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 written 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-recognizer 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 summarize next.

The representation of a term t is a data structure $'t$ so it is natural to consider an interpreter as a pattern-matching function in which each evaluation rule `left` $\rightarrow$ `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 analysed by functions at the next higher level, as in the higher-order polymorphic λ-calculus F_ω and even F_ω^* 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 B to block evaluation,

ICFP'11, September 19–21, 2011, Tokyo, Japan.
Copyright © 2011 ACM 978-1-4503-0865-6/11/09...$10.00

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 λ-calculus when the meta-language is the untyped λ-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 λ-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 $'t$ (pronounced "quote t") ready for interpretation. The second is the application of a self-interpreter to $'t$ 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:

$$\textbf{Self-Recogniser:} \quad \texttt{unquote}('t) \approx t \qquad (1)$$

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 λ-calculus, and established (1) as a consequence of β-equality. Barendregt [2] cited Kleene's paper and used the term *self-interpreter* for any λ-term unquote that satisfies $\texttt{unquote}('t) =_\beta t$. Barendregt [2], Mogensen [25, 26], Berarducci and Böhm [5], and Bel [4] all presented λ-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 $\texttt{unquote}('t)$ being a much larger program than t. This problem is mitigated if the self-recogniser is *strong* in the sense that:

$$\textbf{Strong Self-Recogniser:} \quad \texttt{unquote}('t) \longrightarrow^* t$$

where $\longrightarrow^*$ denotes reduction in a calculus, or a small-step operational semantics of a programing 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:

$$\textbf{Self-Enactor:} \quad t \Rightarrow^* v \quad \text{implies} \quad \texttt{enact}('t) \approx {}'v\ . \qquad (2)$$

That is, if t evaluates to a *value* v then $\texttt{enact}('t)$ 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 $\texttt{unquote}(\texttt{enact}('t))$.

The first use of self-enactors is due to Mogensen [25] who proved the existence of such a term in pure λ-calculus. He used the term *self-reducer* for any λ-term enact that satisfies Equation (2). Mogensen [25], Berarducci and Böhm [5] (who preferred the term *reductor*), and Song, Xu, and Qian [35] presented λ-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:

$$\textbf{Strong Self-Enactor:} \quad t \Rightarrow^* v \quad \text{implies} \quad \texttt{enact}('t) \Rightarrow^* {}'v\ .$$

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:

$$\texttt{equal}('s,'t) \Rightarrow \begin{cases} '\texttt{true} & \text{if } 's = {}'t \\ '\texttt{false} & \text{otherwise.} \end{cases}$$

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 λ-terms by higher-order abstract syntax with a meta-language that itself is the untyped λ-calculus. For such representations, no term equal is known to exist. Binding operations complicate the issues because in λ-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 λ-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 equality of quotations, a self-recogniser and a self-enactor. It is a *factorisation calculus* in the sense of Jay and Given-Wilson [17]. Factorisation calculi are more expressive than traditional SK-combinators [17], which may seem surprising since SK-calculus is *combinatorially 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 alone. 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:

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

Each operator has an *arity*, given by $1, 2, 3, 3, 4$ and ∞, respectively, 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* operator 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 arguments are the same operator or not. B is used to block evaluation.

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

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

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* σ are defined and applied in the usual manner. The substitution of the term u_i for the variable x_i for $1 \le i \le n$ 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 collection of which forms the corresponding *combinatory calculus* [15]. Conceptually, the combinatory 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 application 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.

$$\frac{t \rightharpoonup t'}{t \longrightarrow t'} \qquad \frac{r \longrightarrow r'}{r\ u \longrightarrow r'\ u} \qquad \frac{u \longrightarrow u'}{r\ u \longrightarrow r\ u'}$$

Figure 1. The Reduction Relation

3.3 Reduction Semantics

The *reduction rules* are:

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

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* $\longrightarrow$ is obtained by applying the reduction 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 $\longrightarrow^+$ and the reflexive, transitive closure by $(-)^*$ as in $\longrightarrow^*$. If $t \longrightarrow^* t'$ 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. □

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. □

This theorem provides a form of progress property, in that evaluation of the operators, especially F and E, cannot get blocked. Note, however, that if Ω does *not* have a normal form then $F\ \Omega\ K\ K$ 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 `I`,
- λ-abstraction, written $\lambda^* x.s$ and, in typewriter font, `x -> s`,
- let binding, written `let x = s in t`,
- let rec binding, written `let rec x = t` and
- extensions, written `p -> s | t`.

We de-sugar terms with such constructs before executing them. De-sugaring maps closed terms to closed terms. The type-writer font is used when the emphasis is on programming, rather than the calculus.

4.1 Identity, λ-abstraction, let, and let rec

We de-sugar I to the combinator SKK.

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

$$\begin{array}{rcll} \lambda^* x.x &=& I & \\ \lambda^* x.t &=& K\,t & \text{if } t \text{ avoids } x \\ \lambda^* x.t\,x &=& t & \text{if } t \text{ avoids } x \\ \lambda^* x.(r\,u) &=& S(\lambda^* x.r)\,(\lambda^* x.u) & \text{otherwise .} \end{array}$$

The use of η-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*

$$(\lambda^* x.s)\,u \longrightarrow^* [u/x]s\ .$$

Proof. The proof is by induction on the structure of the term s. If s is x then $(\lambda^* x.s)u = I\ u \longrightarrow^* u = [u/x]s$. If s avoids x then $(\lambda^* x.s)u = K\ s\ u \longrightarrow s = [u/x]s$. 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 $s_1 s_2$ then

$$\begin{array}{rcl} (\lambda^* x.s)u &=& S(\lambda^* x.s_1)(\lambda^* x.s_2)u \\ &\longrightarrow& (\lambda^* x.s_1)u((\lambda^* x.s_2)u) \\ &\longrightarrow^*& [u/x]s_1([u/x]s_2) \\ &=& [u/x]s \end{array}$$

by two applications of induction. □

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

4.2 Extensions

A *pattern-matching function* of the form

$$\begin{array}{l} \ \ p_1 \to s_1 \\ \mid p_2 \to s_2 \\ \ \ \dots \\ \mid p_n \to s_n \\ \mid x \to s \end{array}$$

can be built as a sequence of *extensions* of the form `p -> 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 data type, 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:

$$\begin{array}{rcl} x \to s \mid r &=& \lambda^* x.s \\ O \to s \mid r &=& \lambda^* x.E\ O\ x\ s\ (r\ x) = S(S(E\ O)(K\ s))r \\ p\,q \to s \mid r &=& \lambda^* x.F\ x\ (r\ x) \\ && (\lambda^* y.(p \to (q \to s \mid r'\ y) \mid r')\ y) \\ && \text{(where } r' = \lambda^* y.\lambda^* z.r\ (y\ z) = S(K\ r)\text{)} \end{array}$$

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 `Some` σ where σ is a substitution, or a *match failure* `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 `None`. The *matching* $\{u/p\}$ of a pattern p against a term u is defined by the rules

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

corresponding to those of static pattern calculus [16].

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

$$\begin{array}{lcll} (p \to s \mid r)\ u &\longrightarrow^*& \sigma s & (\textit{if } \{u/p\} = \texttt{Some}\ \sigma) \\ (p \to s \mid r)\ u &\longrightarrow^*& r\ u & (\textit{if } \{u/p\} = \texttt{None}). \end{array}$$

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 λ-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 `unblock` defined by

```
  B x -> x
| x -> x
```

which de-sugars to the combinator

$$\begin{array}{cl} & B\ x \to x \mid x \to x \\ = & \lambda^* x.F\ x\ (I\ x)\ (\lambda^* y.(B \to (x \to x \mid r'\ y) \mid r')\ y) \\ = & \lambda^* x.F\ x\ (I\ x)\ (\lambda^* y.(B \to I \mid r')\ y) \\ = & \lambda^* x.F\ x\ (I\ x)\ (B \to I \mid r') \\ = & \lambda^* x.F\ x\ (I\ x)\ (S(S(E\ B)(K\ I))r') \\ = & S(S\ F\ I)(K(S(S(E\ B)(K\ I))(S(K\ I)))) \end{array}$$

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

5. Self-Recognisers

5.1 Quotation

Quotation for both our self-interpreters is given by

$$\begin{array}{rcl} {}'x &=& x \\ {}'O &=& BO \\ {}'(s\,t) &=& {}'s\,{}'t \end{array}$$

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

$$\frac{t \rightharpoonup t'}{t \Rightarrow t'} \qquad \frac{r \Rightarrow r'}{r\, u \Rightarrow r'\, u} \qquad \frac{p \Rightarrow p'}{F\, p \Rightarrow F\, p'}$$

$$\frac{p \Rightarrow p'}{E\, p \Rightarrow E\, p'} \qquad \frac{q \Rightarrow q'}{E\, p\, q \Rightarrow E\, p\, q'}$$

Figure 2. Call-by-Name Evaluation

```
let rec equal =
  x1 x2 -> (
    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

Define `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* $\longrightarrow^*$.

Proof. The proof is a straightforward induction on the structure of t. If t is a variable or operator then $\texttt{unquote}('t) \longrightarrow^* t$. 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 $\texttt{unquote}('t) = \texttt{unquote}('t_1\ 't_2) \longrightarrow^*\ 't_1\ 't_2$ 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 factorable forms, as required to support its reduction.

To define behavioural equivalence requires a notion of value and of context. Define a *value* 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 $t \Rightarrow^* v$.

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 σ 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* $t_1 \longrightarrow t_2$ *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 `left` $\rightarrow$ `right` as a case, namely the *reduction approach* and the *meta-circular approach* (pace [32]). The reduction approach represents the rule by the case

$$|\,'\texttt{left} \rightarrow \texttt{enact}\,'\texttt{right}$$

where the left- and right-hand sides of the rule have been quoted. For the reduction rule for S, this yields

```
| B S x3 x2 x1 -> enact (x3 x1 (x2 x1))
```

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 -> enact (S x3 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 `enactSK` defined by

```
let rec enactSK =
  let enact1 =
    B K x2 x1 -> enactSK (K x2 x1)
  | B S x3 x2 x1 -> enactSK (S x3 x2 x1)
  | x1 -> x1
  in
    x2 x1 -> enact1 (enactSK x2 x1)
  | x1 -> x1
```

The function `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 `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 `enactSK` itself has two cases: that for a compound enacts the left-hand component and then reduces the whole by `enact1`.

For example,

```
enactSK '(K S K)
    =    enactSK (B K (B S) (B K))
  ⟶*    enact1 (enactSK (B K (B S)) (B K))
  ⟶*    enact1 (B K (B S) (B K))
  ⟶*    enactSK (B S)
  ⟶*    B S = 'S
```

Note that the evaluation is lazy. To make it eager the special case for `enactSK` must be changed to

```
x2 x1 -> enact1 (enactSK x2 (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 `evalop` can

```
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 3. A Self-Enactor that Handles E Explicitly

be illustrated by an example. Consider

$$\begin{array}{ll} & \texttt{enactexp '(F K S K)} \\ = & \texttt{enactexp (B F (B K) (B S) (B K))} \\ \longrightarrow^* & \texttt{enact1 (B F (B K) (B S) (B K))} \\ \longrightarrow^* & \texttt{enactexp (F (evalop (B K)) (B S) (B K)} \end{array}$$

If `evalop (B K)` were replaced by `enact (B K)` then `F` would be applied to `B K` when it should be applied to `K`. So `evalop` is used to “unquote” operators while leaving everything else unchanged. Similar remarks apply to the interpretation of `E`. Note that there is no case for `B` in `enact1` as it has no reduction rules.

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* `enact`$('v) \longrightarrow^* {}'v$.

Proof. The proof is by a straightforward case analysis on the nature of factorable forms, since none of the special cases of `enact1` apply. □

LEMMA 6.3. *If $t_1 \longrightarrow t_2$ then* `enact`$('t_1)$ *and* `enact`$('t_2)$ *have a common reduct.*

Proof. The proof is by induction on the length of the reduction. If $t_1 \rightharpoonup t_2$ then routine case analysis shows that there is a reduction `enact` $'t_1 \longrightarrow^+$ `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

$$\begin{array}{lll} \texttt{evalop}\,(B\ O) & \longrightarrow^* & \texttt{unblock}\,(\texttt{enact}\,(B\ O)) \\ & \longrightarrow^* & \texttt{unblock}\,(B\ O) \qquad \text{(by Lemma 6.2)} \\ & \longrightarrow^+ & O \end{array}$$

and so `enact` $'t_1 \longrightarrow^*$ `enact` $(E\ O\ O\ 's\ 'r) \longrightarrow$ `enact` $'s$.

```
let rec enact =
  let unblock = B x -> x | x -> x in
  let evalop = x -> unblock (enact x) in
  let enact1 =
    B Y x1 -> enact (x1 (B Y x1))
  | B K x2 x1 -> enact (K x2 x1)
  | B S x3 x2 x1 -> enact (S x3 x2 x1)
  | B F x3 x2 x1 -> enact (F (evalop x3) x2 x1)
  | B B x4 x3 x2 x1 -> B B x4 x3 x2 x1
  | B x5 x4 x3 x2 x1 ->
      enact (x5 (evalop x4) (evalop x3) x2 x1)
  | x1 -> x1
  in
    x2 x1 -> enact1 (enact 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 $'t_1$ is the term $B\ E\ (B\ O)\ 'u\ 's\ 'r$. If u is an operator O_1 (other than O) then `evalop` $(B\ O_1) \longrightarrow^* O_1$ as before, and so $B\ E\ (B\ O)\ 'u\ 's\ 'r \longrightarrow^* E\ O\ O_1\ 's\ 'r \longrightarrow\ 'r$ as required. Similarly, if u is some compound then `evalop` $'u$ reduces to a compound q and so $B\ E\ (B\ O)\ 'u\ 's\ 'r \longrightarrow^* E\ O\ q\ 's\ 'r \longrightarrow\ 'r$ as required.

If t_1 is an application $r_1\ u_1$ and $r_1 \longrightarrow r_2$ then, by induction, `enact` $'r_1$ and `enact` $'r_2$ have a common reduct r_3. Hence `enact` $'t_1$ reduces to the term `enact1` (`enact` $'r_1\ 'u_1$) which has a common reduct with `enact1` (`enact` $'r_2\ 'u_1$) which is a reduct of `enact` $'(r_2\ u_1)$. A similar argument applies if $u_1 \longrightarrow u_2$.

If t_1 is some $F\ p\ s\ r$ and $p_1 \longrightarrow p_2$ then, by induction, `enact` $'p_1$ and `enact` $'p_2$ have a common reduct p_3. Thus `enact`$('t_1)$ and `enact`$('t_2)$ both reduce to

$$\texttt{enact}\,(F\,(\texttt{unblock}\,p_3)\ 's\ 'r)\ .$$

Similar arguments apply if t_1 is of the form $E\ p\ s\ r\ q$. □

LEMMA 6.4. *Let t be a term. If* `enact` $'t$ *reduces to a factorable form v then v is a quotation of some t_1 such that $t \longrightarrow^* t_1$.*

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 `enact` $'t$ can reduce to is $'O$ as required.

If t is an application $r\ u$ then any reduction of `enact` $'t$ produces the term `enact1` (`enact` $'r\ 'u$). Now if `enact` $'t$ is to produce a factorable form then `enact` $'r$ must reduce to a factorable form which, by induction, is some $'v_1$ where v_1 is factorable. Now consider the cases of `enact1` in turn. If v_1 is Y then the whole reduces to `enact` $('u\ (B\ Y\ 'u))$ which is `enact` $'t_1$ where $t_1 = u\ (Y\ u)$ 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 `enact` $'x_3$ must produce a factorable form which, by induction, must be a quotation $'p_1$ where $x_3 \longrightarrow^* p_1$. If p_1 is an operator O then `evalop` $'p_1$ reduces to O and the whole reduces to `enact` $'x_2$ so induction applies as $F\ x_3\ x_2\ x_1 \longrightarrow x_2$. Alternatively, if p_1 is a compound $p_2\ p_3$ then the whole reduces to `enact` $('u\ 'p_2\ 'p_3)$ to which induction can be applied.

Suppose that v_1 is of the form $B\ x_4\ x_3\ x_2$. Then the whole produces $'(v_1\ u)$ 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 $'(v_1\ u)$ as required. □

THEOREM 6.5. *If t is a term such that* `enact` *$'t$ reduces to some factorable form n then t reduces to some factorable form v. Conversely, if t reduces to some factorable form v then* `enact` *$'t$ reduces to $'v$. Hence* `enact` *is a self-enactor for the blocking factorisation calculus.*

Proof. If `enact` $'t$ has a factorable form, then apply Lemma 6.4. Conversely, suppose that $t \longrightarrow^* v$ where v is factorable. By Lemma 6.3, `enact` $'t$ and `enact` $'v$ have a common reduct and Lemma 6.2 implies that the latter term evaluates to $'v$ 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 F^*_ω. 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* $\mathsf{Ty}[O]$ of the form

$$\begin{array}{rcl} Y & : & (X \to X) \to X \\ K & : & X \to Y \to X \\ S & : & (X \to Y \to Z) \to (X \to Y) \to X \to Z \\ F & : & X \to Y \to (\forall Z.(Z \to X) \to Z \to Y) \to Y \\ B & : & X \to X\ . \end{array}$$

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 $\mathsf{Expr}\ T$ 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

$$\lambda^* x.E\ O\ x\ s\ (r\ x) = S(S(E\ O)(K\ s))r\ .$$

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

$$\mathsf{Ty}[O] \to U \to S \to T \to T\ .$$

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 $\mathsf{Ty}[O] = U$ 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 α-conversion to align quantified type variables.

Returning to our example, $S = \{\mathsf{Ty}[O] = U\}T$ and so E must have the type

$$E : \mathsf{Ty}[O] \to U \to \{\mathsf{Ty}[O] = U\}T \to T \to T$$

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 $\lambda x.f\ x\ U : S \to \{U/X\}T$. In the Curry-style, this becomes $\lambda x.f\ x : S \to \{U/X\}T$. Mitchell [24], and later Remy [30], considered the consequences of adding η-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

$$\frac{t : S \to \forall X.T}{t : S \to \{U/X\}T}\ .$$

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 Δ of type variables, so that the judgments take the form $\Delta \vdash T_1 \prec T_2$ which asserts that T_2 is an instance of T_1 in context Δ. For example, $\Delta \vdash T \prec \forall X.T$ whenever X is not free in Δ.

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

A *type context* Γ is a sequence of distinct, typed term variables $x_1 : T_1, \ldots, x_n : T_n$ as usual. The *free type variables* $\mathsf{FV}(\Gamma)$ of Γ 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.

$$\frac{}{\Delta \vdash \forall X.T \prec [U/X]T} \qquad \frac{}{\Delta \vdash T \prec \forall X.T}\; X \notin \Delta$$

$$\frac{\Delta, \mathsf{FV}(S) \vdash T_1 \prec T_2}{\Delta \vdash S \to T_1 \prec S \to T_2} \qquad \frac{\Delta \vdash S_2 \prec S_1}{\Delta \vdash S_1 \to T \prec S_2 \to T}$$

Figure 5. Type Instantiation

THEOREM 7.1. *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow u$ *then* $\Gamma \vdash u : T$.

Proof. Consider the reduction $E\ O\ O\ s\ t \longrightarrow s$. Since U is a type for O and $\upsilon = \{U = \mathsf{Ty}[O]\}$ exists it follows that the domain of υ can be limited to the free type variables of $\mathsf{Ty}[O]$ so that $t : \{\mathsf{Ty}[O] = U\}T = T$ as required. That the other reduction rules preserve typing is routine. □

7.3 Derived Typing Rules

LEMMA 7.2. *The following rule can be derived for abstractions*

$$\frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda^* x.t : U \to T}\ .$$

Proof. The proof is by induction on the structure of the type derivation for t. If the last step in the derivation uses a type instantiation $\mathsf{FV}(\Gamma, x : U) \vdash T_1 \prec T$ then it follows that $\mathsf{FV}(\Gamma) \vdash U \to T_1 \prec U \to T$ so induction applies. The remaining possibilities follow the structure of t. If t is x then T is U and so $\lambda^* x.x = I : \forall X.X \to X \prec U \to U = U \to T$ as required. If x is not free in t then $\lambda^* x.t = K\ t$ and $\Gamma \vdash K\ t : U \to T$ as required. If t is of the form $r\ x$ where x is not free in r then $\lambda^* x.r\ x$ is r. Now the type derivation for t ends with some

$$\frac{\Gamma, x : U \vdash r : U_1 \to T \quad \Gamma, x : U \vdash x : U_1}{\Gamma, x : U \vdash r\ x : T}\ .$$

It follows that $\mathsf{FV}(\Gamma, x : U) \vdash U \prec U_1$ and so $\mathsf{FV}(\Gamma, x : U) \vdash U_1 \to T \prec U \to T$ by contravariance of the order with respect to argument types, which yields the desired typing for r. Otherwise, if t is an application $t_1\ t_2$ then there are types T_1 and T_2 such that $\Gamma \vdash t_1 : T_2 \to T$ and $\Gamma \vdash t_2 : T_2$. By two applications of induction, it follows that $\Gamma \vdash \lambda^* x.t_1 : U \to T_2 \to T$ and $\Gamma \vdash \lambda^* x.t_2 : U \to T_2$ whence $\lambda^* x.t = S(\lambda^* x.t_1)(\lambda^* x.t_2)$ has type $U \to T$ as required.

□

The intended typing rules for extensions depend upon patterns taking their most general types, as described by type judgments of the form $\Delta; B \vdash p : P$ where Δ is as before and B (big beta) is a type context in which each term takes a mono-type. The rules are presented in Figure 7. In the last rule it is implicit that the type variables in Δ_1, Δ_2 and X are distinct. Note that E does not have a principal type, and so cannot appear in patterns.

THEOREM 7.3. *The following rule can be derived for extensions*

$$\frac{\Gamma \vdash r : U \to T \quad \Delta; B \vdash p : P \quad \upsilon(\Gamma, B) \vdash s : \upsilon T \qquad \Delta \cap (\mathsf{FV}(\Gamma) \cup \mathsf{FV}(U \to T)) = \{\} \quad \upsilon = \{U = P\}}{\Gamma \vdash p \to s \mid r : U \to T}\ .$$

Proof. The proof is by induction upon the structure of p.

If p is a variable then apply Lemma 7.2.

If p is an operator O other than E then $p \to s \mid r = \lambda^* x.E\ O\ x\ s\ (r\ x) : U \to T$ as required.

If p is an application $p_1 p_2$ then $p \to s \mid r = \lambda^* x.F\ x\ (r\ x)\ (\lambda^* y.(p_1 \to (p_2 \to s \mid r'\ y) \mid r')\ y)$ where $r' = S(K\ r)$. This has type $U \to T$ if there is derivation of

$$\Gamma, y : Z \to U \vdash (p_1 \to (p_2 \to s \mid r'\ y) \mid r')\ y : Z \to T$$

or, equivalently

$$\Gamma, y \colon Z \to U \vdash p_1 \to (p_2 \to s \mid r'\ y) \mid r' : (Z \to U) \to Z \to T\ .$$

Now the typing of the pattern $p_1\ p_2$ is of the form

$$\frac{\Delta_1; B_1 \vdash p_1 : P_1 \quad \Delta_2; B_2 \vdash p_2 : P_2}{\Delta_1, \Delta_2; \upsilon(B_1, B_2) \vdash p_1\ p_2 : \upsilon X}\ \upsilon = \{P_1 = P_2 \to X\}.$$

Hence, it is enough to prove that

$$\upsilon_1(\Gamma, y : Z \to U, B_1) \vdash p_2 \to s \mid r'\ y : \upsilon_1(Z \to T)$$

where $\upsilon_1 = \{P_1 = Z \to U\}$. Since $r'\ y$ has the desired type, this hold s if

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

where $\upsilon_2 = \{P_2 = \upsilon_1 Z\}$. Further, we have the premise

$$\{U = \upsilon X\}(\upsilon(\Gamma, B_1, B_2)) \vdash s : \{U = \upsilon X\}(\upsilon(T))\ .$$

Hence, it is enough to show that the restrictions of the compositions $\upsilon_2 \circ \upsilon_1$ and $\{U = \upsilon X\} \circ \upsilon$ to Γ, B_1, B_2 and T are the same.

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

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

$$\frac{\Gamma \vdash r : \forall X.X \to X \quad \Delta; B \vdash p : X \quad \Gamma, B \vdash s : X \qquad \mathsf{FV}(\Gamma) \cap \Delta = \{\}}{\Gamma \vdash p \to s \mid r : \forall X.X \to X}\ .$$

Proof. Instantiate the type of r to be $Y \to Y$ for some fresh variable Y and apply the theorem with $\{Y = X\}$ mapping Y to X. □

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* `equal` *defined in Section 5 has typing*

$$\emptyset \vdash \texttt{equal} : \forall X.\forall Y.X \to Y \to \texttt{Bool}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\frac{}{\Gamma \vdash O : \mathsf{Ty}[O]}$$

$$\frac{\mathsf{FV}(\mathsf{Ty}[O]) \cap \mathsf{FV}(U \to T) = \{\}}{\Gamma \vdash E : \mathsf{Ty}[O] \to U \to \{\mathsf{Ty}[O] = U\}T \to T \to T}$$

$$\frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T}$$

$$\frac{\Gamma \vdash t : T_1 \quad \mathsf{FV}(\Gamma) \vdash T_1 \prec T_2}{\Gamma \vdash t : T_2}$$

Figure 6. Type Rules for Terms

$$\frac{}{X; x : X \vdash x : X} \qquad \frac{}{\mathsf{FV}(\mathsf{Ty}[O]) \vdash O : \mathsf{Ty}[O]}$$

$$\frac{\Delta_1; B_1 \vdash p_1 : P_1 \quad \Delta_2; B_2 \vdash p_2 : P_2}{\Delta_1, \Delta_2, X; \upsilon(B_1, B_2) \vdash p_1\, p_2 : \upsilon X}\ \upsilon = \{P_1 = P_2 \to X\}$$

Figure 7. Type Rules for Patterns

where `Bool` $= \forall Z.Z \to Z \to Z.$

Proof. This is a consequence of derived type inference rule for extensions. The calculations are relatively straightforward since the body of each case has the same type `Bool`. □

THEOREM 7.7. *The self-recogniser* `unquote` *defined in Section 5 has typing*

$$\emptyset \vdash \texttt{unquote} : \forall X.X \to X\ .$$

Proof. Apply Corollary 7.4. □

THEOREM 7.8. *The self-enactor defined in Figure 4 has type*

$$\emptyset \vdash \texttt{enact} : \forall X.X \to X\ .$$

Proof. The proof is by repeated applications of Corollary 7.4. □

Notice that we can easily type check self-applications of `unquote` and `enact`, such as `enact`(’`enact`).

7.5 Adequacy

A weakness of our approach is that the type system does not distinguish terms from their quotations. A more refined approach associates to each type T a new type form `Expr` T to type its quoted expressions. Then quotation is said to be *adequate* [31] if each closed normal form of type `Expr` T is the quotation of some term.

Future work may well adapt the self-interpreters given here to make them adequate. In the meantime, observe that there is not much scope for confusion, as there is a simple test for being a quotation, given by

```
let rec isquote =
  B (x y) -> false
| B x -> true
| x y -> isquote x && (isquote y)
| x -> false
```

where `&&` is conjunction. Similarly, given an interpretation of strings there is a pretty printer for quotations given by

```
let rec pretty_print  =
  B Y -> "Y"
| B K -> "K"
| B S -> "S"
| B F -> "F"
| B E -> "E"
| B B -> "B"
| x y -> pretty_print x ^ "(" ^ pretty_print y ^ ")"
| x -> "<not a quotation>"
```

8. Experimental Results

We have two implementations of both reduction and desugaring, one in bondi [7] and one in Scheme.

Type inference in the **bondi** interpreter confirms that all examples have the expected types. For example, `enact` in Figure 4 has the same type as the polymorphic identity function. The inference algorithm adapts the standard techniques by adding a rule for typing extensions. This is delicate as it is not obvious how any type substitutions required to infer a type for the body of the extension can be incorporated into the result without forcing the default to take the type of the special case. For our purposes, it is sufficient to merely check the type of the body, without propagating any changes.

In converting the (well-typed) extensions into combinators, it is worth adopting some optimisations when de-sugaring extensions with compound patterns.

A common situation concerns

$$p\,x \to s \mid r$$

which, when de-sugared, reduces to

$$S\ (S\ F\ r)(K(p \to \lambda^* x.s \mid (S(K\ r)))\ .$$

Otherwise, when desugaring $|p\,q \to s \mid r$ the default term r appears three times. When this is inefficient, the extension $p \to s \mid r$ will be interpreted by its β-expansion

$$(\lambda^* x.(p \to s \mid x))\ r$$

to avoid the copying.

After de-sugaring, `unquote` is:

```
Y(S(K(S(S(K S)(S F))(S(K K)(S(K(S(S(E B)(K(S K K)))))
(S(K S) K)))))(S(K(S(S F(S K K))))(S(K K)(S(S(K S)
(S(K(S(K S)))(S(K K))))K))))
```

which is built from 50 operators. The combinator for `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 `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:

$$\texttt{[(FunId, Exp)]} \to \texttt{Exp}\ .$$

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 `Exp` is a tagged union of integers, variables, abstractions, applications, etc:

```
data Exp =  App Exp Exp   |  Lambda VarId Exp
         |  Fun FunId     |  Var VarId
         |  Int Int       |  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 `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 `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.

```
Y(S(S(S(KS)(S(KK)(S(S(KS)(S(KK)(S(K(S(K(S(SF(SKK))))))(S(K(S(KK)))(S(K(S(S(KS)(S(KK)(S(KS)K)))))K)))))(S(S(KS)(S(KK)(S
(K(S(S(KS)(SF))))(S(K(S(KK)))(S(S(KS)(S(KK)(S(K(S(S(KS)(SF))))(S(K(S(KK)))(S(S(KS)(S(K(S(KS)))(S(S(KS)(S(K(S(KS)))(S(K
(S(K(S(KS)))))(S(K(S(K(S(K(S(EB)))))))(S(K(S(K(S(KK)))))(S(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S(K(S(EY)))(S(KK)(S(S(KS)K)(K
(S(SKK)(BY))))))))))(K(S(KS)K))))))))(K(S(KK)(S(KS)K))))))(K(K(SKK))))))))(K(S(KS)K)))))))(S(S(KS)(S(KK)(S(K(S(S(KS)(
SF))))(S(K(S(KK)))(S(K(S(S(KS)(S(K(SF))(S(KS)K)))))(S(K(S(KK)))(S(S(KS)(S(KK)(S(K(S(S(KS)(SF))))(S(K(S(KK)))(S(S(KS)(S
(K(S(KS)))(S(S(KS)(S(K(S(KS)))(S(K(S(K(S(KS)))))(S(K(S(K(S(K(S(EB)))))))(S(K(S(K(S(KK)))))(S(S(KS)(S(KK)(S(KS)(S(KK)(S
(KS)(S(K(S(EK)))(S(KK)(S(S(KS)(S(KK)(S(KS)K)))(KK)))))))))(K(S(KS)K))))))))(K(S(KK)(S(KS)K))))))(K(K(SKK)))))))(K(S(K
S)(S(KK)(S(KS)K))))))))))(S(S(KS)(S(KK)(S(K(S(S(KS)(SF))))(S(K(S(KK)))(S(K(S(S(KS)(S(K(SF))(S(KS)K)))))(S(K(S(KK)))(S
(K(S(S(KS)(S(K(SF))(S(KS)(S(KK)(S(KS)K)))))))(S(K(S(KK)))(S(S(KS)(S(KK)(S(K(S(S(KS)(SF))))(S(K(S(KK)))(S(S(KS)(S(K(S(K
S)))(S(S(KS)(S(K(S(KS)))(S(K(S(K(S(KS)))))(S(K(S(K(S(K(S(EB)))))))(S(K(S(K(S(KK)))))(S(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S
(K(S(ES)))(S(KK)(S(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)K)))))(KS)))))))))(K(S(KS)K))))))))(K(S(KK)(S(KS)K))))))(K(K(SKK)))))
))(K(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)K)))))))))))))(S(S(KS)(S(K(S(K(S(S(KS)(SF))))))(S(K(S(K(S(KK)))))(S(K(S(K(S(S(KS)(
S(K(SF))(S(KS)K)))))))(S(K(S(K(S(KK)))))(S(K(S(K(S(S(KS)(S(K(SF))(S(KS)(S(KK)(S(KS)K)))))))))(S(K(S(K(S(KK)))))(S(S(KS
)(S(K(S(KS)))(S(K(S(KK)))(S(K(S(K(S(S(KS)(SF))))))(S(K(S(K(S(KK)))))(S(S(KS)(S(K(S(KS)))(S(K(S(K(S(KS)))))(S(S(KS)(S(K
(S(KS)))(S(K(S(K(S(KS)))))(S(K(S(K(S(K(S(KS)))))))(S(K(S(K(S(K(S(K(S(EB)))))))))(S(K(S(K(S(K(S(KK)))))))(S(S(KS)(S(K(S
(KS)))(S(K(S(KK)))(S(K(S(KS)))(S(K(S(KK)))(S(K(S(KS)))(S(K(S(K(S(EF)))))(S(K(S(KK)))(S(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S
(KK)(S(KS)K)))))))(K(S(KF))))))))))))(K(K(S(KS)K))))))))))(K(K(S(KK)(S(KS)K))))))))(K(K(K(SKK))))))))))(K(K(S(KS)(S(KK
)(S(KS)(S(KK)(S(KS)K))))))))))))))(S(K(S(K(S(S(KS)(SF))(S(KK)(S(S(KS)(S(K(SF))(S(KS)K)))(S(KK)(S(S(KS)(S(K(SF))(S(KS)
(S(KK)(S(KS)K)))))(S(KK)(S(S(KS)(S(K(SF))(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)K)))))))(S(KK)(S(K(S(S(KS)(SF))(S(KK)(S(S(KS)(S
(S(KS)(S(K(S(KS)))(S(K(S(K(S(EB)))))(S(K(S(KK)))(S(K(S(K(S(S(EB)(K(BB)))))))(S(KS)K))))))(S(KK)(S(KS)K))))(K(SKK)))))
(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)K))))))))))))))))))(S(K(S(K(S(SF(SKK))))))(S(K(S(KK)))(S(K(S(K(S(SF(S(K(SKK
))))))))(S(K(S(KK)))(S(K(S(K(S(SF(S(K(S(K(SKK))))))))))(S(K(S(KK)))(S(K(S(K(S(SF(S(K(S(K(S(K(SKK)))))))))))(S(K(S(KK)
))(S(K(S(K(S(SF(S(K(S(K(S(K(S(K(SKK))))))))))))))(S(K(S(KK)))(S(S(KS)(S(K(S(KS)))(S(K(S(K(S(EB)))))(S(K(S(KK)))(S(S(KS
)(S(KK)(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)(S(KK)(S(KS)K))))))))))(K(S(S(KS)(S(K(S(KS)))(S(K(S(K(S(KS)))))(S(K(
S(K(S(KK)))))(S(K(S(S(KS)K)))K)))))(S(KK)K))))))))(K(K(S(K(S(K(S(K(S(K(S(K(SKK)))))))))))))))))))))))))))))))))(S(K(S(S
(KS)K)))K))(K(S(SF(SKK))(K(S(S(EB)(K(SKK)))(S(K(SKK)))))))
```

Figure 8. `enact` de-sugared

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 implementers to catch compiler errors by type checking 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 and presented an interpreter for F_2 written in F_3 as well as an interpreter for F_ω written in F_ω^+, and by Carette, Kiselyov, and Shan [9] who wrote tagless interpreters in OCaml and Haskell for a simply-typed λ-calculus. They showed the viability of program representations with polymorphic types, without presenting self-interpreters.

The literature contains just one example (as far as we know) of a self-interpreter for a statically-typed language without a universal type. Rendel, Ostermann, and Hofer [31] presented a self-recogniser for F_ω^*, which is an extension of the higher-order polymorphic λ-calculus F_ω that has a type:type rule. The self-recogniser has type:

$$\forall X.(\mathsf{Expr}\ X) \to X$$

Their notion of quotation is *type-monomorphic* in that if $'s$ and $'t$ have the same type, then s and t have the same type too. Equivalently, terms of distinct types yield representations of distinct types. The use of type-monomorphic quotation is a radical departure from the use of a single universal type for all program representations, where all $'s$ and $'t$ have the same type, irrespectively of the types of s and t. They presented a self-recogniser (not strong), and left open the problems of writing an equality checker and a self-enactor. Their results inspired our work.

Rendel, Ostermann, and Hofer discuss the notion of typed self-representation, that is, "representing terms of a programming language in the language itself" [31, Section 2], and list five desirable properties. Let us evaluate how many of those properties our language, type system, and self-interpreters have. We list each property in *this font*, followed by our evaluation.

1. **Representation.** *There is a family of types* $(\mathsf{Expr}\ T)$ *such that* $'t$ *has type* $(\mathsf{Expr}\ T)$ *if and only if* t *has type* T. We use $(\mathsf{Expr}\ T) = T$ and the stated equivalence is our Theorem 7.5.
2. **Adequacy.** *Every term* s *of type* $(\mathsf{Expr}\ T)$ *corresponds to a term* t *of type* T*, which means that for every* s *as above there exists a* t *such that* $s = \ 't$. Our definition of quotation doesn't have this property; the reason is that we use $(\mathsf{Expr}\ T) = T$ so an unquoted term of type T also has type $(\mathsf{Expr}\ T)$.
3. **First class interpretations.** *It is possible to express operations on quoted terms so that they are well-typed for all terms of type* $(\mathsf{Expr}\ T)$*, without the need to refer to any specific such terms.* Our language doesn't have this property; rather, we use pattern matching pervasively.
4. **Self interpretation.** *There is a family of contexts* $\mathsf{eval}_T\langle\rangle$ *such that* $\mathsf{eval}_T\langle 't\rangle$ *is observationally equivalent to* t *if* t *has type* T. This property is essentially Equation (1) which is implied by our Theorem 5.2.
5. **Reflection.** $'t$ *exhibits the intensional structure of* t *in a useful way.* Our self-enactor is a good example of how we can make good use of the intensional structure of a quoted term.

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 $\forall X.X \rightarrow$ `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 everything somewhat more obscure. Is there a self-interpreter for a language with decidable type-checking? This is a harder question, since it is not clear how to factorise the application of a term to a type.

Practical questions include the following. Does the calculus admit an efficient implementation? This seems plausible, since factorisation is a formalisation of the `car` and `cdr` of Lisp. Can these techniques be applied to λ-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 λ-calculus by Mogensen and then Berarducci and Bohm. 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, Walid 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

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Henk Barendregt. Self-interpretations in lambda calculus. *J. Funct. Program*, 1(2):229–233, 1991.

[3] HP Barendregt. *Handbook of Logic in Computer Science (vol. 2): Background: Computational Structures: Abramski,S (ed)*, chapter Lambda Calculi with Types. Oxford University Press, Inc., New York, NY, 1993.

[4] Michel Bel. A recursion theoretic self interpreter for the lambda-calculus. http://www.belxs.com/michel/#selfint.

[5] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In *CSL*, pages 85–99, 1992.

[6] Mathieu Boespflug. From self-interpreters to normalization by evaluation. In Olivier Danvy, editor, *Proceedings of Workshop on Normalization by Evaluation*, 2009.

[7] bondi programming language. `www-staff.it.uts.edu.au/~cbj/bondi`.

[8] Reg Braithwaite. The significance of the meta-circular interpreter. http://weblog.raganwald.com/2006/11/significance-of-meta-circular_22.html, November 2006.

[9] Jacques Carette, Oleg Kiselyov, and Chung chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[10] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In *Proceedings of ESOP'03, European Symposium on Programming*, pages 335–347. Springer-Verlag (*LNCS*), 2003.

[11] Sir Arthur Conan Doyle. *The Sign of the Four.* Lippincott's Monthly Magazine, February 1890.

[12] Brendan Eich. Narcissus. `http://mxr.mozilla.org/mozilla/source/js/narcissus/jsexec.js`, 2010.

[13] J-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[14] Masami Hagiya. Meta-circular interpreter for a strongly typed language. *Journal of Symbolic Computation*, 8(6):651–680, 1989.

[15] R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-calculus.* Cambridge University Press, 1986.

[16] Barry Jay. *Pattern Calculus: Computing with Functions and Structures.* Springer, 2009.

[17] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 2011. To appear. `http://www-staff.it.uts.edu.au/~cbj/Publications/factorisation.pdf`.

[18] Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.

[19] C.B. Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.

[20] S.C. Kleene. *Introduction to Methamathematics*. van Nostrand, 1952.

[21] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, pages 340–353, 1936.

[22] Konstantin Läufer and Martin Odersky. Self-interpretation and reflection in a statically typed language. In *Proceedings of OOPSLA Workshop on Reflection and Metalevel Architectures*. ACM, October 1993.

[23] Oleg Mazonka and Daniel B. Cristofani. A very short self-interpreter. http://arxiv.org/html/cs/0311032v1, November 2003.

[24] J.C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 1985.

[25] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *Journal of Functional Programming*, 2(3):345–363, 1992. See also DIKU Report D–128, Sep 2, 1994.

[26] Torben Æ. Mogensen. Linear-time self-interpretation of the pure lambda calculus. *Higher-Order and Symbolic Computation*, 13(3):217–237, 2000.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.

[28] Matthew Naylor. Evaluating Haskell in Haskell. *The Monad.Reader*, 10:25–33, 2008.

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

[30] Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 130–143, New York, NY, USA, 2005. ACM.

[31] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. Typed self-representation. In *Proceedings of PLDI'09, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303, June 2009.

[32] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740. ACM Press, 1972. The paper later appeared in Higher-Order and Symbolic Computation, 11, 363–397 (1998).

[33] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *OOPSLA Companion*, pages 044–953, 2006.

[34] Andreas Rossberg. HaMLet. http://www.mpi-sws.org/ rossberg/hamlet, 2010.

[35] Fangmin Song, Yongsen Xu, and Yuechen Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171–181, March 2000.

[36] Walid Taha, Henning Makholm, and John Hughes. Tag elimination and Jones-optimality. In *Proceedings of PADO'01, Programs as Data Objects, Second Symposium*, pages 257–275, 2001.

[37] Terese. *Term Rewriting Systems*, volume 53 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[38] John Tromp. Binary lambda calculus and combinatory logic. In *Kolmogorov Complexity and Applications*, 2006. A Revised Version is available at http://homepages.cwi.nl/ tromp/cl/LC.pdf.

[39] J. B. Wells. Typability and type checking in the second-order λ-calculus are equivalent and undecidable. In *Proceedings of LICS'94, Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994.

[40] Wikipedia. Pypy. http://en.wikipedia.org/wiki/PyPy, 2010.

[41] Wikipedia. Rubinius. http://en.wikipedia.org/wiki/Rubinius, 2010.

[42] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Proceedings of PEPM'07, ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.