strong self-enactor.

The self-interpreters for Standard ML [34], Haskell [28], Scheme [1], Javascript [12], Python [40], and Ruby [41] all do evaluation. The documentations suggest that the programmers intended them to be self-enactors or, in some case, a self-enactor followed by an application of a self-recognizer to print a value rather than a representation of a value.

A desirable quality of program representation is that we can decide equality of program representations by a term equal, as in:

**Equality of Representations:**

\[
\text{equal}(s, t) = \begin{cases} 
\text{true} & \text{if } s = 't' \\
\text{false} & \text{otherwise} 
\end{cases}
\]

Some quotation mechanisms support that; some don’t. Even the presence of unquote doesn’t guarantee the existence of equal. For example, many of the above papers represent $\lambda$-terms by higher-order abstract syntax with a meta-language that itself is the untyped $\lambda$-calculus. For such representations, no term equal is known to exist. Binding operations complicate the issues because in $\lambda$-calculus, closed terms are built from open terms. Kleene [21] avoided the problems with open terms by representing programs with only closed terms that are built from combinators, that is, small, closed $\lambda$-terms. In general, combinatory calculi present an easier equality-checking problem.

3. Blocking Factorisation Calculus

3.1 Overview

Our language is a combinatory calculus called the blocking factorisation calculus which has all the properties above: decidable equal-
ity of quotations, a self-recogniser and a self-enactor. It is a factori-

sation calculus in the sense of Jay and Given-Wilson [17]. Factori-
sation calculi are more expressive than traditional SK-combinators
[17], which may seem surprising since SK-calculus is combinator-
torially complete [20]. However, its factorisation operator $F$ can
decompose an identity function $SKX$ to recover the value of the
combinator $X$, something that cannot be done using $S$ and $K$
onely. Note that the corresponding logic would be unsound if $F$
could decompose an arbitrary application, e.g. to recover $X$ from
$KKX$. To ensure soundness, the reduction rules for $F$ require its
first argument to be factorable, that is, a partial application of an
operator. As a result, the factorisation calculus is confluent, which
implies soundness of the corresponding logic. This expressive power
supports analysis of normal forms such as quotations, including
decidable equality of program representations. More generally, it
supports pattern matching that is expressive enough to support self-
interpreters.

3.2 Syntax

We assume a countable set of variables (meta-variables $w, x, y, z$). The
operators (meta-variable $O$) of our language are given by:

$$(\text{Operator}) \quad O ::= Y \mid K \mid S \mid F \mid E \mid B .$$

Each operator has an arity, given by 1, 2, 3, 3, 4 and $\infty$, respec-
tively, that helps us define the intended semantics. $S$ and $K$ take
their usual meanings from combinatory logic. $Y$ is a fixed-point
operator. In a pure setting it could be defined from $S$ and $K$ but this
interpretation will not support the typing. $F$ is the factorisation op-
erator used to decompose compounds, as described later. Perhaps
surprisingly, it cannot be defined in terms of $S$ and $K$ [17]. $E$ is an
equality operator that takes four arguments; two to compare, and two
alternative results, chosen according to whether the compared argu-
ments are the same operator or not. $B$ is used to block evalua-
tion.

The terms (meta-variables $p, q, r, s, t, u$) of our term calculus
are given by:

$$(\text{Term}) \quad t ::= x \mid O \mid t \ t .$$

Terms are generated by the variables and operators and they are
closed under application.

The free variables of a term are all the variables that occur in the
term, since there are no binding operations. A term $t$ avoids a
variable $x$ if $x$ is not a free variable of $t$. Term substitutions $\sigma$
are defined and applied in the usual manner. The substitution of the
term $n_i$ for the variable $x_i$ for $1 \leq i \leq n$ in the term $t$ may be
denoted $[u_1/x_1, u_2/x_2, \ldots, u_n/x_n]$.

A combinator is a term built without any variables, the collect-
ion of which forms the corresponding combinatory calculus [15].
Conceptually, the combinatoric calculus is more fundamental than
the term calculus, but to define operations such as pattern-matching
on combinators requires the larger, term calculus, so that our efforts
will focus there.

A term is factorable if it is a partial application of an operator, as
determined by its arity. That is, $O \ t_1 \ldots t_k$ is a partial applica-
tion of operator $O$ if $k$ is strictly less than the arity of $O$. In particular,
all operators are factorable. A compound is a factorable application.

<table>
<thead>
<tr>
<th>$t \rightarrow t'$</th>
<th>$r \rightarrow r'$</th>
<th>$u \rightarrow u'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \rightarrow t'$</td>
<td>$r \ u \rightarrow r' \ u$</td>
<td>$r \ u \rightarrow r' \ u$</td>
</tr>
</tbody>
</table>

Figure 1. The Reduction Relation

3.3 Reduction Semantics

The reduction rules are:

$$Y \ t \rightarrow t \ (Y \ t)$$
$$K \ s \ t \rightarrow s$$
$$S \ s \ t \ u \rightarrow s \ u \ (t \ u)$$
$$F \ O \ s \ t \rightarrow s \ \text{if } O \text{ is an operator}$$
$$F \ (p \ q) \ s \ t \rightarrow t \ p \ q \ \text{if } p \ q \text{ is a compound}$$
$$E \ O \ O \ s \ t \rightarrow s \ \text{if } O \text{ is an operator}$$
$$E \ p \ q \ s \ t \rightarrow t \ \text{otherwise, if } p \text{ and } q \text{ are factorable.}$$

That is, $Y$ is the fixed-point operator, $K$ eliminates its second
argument, and $S$ duplicates its third argument, all as usual. The
factorisation operator $F$ branches according to the value of its first
argument. If this is a compound then apply the third argument to the
components, else return the second argument. The equality operator
$E$ decides equality of operators. If the first two arguments are the
same operator then return the third argument, else if the first two
arguments are both factorable (whether equal or not) then return
the fourth argument. Note that there are no reduction rules for $B$,
which may thus be thought of as a constructor.

The reduction relation $\rightarrow$ is obtained by applying the reduc-
tion rules to arbitrary sub-terms, as described in Figure 1. As usual
in rewriting, the transitive closure of a relation is denoted by $(-)^*$
as in $\rightarrow^*$ and the reflexive, transitive closure by $(-)^*\rightarrow$ as in $\rightarrow^*$. If $t \rightarrow^* t'$ then $t$ reduces to $t'$.

The basic properties of the calculus are easily established.

**Theorem 3.1.** Reduction is confluent.

**Proof.** If $p \ q$ is a compound then, by inspecting the arities, it is
clear that it is not an instance of any reduction rule. Hence, any
reduction of $p \ q$ is a reduction of $p$ or of $q$ which implies that there
are no critical pairs [37] involving $F \ (p \ q) \ s \ t$. Similarly, there are
no other critical pairs. $\square$

**Theorem 3.2.** Every combinator in normal form is factorable.

**Proof.** The proof is by induction on the structure of the combinator.
For example, if it is of the form $F \ p \ s \ t$ then induction implies $p$ is a
factorable form, so that a reduction rule applies. Similar remarks
apply to combinators of the form $E \ p \ q \ s \ t$. The other cases are
straightforward. $\square$

This theorem provides a form of progress property, in that evalu-
ation of the operators, especially $F$ and $E$, cannot get blocked.
Note, however, that if $\Omega$ does not have a normal form then
$F \ \Omega \ K \ K \ ω$ cannot become an instance of a rule.

4. Syntactic Sugar

For the purpose of practical programming, particularly of our two
self-interpreters, we use five forms of syntactic sugar:

- identity operator, written $\mathbf{1}$,
- $\lambda$-abstraction, written $\lambda \ x . s$ and, in typewriter font, $\mathbf{x} \rightarrow \mathbf{s}$,
- let binding, written let $x = s$ in $t$,
- let rec binding, written let rec $x = t$ and
- extensions, written $p \rightarrow s \mid t$.

We de-sugar terms with such constructs before executing them. De-
sugaring maps closed terms to closed terms. The type-writer
4.1 Identity, $\lambda$-abstraction, let, and let rec

We de-sugar $I$ to the combinator $SKK$.

One of the oldest results on computability is that $\lambda$-abstraction can be defined by $SKK$-terms (e.g. [15]). The definition of $\lambda^*x.t$ is as follows:

\[
\begin{align*}
\lambda^*x.x &= I \\
\lambda^*x.t &= K t & \text{if } t \text{ avoids } x \\
\lambda^*x.t \| x &= t & \text{if } t \text{ avoids } x \\
\lambda^*x.(u \| v) &= S(\lambda^*x.u)(\lambda^*x.v) & \text{otherwise}.
\end{align*}
\]

The use of $\eta$-contraction in the third line above is not theoretically necessary but makes a big difference in the size of the resulting term.

**Lemma 4.1.** For all terms $s$ and $u$ and variable $x$ there is a reduction

\[(\lambda^*x.s)u \rightarrow^* u[x]/s.\]

**Proof.** The proof is by induction on the structure of the term $s$. If $s$ is $x$ then $(\lambda^*x.s)u = I u \rightarrow^* u = [u/x]s$. If $s$ avoids $x$ then $(\lambda^*x.s)u = K u \rightarrow^* u = [u/x]s$. Otherwise, if $s$ is of the form $t \| x$ where $t$ avoids $x$ then $(\lambda^*x.s)u = t u = [u/x]s$. Otherwise, if $s$ is of the form $s_1s_2$ then

\[
\begin{align*}
(\lambda^*x.s)u &= S(\lambda^*x.s_1)(\lambda^*x.s_2)u \\
&\rightarrow (\lambda^*x.s_1)((\lambda^*x.s_2)u) \\
&\rightarrow^* [u/x]s_1([u/x]s_2) \\
&= [u/x]s
\end{align*}
\]

by two applications of induction. 

We de-sugar the syntax $\text{let } x = u \text{ in } t$ to $(\lambda^*x.t)u$ and we de-sugar $\text{let rec } f = t \text{ to } Y(\lambda^*f.t)$, as usual.

4.2 Extensions

A pattern-matching function of the form

\[
\begin{align*}
p_1 &\rightarrow s_1 \\
p_2 &\rightarrow s_2 \\
&\ldots \\
p_n &\rightarrow s_n \\
| &\rightarrow s
\end{align*}
\]

can be built as a sequence of extensions of the form $p \rightarrow s \ | \ t$ by declaring the vertical bar to be right-associative, and replacing the final case by $\lambda^*x.s$. In such an extension, the first subterm $p$ is a pattern, which is a term in normal form. While generalisations are possible, our notion of extension is sufficient for typed self-interpretation, and already generalises the usual approaches to pattern matching in functional programming. Traditionally, pattern matching is a technique for destructing values of a given algebraic datatype, each pattern being headed by one of the type’s constructors. In contrast, we allow patterns such as $(y \ x)$ that is not headed by any constructor, but rather denotes an arbitrary compound data structure. Our notion of pattern matching is widely applicable; for example, it is easy to program an equality checker for normal forms.

We use the following recursive function to de-sugar extensions:

\[
\begin{align*}
x &\rightarrow s \ | \ r = \lambda^*x.r \\
O &\rightarrow s \ | \ r = \lambda^*x.E O x s (r x) = S(E\ O)(K\ s)\ r \\
p \ q &\rightarrow s \ | \ r = \lambda^*x.F x (r x) \\
\lambda^*y.(p \rightarrow (g \rightarrow s \ | \ r' y) \ | \ r') y \\
\text{(where } r' = \lambda^*y.\lambda^*z.\ y \ x = S(K\ r') \text{)}
\end{align*}
\]

where $x$ is chosen fresh. The first two rules are clear enough. The third defines matching against an applicative pattern $p q$ by matching the components of the argument against $p$ and then against $q$. The complexity of the term is caused by the need to handle the various sorts of match failure.

The intended semantics is given by defining matching. A match is either a successful match given by $\text{Some } \sigma$ where $\sigma$ is a substitution, or a match failure $\text{None}$. The disjoint union $\uplus$ of successful matches is the successful match obtained from the disjoint union of their substitutions, if this exists. All other disjoint unions are $\text{None}$. The matching $\{u/p\}$ of a pattern $p$ against a term $u$ is defined by the rules

\[
\begin{align*}
\{u/x\} &= \text{Some } [u/x] \\
\{O/O\} &= \text{Some } [] \\
\{u_1 u_2/p_1 p_2\} &= \{u_1/p_1\} \uplus \{u_2/p_2\} & \text{if } u_1 u_2 \text{ is factorable} \\
\{u/p\} &= \text{None} & \text{otherwise, if } u \text{ is factorable} \\
\{u/p\} &= \text{undefined} & \text{otherwise}
\end{align*}
\]

corresponding to those of static pattern calculus [16].

**Lemma 4.2.** Extensions satisfy the following derived reduction rules:

\[
\begin{align*}
(p \rightarrow s \ | \ r) u &\rightarrow^* \sigma s & \text{if} \{u/p\} = \text{Some } \sigma \\
(p \rightarrow s \ | \ r) u &\rightarrow^* r u & \text{if} \{u/p\} = \text{None}.
\end{align*}
\]

**Proof.** The proof is a routine induction on the structure of the pattern, given Lemma 4.1. 

This style of pattern matching, also known as path polymorphism [16, 18, 19], cannot be expressed in pure $\lambda$-calculus or even in a combinator calculus with the operators $Y$, $S$, $K$, $B$. So, our calculus has the operator $F$ and the novel operator $E$, and we make them play key roles when we de-sugar pattern matching.

For example, to unblock reduction we will use $\text{unblock}$ defined by

\[
\begin{align*}
\text{B} x &\rightarrow x \\
x &\rightarrow x
\end{align*}
\]

which de-sugars to the combinator

\[
\begin{align*}
B x &\rightarrow x \ | \ x \rightarrow x \\
&= \lambda^*x.F x (I x) (\lambda^*y.(B \rightarrow (x \rightarrow x) \ | \ r' y) \ | \ r') y \\
&= \lambda^*x.F x (I x) (\lambda^*y.(B \rightarrow I) \ | \ r' y) \\
&= \lambda^*x.F x (I x) (B \rightarrow I) \\
&= \lambda^*x.F x (I x) (S(S(E\ B)(K\ I))\ r') \\
&= S(F\ I)(K(S(S(E\ B)(K\ I))(S(K\ I))))
\end{align*}
\]

where $r' = S(K\ I)$.

5. Self-Recognisers

5.1 Quotation

Quotation for both our self-interpreters is given by

\[
\begin{align*}
\text{'}x\text{'} &= x \\
\text{'}O\text{'} &= BO \\
\text{'}(s \ t)\text{'} &= \text{'s't'}
\end{align*}
\]

Clearly, quoted terms are always normal forms, whose internal structure can be examined by factorising.

**Theorem 5.1.** There is a decidable equality of quotations of closed terms.

**Proof.** The booleans are given by $K$ (true) and $K\ I$ (false) as usual. The required equality term is
Let rec equal =
  x1 x2 -> (t y1 y2 -> (equal x1 y1) (equal x2 y2) (K I))
  | y -> K I
  | x -> | y -> E x y K (K I)

It decides equality of arbitrary closed normal forms, be they quotations or not. If applied to two compounds then it checks equality of both components (by applying the boolean (equal x1 y1) to (equal x2 y2) and (K I) to represent the conditional for conjunction). Alternatively, if the first argument is an operator then E is applied. □

5.2 A Self-Recogniser

Definite unquote by

let rec unquote =
  B x -> x
  | y x -> (unquote y) (unquote x)
  | x -> x

Theorem 5.2. unquote is a strong self-recogniser with respect to \( \Rightarrow^\ast \).

Proof. The proof is a straightforward induction on the structure of \( t \). If \( t \) is a variable or operator then \( \text{unquote}^\ast(t) \Rightarrow^\ast t \). If \( t \) is an application \( t_1 t_2 \), then \( 't \) is a compound \( 't_1 't_2 \) (since all quotations are headed by \( B \)). Hence \( \text{unquote}^\ast(t) = \text{unquote}^\ast('t_1 't_2) \Rightarrow^\ast 't_1 't_2 \) by two applications of induction. □

6. Self-Enactors

6.1 Evaluation

We choose a call-by-name semantics, given by an evaluation relation \( \Rightarrow \) as defined in Figure 2. There is not much scope for variation here: the operator \( F \) behaves like other branching constructs, such as conditionals, being eager in its first argument but deferring evaluation of the other two; and \( E \) evaluates its first two arguments to be a term that is irreducible with respect to \( \Rightarrow \). A term \( t \) has a value if there is a value \( v \) such that \( t \Rightarrow^\ast v \).

Usually, a context is described as a term with a hole in it but our terms contain free variables that cannot be bound, so a context \( C[\_] \) here must also allow a term substitution \( \sigma \) that is to be applied to the term that fills the hole.

Now, two terms \( t_1 \) and \( t_2 \) are behaviourally equivalent (written \( t_1 \approx t_2 \)) if, for any context \( C[\_] \), the term \( C[t_1] \) has a value if and only if \( C[t_2] \) does. The following lemma will be our main tool in establishing behavioural equivalence.

Lemma 6.1. If \( t_1 \Rightarrow^\ast t_2 \) then \( t_1 \approx t_2 \).

Proof. The proof is by case analysis on the reduction rules.

Before giving the self-enactor for the blocking factorisation calculus that we will type, the approach can be illustrated by a pair of simpler examples.

6.2 A Self-Enactor for \( SK \)-calculus

Consider the interpretation of \( SK \)-calculus in the blocking factorisation calculus. There are two natural approaches to the representation of a rule \( \text{left} \rightarrow \text{right} \) as a case, namely the reduction approach and the meta-circular approach (pace [32]). The reduction approach represents the rule by the case

\[
\text{left} \rightarrow \text{right}
\]

where the left- and right-hand sides of the rule have been quoted.

The meta-circular approach replaces ‘right’ above by an application of the operator that is the subject of the rule. For \( S \) this yields

\[
| B S x3 x2 x1 \rightarrow \text{enact} (x3 x1 (x2 x1))
\]

Although the meta-circular approach is sometimes more concise, and so will be preferred, it won’t always be applicable.

The interpretation of \( SK \)-calculus is thus given by the combinator \( \text{enact}SK \) defined by

\[
\begin{align*}
\text{let rec enactSK} &= \text{let enact1} = \\
& B K x2 x1 \rightarrow \text{enactSK} (K x2 x1) \\
& | B S x3 x2 x1 \rightarrow \text{enactSK} (S x3 x2 x1) \\
& | x1 \rightarrow x1 \\
& \quad \text{in} \\
& x2 x1 \rightarrow \text{enact1} (\text{enactSK} x2 x1) \\
& | x1 \rightarrow x1
\end{align*}
\]

The function \( \text{enact1} \) tries to perform one step of the evaluation. It is a pattern-matching function with one case for each reduction rule of the calculus, which then performs a recursive call to \( \text{enactSK} \). This stops if no reduction rule can be applied, as indicated by the default identity function. This handles partially applied operators. The pattern-matching function for \( \text{enactSK} \) itself has two cases: that for a compound \( \text{enact} \) the left-hand component and then reduces the whole by \( \text{enact1} \).

For example,

\[
\begin{align*}
\text{enactSK} & \quad ' (K S K) \\
& = \text{enactSK} (B K (B S) (B K)) \\
& \rightarrow* \text{enact1} (\text{enactSK} (B K (B S)) (B K)) \\
& \rightarrow* \text{enact1} (B K (B S) (B K)) \\
& \rightarrow* \text{enactSK} (B S) \\
& \rightarrow* B S = 'S
\end{align*}
\]

Note that the evaluation is lazy. To make it eager the special case for \( \text{enactSK} \) must be changed to

\[
x2 x1 \rightarrow \text{enact1} (\text{enactSK} x2 (\text{enactSK} x1))
\]

6.3 An Explicit Self-Enactor

Figure 3 displays a self-enactor for the blocking factorisation calculus that mentions \( E \) explicitly, but will resist typing later on. The case for \( Y \) uses the reduction approach, as the meta-circular approach as the term \( Y x1 \) reduces to \( x1 \) \((Y x1) \) instead of the intended \( x1 \) \((B Y x1) \). The operators \( K \) and \( S \) are handled using the meta-circular approach, as before. The rules for \( F \) and \( E \) are also meta-circular, which proves to be more concise than writing out all of the alternative elaborations of the rules. The role of \( \text{eval1op} \) can
Similar remarks apply to the interpretation of a "dummy" case for \( t \) aren't allowed to use one case per construct in the language, but at the same time we aren't allowed to use \( E \) in a pattern. We overcome this difficulty by applying the dictum of Sherlock Holmes:

"Eliminate all other factors, and the one which remains must be the truth." Sherlock Holmes [11].

That is, by first giving cases for all the other five operators (including a "dummy" case for \( B \)) we can infer the presence of \( E \) without naming it explicitly in a pattern. The resulting self-enactor in which \( E \) is handled implicitly is in Figure 4.

### 6.4 An Implicit Self-Enactor

The type machinery developed in Section 7 is not able to type patterns that contain \( E \). Even though this has only arisen once, in the self-enactor in Figure 3, it creates a major technical challenge: we want the self-enactor to use a pattern-matching function with one case per construct in the language, but at the same time we aren't allowed to use \( E \) in a pattern! We overcome this difficulty by applying the dictum of Sherlock Holmes:

"Eliminate all other factors, and the one which remains must be the truth." Sherlock Holmes [11].

That is, by first giving cases for all the other five operators (including a "dummy" case for \( B \)) we can infer the presence of \( E \) without naming it explicitly in a pattern. The resulting self-enactor in which \( E \) is handled implicitly is in Figure 4.

### 6.5 Correctness

**Lemma 6.2.** If \( v \) is a factorable form then \( \text{enact}('v') \rightarrow* 'v.**

**Proof.** The proof is by a straightforward case analysis on the nature of factorable forms, since none of the special cases of \( \text{enact} \) apply.

**Lemma 6.3.** If \( t_1 \rightarrow t_2 \) then \( \text{enact}(t_1) \) and \( \text{enact}(t_2) \) have a common reduct.

**Proof.** The proof is by induction on the length of the reduction. If \( t_1 \rightarrow t_2 \) then routine case analysis shows that there is a reduction \( \text{enact}(t_1) \rightarrow* \text{enact}(t_2) \). For example, if \( t_1 \) is \( E O O s r \) and \( t_2 \) is \( s \) then \( t_1 \) is the term \( B E (B O) (B O) s 'r \) and

\[
\text{evalop}(B O) \rightarrow* \text{unblock}(\text{enact}(B O)) \rightarrow* \text{unblock}(B O) \quad (\text{by Lemma 6.2}) \rightarrow O
\]

and so \( \text{enact}(t_1) \rightarrow* \text{enact}(E O O s 'r) \rightarrow* \text{enact}('s.**

**Figure 3.** A Self-Enactor that Handles \( E \) Explicitly

```plaintext
let rec enactexp =
  let unblock = B x -> x | x -> x in
  let evalop = x -> unblock (enactexp x) in
  let enact1 =
    B Y x1 -> enactexp (x1 (B Y x1))
  | B K x2 x1 -> enactexp (K x2 x1)
  | B S x3 x2 x1 -> enactexp (S x3 x2 x1)
  | B F x3 x2 x1 -> enactexp (F (evalop x3) x2 x1)
  | B E x4 x3 x2 x1 -> enactexp (E (evalop x4) (evalop x3) x2 x1)
  | x1 -> x1
  in
  x2 x1 -> enact1 (enactexp x2 x1)
  | x1 -> x1
```

**Figure 4.** A Self-Enactor that Handles \( E \) Implicitly

Again, if \( t_1 \) is \( E O u s r \) where \( u \) is a factorable form other than \( O \) and \( t_2 \) is \( r \) then \( t_2 \) is the term \( B E (B O)'u s 'r \) if \( u \) is an operator \( O_1 \) (other than \( O \)) then \( \text{evalop}(B O) \rightarrow* O_1 \) as before, and so \( B E (B O)'u s 'r \rightarrow* E O O \text{enact}(t_2) \) as required. Similarly, if \( u \) is some compound then \( \text{evalop}\ 'u \) reduces to a compound \( q \) and so \( B E (B O)'u s 'r \rightarrow* E O O \text{enact}(t_2) \) as required.

If \( t_1 \) is an application \( r_1 u_1 \) and \( r_2 \) then, by induction, \( \text{enact}(r_1) \) and \( \text{enact}(r_2) \) have a common reduct \( r_3 \). Hence \( \text{enact}(t_1) \) reduces to the term \( \text{enact}(\text{enact}(r_1) u_1) \) which has a common reduct with \( \text{enact}(\text{enact}(r_2) u_2) \) which is a reduct of \( \text{enact}(r_3) \). A similar argument applies if \( u_1 \rightarrow u_2 \).

If \( t_1 \) is some \( F p s r \) and \( p_1 \rightarrow p_2 \) then, by induction, \( \text{enact}(p_1) \) and \( \text{enact}(p_2) \) have a common reduct \( p_3 \). Thus \( \text{enact}(t_1) \) and \( \text{enact}(t_2) \) both reduce to

\[ \text{enact}(F (\text{unblock} p_3) s 'r) . \]

Similar arguments apply if \( t_1 \) is of the form \( E O p s r \).

**Lemma 6.4.** Let \( t \) be a term. If \( \text{enact}'t \rightarrow* \text{enact}'v \) then \( \text{enact}'v \rightarrow* \text{enact}'w \).

**Proof.** The proof is by induction upon the length of the reduction to \( v \).

If \( t \) is an operator \( O \) then the only factorable form \( \text{enact}'t \) can reduce to is \( O \) as required.

If \( t \) is an application \( r u \) then any reduction of \( \text{enact}'t \) produces the term \( \text{enact}(\text{enact}'u) \). Now if \( \text{enact}'t \) is to produce a factorable form then \( \text{enact}'t \) must reduce to a factorable form which, by induction, is some \( t_1 \) where \( v_1 \) is factorable. Now consider the cases of \( \text{enact} \) in turn. If \( v_1 \) is \( Y \) then the whole reduces to \( \text{enact}'(u (B Y u)) \) which is \( \text{enact}'t_1 \) where \( t_1 \rightarrow u (Y u) \) arises from the reduction of \( t \). Now apply induction to \( t_1 \).

Similar arguments apply if \( v_1 \) is of the form \( K x_2 \) or \( S x_3 x_2 \).

Suppose that \( v_1 \) is of the form \( F x_3 x_2 \). If the whole is to produce a factorable form then \( \text{enact}'t_3 \) must produce a factorable form which, by induction, must be a quotation \( p_1 \) where \( x_3 \rightarrow* p_1 \). If \( p_1 \) is an operator \( O \) then \( \text{evalop}(p_1) \) reduces to \( O \) and the whole reduces to \( \text{enact}'x_2 \) so induction applies as \( F x_3 x_2 x_1 \rightarrow x_2 \). Alternatively, if \( p_1 \) is a compound \( p_2 p_3 \) then the whole reduces to \( \text{enact}(u p_2 p_3) \) to which induction can be applied.

Suppose that \( v_1 \) is of the form \( B x_4 x_2 \). Then the whole produces \( (t_1 u) \) which is a quotation of a reduct of \( t \).

Suppose that \( v_1 \) is of the form \( x_5 x_4 x_3 x_2 \). Then it must be that \( x_5 \) is \( E \). Now proceed as before.
Otherwise, \( v_1 \, u \) is a factorable form and the whole reduces to \( \langle v_1 \, u \rangle \) as required.

**Theorem 6.5.** If \( t \) is a term such that \( \text{enact} \, \langle t \rangle \) reduces to some factorable form \( n \) then \( t \) reduces to some factorable form \( v \). Conversely, if \( t \) reduces to some factorable form \( v \) then \( \text{enact} \, \langle t \rangle \) reduces to \( \langle v \rangle \). Hence \( \text{enact} \) is a self-enactor for the blocking factorisation calculus.

**Proof.** If \( \text{enact} \, \langle t \rangle \) has a factorable form, then apply Lemma 6.4. Conversely, suppose that \( t \rightarrow^* \langle v \rangle \) where \( v \) is factorable. By Lemma 6.3, \( \text{enact} \, \langle t \rangle \) and \( \text{enact} \, \langle v \rangle \) have a common reduct and Lemma 6.2 implies that the latter term evaluates to \( \langle v \rangle \) which is normal, as required.

7. Static Type System

7.1 Overview

We approach typing from a Curry-style perspective, in that the terms are fixed in advance, with types merely used to describe terms. This has several consequences, which will be noted when appropriate. Our type system uses the types of System F [13], given by

\[
T ::= X \mid T \to T \mid \forall X.T
\]

where \( X, Y, Z, \ldots \) are meta-variables for type variables, and \( U \) and \( T \) are meta-variables for types. These are much simpler than those of \( \mathbb{F}^n \). The key enabler is the ability to factorise functions in situ, without rising a level in the type hierarchy.

Each operator \( O \) other than \( E \) has a principal type \( \text{Ty}(O) \) of the form

\[
\begin{align*}
Y &: (X \to X) \to X \\
K &: X \to Y \to X \\
S &: (X \to Y \to Z) \to (X \to Y) \to X \to Z \\
F &: X \to Y \to (\forall Z.(Z \to X) \to Z \to Y) \to Y \\
B &: X \to X.
\end{align*}
\]

For later convenience, these types are not quantified, but the type variables are typically assumed fresh.

The types for \( Y, S \) and \( K \) are all standard. Indeed, there is an embedding of Curry-style System F [3] into the blocking factorisation calculus. Hence, the undecidability of type inference for System F [39] carries over to here. However, we have designed and implemented a partial type inference algorithm that can type check our self-interpreters and also catch some mistakes in self-interpreters.

The operator \( B \) has type \( X \to X \). A consequence of the type of \( B \) is that our notion of quotation is type-preserving: if a program has type \( T \), then its representation has type \( T \), too. This shows that different quotations may have different types. An alternative would be to follow Rendel, Ostermann, and Hofer and introduce a new type \( \text{Expr} \) of program representations so that terms cannot be confused with their representations; we leave this for future work.

Following Jay and Given-Wilson [17], the type for \( F \) contains a quantified argument type

\[
\forall Z.(Z \to X) \to Z \to Y.
\]

The variable \( Z \) is used to represent the unknown type of the second component of a compound. This is unnecessary when every pattern is headed by a constructor that determines the types, but knowing that the pattern \( x \, z \) has type \( X \) conveys no information about the type \( Z \) of \( z \).

It is easy to specify a type scheme for \( E \), namely

\[
X \to X \to Y \to Y \to Y
\]

but this is not sufficiently general to type the pattern-matching functions of interest, as each case may have a type that specialises the default type with respect to its pattern. For example, consider an extension of the form

\[
O \to s \mid r
\]

where \( r : U \to T \) and \( s : S \). Its de-sugared form is

\[
\lambda^* x. E \pi x \, s \, (r \, x) = S(S(E \pi O)(K \, s)) r .
\]

Now this should have type \( U \to T \) so take \( x : U \). Then \( E \) must have type

\[
\text{Ty}(O) \to U \to S \to T \to T .
\]

Of course, this is type-safe if \( S = T \) but, following the approach developed in pattern calculus [16], it is enough that any solution of \( \text{Ty}(O) \equiv U \) also solves \( S = T \).

Define \( \{ T_1 = T_2 \} \) to be the most general unifier of \( T_1 \) and \( T_2 \). This is computed in the obvious manner, using \( \alpha \)-conversion to align quantified type variables.

Returning to our example, \( S = \{ \text{Ty}(O) = U \} T \) and so \( E \) must have the type

\[
E : \text{Ty}(O) \to U \to \{ \text{Ty}(O) = U \} T \to T \to T
\]

for any operator \( O \) other than \( E \). It is clear from this that \( E \) cannot have a principal type, and so is excluded from this analysis.

This is good enough for the Curry-style, but from the perspective of the Church-style, or types-as-propositions, this is all very ad hoc, and yet it is not clear how the situation might be better managed. The typing suggests that \( E \) be replaced by a family of operators \( E_O \) for each operator \( O \). Yet each of these would in turn require an equality operator, even though they would not have principal types. We leave such considerations to future work.

There remains the challenge of typing patterns involving \( E \) itself. To date, we have not found a technique that works, and so will confine attention to the self-enactor in Figure 4 in which \( E \) does not appear explicitly.

A final issue concerns the typing of lambda-abstractions. When type-level operations are explicit in System F then the standard approach to instantiating a quantified type asserts that if \( t : \forall X.T \) then \( t : \{ U / X \} T \) for any type \( U \). However, given \( f : S \to \forall X.T \) then the instantiation of \( X \) is achieved by first applying \( f \) to some fresh variable \( x : S \) then instantiating at \( U \) and finally abstracting with respect to \( x \) to get \( \lambda x.x \, f \). This is \( \lambda x.x \, f : S \to \{ U / X \} T \). In the Curry-style, this becomes \( \lambda x.f \, x \, s : S \to \{ U / X \} T \). Mitchell [24], and later Remy [30], considered the consequences of adding \( \eta \)-contraction to System F. For us, the situation is not quite the same, as \( \lambda x.f \, x \) is defined to be \( f \). So we require a type-derivation rule of the form

\[
t : S \to \forall X.T
\]

\[
t : S \to \{ U / X \} T .
\]

More generally, we need a subsumption rule with respect to a type instantiation relation \( \prec \) which generalises the usual type manipulations.

7.2 Typing Rules

A context is given by a sequence \( \Delta \) of type variables, so that the judgments take the form \( \Delta \vdash T \prec T \) which asserts that \( T_2 \) is an instance of \( T_1 \) in context \( \Delta \). For example, \( \Delta \vdash T \prec \forall X.T \) whenever \( X \) is not free in \( \Delta \).

The rules for the type instantiation order \( \prec \) are given in Figure 5, where \( \text{FV}(S) \) is the free variables of \( S \).

A type context \( \Gamma \) is a sequence of distinct, typed term variables \( x_1 : T_1, \ldots, x_n : T_n \) as usual. The free type variables \( \text{FV}(\Gamma) \) of \( \Gamma \) is the union of the free type variables of each type \( T_i \) appearing within it. The type derivation rules are given in Figure 6.
\[
\begin{align*}
\Delta \vdash \forall X.T < [U/X]T & \quad \Delta, \text{FV}(S) \vdash T_1 \prec T_2 \\
\Delta \vdash S \rightarrow T_3 \prec S \rightarrow T_2 \\
\Delta \vdash S_1 \rightarrow T \prec S_2 \rightarrow T \\
x \vdash T \in \Gamma & \\
\Gamma \vdash x : T \\
\end{align*}
\]

**Figure 5. Type Instantiation**

\[FV(\text{Ty}[O]) \cap FV(U \rightarrow T) = \{\} \]

\[\Gamma \vdash E : \text{Ty}[O] \rightarrow U \rightarrow \{\text{Ty}[O] = U\}T \rightarrow T \rightarrow T \]

\[\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U \]

\[\Gamma \vdash t : T_1 \quad \text{FV}(\Gamma) \vdash T_1 \prec T_2 \]

\[\Gamma \vdash T_2 \]

**Figure 6. Type Rules for Terms**

or, equivalently

\[\Gamma, y : Z \vdash U \vdash p_1 \rightarrow (p_2 \rightarrow s \mid r' y) \mid r' : (Z \rightarrow U) \rightarrow Z \rightarrow T.\]

Now, the typing of the pattern \(p_1 \rightarrow p_2\) is of the form

\[\Delta_1 ; B_1 \vdash p_1 : P \quad \Delta_2 ; B_2 \vdash p_2 : P \]

\[\Delta_1 \Delta_2 ; \upsilon(B_1, B_2) \vdash p_1 \rightarrow p_2 : \upsilon(x) \quad \upsilon = \{p_1 = P_2 \rightarrow X\}.\]

Hence, it is enough to prove that

\[\upsilon_1(\Gamma, y : Z \rightarrow U, B_1) \vdash p_2 \rightarrow s \mid r' y : \upsilon_1(Z \rightarrow T)\]

where \(\upsilon_1 = \{p_1 = Z \rightarrow U\}\). Since \(r' y\) has the desired type, this holds if

\[\upsilon_2(\upsilon_1(\Gamma, B_1, B_2)) \vdash s : \upsilon_2(\upsilon_1 T)\]

where \(\upsilon_2 = \{p_2 = v_1 Z\}\). Further, we have the premise

\[\{U = v X\} \mid \upsilon_1(B_1, B_2) \vdash s : \{U = v X\} (\upsilon(T)).\]

Hence, it is enough to show that the restrictions of the compositions

\[\upsilon_2 \circ \upsilon_1 \text{ and } \{U = v X\} \circ v \text{ to } \Gamma, B_1, B_2 \text{ and } T \text{ are the same.}\]

Now, the former is the most general solution of \(P_1 = Z \rightarrow U\) and \(Z = P_2\) or, equivalently, of \(p_1 = P_2 \rightarrow U\) and \(Z = P_2\). Similarly, the latter is the most general solution of \(p_1 = P_2 \rightarrow X\) and \(X = U\) or, equivalently, of \(p_1 = P_2 \rightarrow U\) and \(X = U\). As neither restriction involves \(X\) or \(Z\) it follows that both are simply \(\{P_1 = P_2 \rightarrow U\}\).

**Corollary 7.4. The following rule can be derived for extensions**

\[\Gamma \vdash r : \forall X.X \rightarrow X \quad \Delta, B \vdash p : X \quad \Gamma, B \vdash s : X \quad FV(\Gamma) \cap \Delta = \{\} \]

\[\Gamma \vdash p \rightarrow s \mid r : \forall X.X \rightarrow X\]

**Proof.** Instantiate the type of \(r\) to be \(Y \rightarrow Y\) for some fresh variable \(Y\) and apply the theorem with \(\{Y = Y\}\) mapping \(Y\) to \(T\).

Note that although the corollary above will be sufficient to type our self-interpreters, it is not clear how to prove the corollary without first proving the more general theorem, since the typing of an extension with a compound pattern requires unification to handle the quantified type of the third argument of \(F\).

**7.4 Type Checking the Self-Interpreters**

**Theorem 7.5.** We have \(\Gamma \vdash t : T\) if and only if \(\Gamma \vdash t : T\).

**Proof.** Each direction is straightforward by induction on \(t\).

**Theorem 7.6.** The function \(\text{equal}\) defined in Section 5 has typing

\[0 \vdash \text{equal} : \forall X.\forall Y.X \rightarrow Y \rightarrow \text{Bool} \]
The key thing to note is that the type \(\forall Z.Z \rightarrow Z\), which, when de-sugared, reduces to
\[
\left(\lambda x.\left(p \rightarrow \lambda x.s \mid (S(K r))\right)\right).
\]
Otherwise, when desugaring \(p q \rightarrow s \mid r\) the default term \(r\) appears three times. When this is inefficient, the extension \(p \rightarrow s \mid r\) will be interpreted by its \(\beta\)-expansion
\[
(\lambda x.\left(p \rightarrow s \mid x\right))\ r
\]
to avoid the copying.

After de-sugaring, unquote is:
\[
\begin{array}{l}
\text{Y}(\text{S}(\text{S}(\text{S}(\text{K} \text{S})(\text{S} \text{F})(\text{K} \text{S})))\text{S}(\text{S}(\text{S}(\text{S} \text{E})(\text{K} \text{S} \text{K}))))
\end{array}
\]
which is built from 50 operators. The combinator for \(\text{enact}\) is shown in Figure 8; it uses 1185 operators. Both work fine in all our experiments, which have tested all of the cases in the extensions used in defining \(\text{enact}\).

9. Related Work on Typed Self-Interpretation

A major source of difficulties for static type checking is that programs must be of function type, while their quotations must be data structures, amenable to analysis. The issues are well illustrated by Naylor’s [28] self-interpreter for Haskell that has type:
\[
[\text{FunId}, \text{Exp}] \rightarrow \text{Exp}.
\]
The input is a list of function definitions that each pairs a function identifier with an expression, and the output is also an expression. The key thing to note is that the type \(\text{Exp}\) is a tagged union of integers, variables, abstractions, applications, etc:

\[
\text{data Exp} = \text{App Exp Exp} \mid \text{Lambda VarId Exp} \mid \text{Fun FunId} \mid \text{Var VarId} \mid \text{Int Int} \mid \text{Lam (Exp -> Exp)}
\]

This type supports pattern-matching of the traditional kind (driven by the structure of an algebraic data type) but also brings some disadvantages too. Note that, although it is straightforward to decide equality of the five forms of expression that are used to represent input programs, the presence of arbitrary Haskell functions (tagged by the constructor \(\text{Lam}\)) within expressions will complicate any analysis of interpretations. More significant for the typing is that the resulting quotation process gives all program representations the same type \(\text{Exp}\), which severely limits the usefulness of static type checking. The self-interpreter uses tagging and untagging operations at every step of computation, which amounts to little more than dynamic type checking.

Others have used tags in a similar manner, including Rossberg in his self-interpreter [34] for Standard ML, and Läufer and Odersky [22] in their self-interpreter for a typed version of the SK combinator calculus. Taha, Makholm, and Hughes [36], and also Danvy and López [10], showed how to eliminate superfluous tags.
In the examples above, tags arise as constructors of an algebraic data type of expressions. Certainly, our approach is not tagged in this sense. Rather, intensionality is built into the calculus in a fundamental way. Since equality is decidable for normal forms, including operators, there is no need for any additional tagging.

Before these efforts, Hagiya [14] presented a self-interpreter for a λ-calculus which implicitly defines a type system and does dynamic type checking.

While typing is unnecessary for self-interpretation in general, we are inspired by typability-preserving compilers [27] that compile a typed source program to a typed target program and enable interpreters to type check the target program.

An entirely different approach to typed interpretation is to use polymorphic types instead of a single universal type for all program representations. Significant progress in this direction was made by Pfenning and Lee [29] who studied the polymorphic λ-calculus.

While typing is unnecessary for self-interpretation in general, we are inspired by typability-preserving compilers [27] that compile a typed source program to a typed target program and enable interpreters to type check the target program.

An entirely different approach to typed interpretation is to use polymorphic types instead of a single universal type for all program representations. Significant progress in this direction was made by Pfenning and Lee [29] who studied the polymorphic λ-calculus.

While typing is unnecessary for self-interpretation in general, we are inspired by typability-preserving compilers [27] that compile a typed source program to a typed target program and enable interpreters to type check the target program.

An entirely different approach to typed interpretation is to use polymorphic types instead of a single universal type for all program representations. Significant progress in this direction was made by Pfenning and Lee [29] who studied the polymorphic λ-calculus.

While typing is unnecessary for self-interpretation in general, we are inspired by typability-preserving compilers [27] that compile a typed source program to a typed target program and enable interpreters to type check the target program.

An entirely different approach to typed interpretation is to use polymorphic types instead of a single universal type for all program representations. Significant progress in this direction was made by Pfenning and Lee [29] who studied the polymorphic λ-calculus.

While typing is unnecessary for self-interpretation in general, we are inspired by typability-preserving compilers [27] that compile a typed source program to a typed target program and enable interpreters to type check the target program.

In summary, our language, type system, and self-interpreters have three of those properties (1,4,5), while our language intentionally does not have this property; rather, we use pattern matching pervasively.

4. **Self interpretation.** There is a family of contexts eval_{t} such that t has type (Expr T) if and only if t has type T. We use \( (Expr T) = T \) and the stated equivalence is our Theorem 7.5.

5. **Reflection.** There is a family of contexts eval_{t} such that t has type (Expr T) if and only if t has type T. We use \( (Expr T) = T \) and the stated equivalence is our Theorem 7.5.

In summary, our language, type system, and self-interpreters have three of those properties (1,4,5), while our language intentionally doesn’t have property 3, and we leave property 2 for future work.

The approach of Rendel, Ostermann, and Hofer can be characterised as follows. A function at one level of the type hierarchy can be tagged to become a data structure at the next level. This requires a countable sequence of levels. By contrast, the ability to factorise
means that functions (in normal form) are already data structures without any need to tag them or shift levels. Hence, the types of System F are good enough.

The other notable difference is that both applications of terms to types, and type abstractions, are explicit in their work but implicit here. This saves us from having to factorise type applications, but at the cost of type inference being undecidable.

10. Future Work

The use of factorisation to support self-interpreters raises many interesting questions about foundations and self-interpretation, as well as several practical questions.

When pattern-matching is driven by the definition of an algebraic data type then it is easy to decide whether a pattern-matching function covers all cases, but here “all cases” must include every factorable form of the calculus. Such analyses of coverage await development. In practice, such open-ended, or extensible functions prove quite useful. For example, a pretty-printer of type $\forall X. X \to \text{String}$ may have default behaviour that produces an exception, but as new types are declared, new cases are added for the new term forms.

It seems likely that the calculus without the fixpoint operator $Y$ is strongly normalising, for reasons similar to those for query calculus [16] which extends System F with generic queries for searching and updating. A more interesting challenge is whether the results in this paper can be applied to a strongly normalising calculus. After all, if the basic calculus is strongly normalising, why shouldn’t the interpretations be so too? To put it another way, can the $Y$ operator be replaced by something that is strongly normalising?

By contrast, the denotational semantics of factorisation is quite undeveloped. For example, there is not yet an account of compounds and atoms in category theory.

Open questions within self-interpretation include the following. Is there a self-interpreter that is adequate, in the sense of Rendel, Ostermann, and Hofer? This seems plausible, at the price of making the status of self-interpreters for typed calculi close to the standard ground-breaking work of Rendel, Ostermann, and Hofer, it brings practical questions include the following. Does the calculus admit an efficient implementation? This seems plausible, since factorisation is a formalisation of the care and cadr of Lisp. Can these techniques be applied to $\lambda$-abstractions without first converting to combinators? How easy is it to adapt the given self-interpreters to explore alternative interpretations, e.g. to handle closures?

11. Conclusion

The blocking factorisation calculus is statically typed and supports a quotation mechanism that preserves types and supports both a typed self-recogniser and a typed self-enactor. Building on the ground-breaking work of Rendel, Ostermann, and Hofer, it brings the status of self-interpreters for typed calculi close to the standard set for pure $\lambda$-calculus by Mogensen and then Berarducci and Böhm. Future work may develop strong self-recognisers and self-enactors, and support types of the form Exp $T$ for representations that are distinct from the type $T$ of source programs.

The self-recogniser and self-enactors developed for the blocking factorisation calculus have a very natural development as pattern-matching functions. Each evaluation rule becomes a case of the one-step reducer enact1 with the evaluation strategy captured by the nature of the recursion within which this is embedded. It will be easy enough to modify the strategy, or the reduction rules to suit evolving tastes. In a sense, all self-interpreters can be seen as encodings of such pattern-matching functions.

Further, we anticipate using this approach to model various program transformations, e.g. to produce code in continuation-passing style, and also evaluation strategies involving, say, closures. In general, this work opens up new possibilities for the interpretation of typed programming languages during compiler construction.

More generally, this work illustrates some of the expressive power that the pattern-matching approach brings to bear when one is able to analyse internal structure with the same facility used to apply functions.

Acknowledgments. The second author thanks Oleg Kiselyov, Torben Mogensen, Klaus Ostermann, Frank Pfenning, Andreas Rossberg, Jeff Siskind, Aaron Stump, Wadef Taha, and Mitch Wand for discussions about the state of the art of self-interpretation. We thank Thomas Given-Wilson, Shu-yu Guo, Mohsen Lesani, Todd Millstein and Jose Vergara for helpful comments on a draft of the paper.

References