Self-Quotation in a Typed, Intensional Lambda-Calculus

Barry Jay

Centre for Artificial Intelligence
School of Software
University of Technology Sydney
Sydney
Australia

Abstract

Intensional lambda-calculus adds intensional combinators to lambda-calculus to facilitate analysis. In particular, they are used to factorise data structures and programs into their components, which can be used to convert abstractions into combinators. This paper shows how to type an intensional lambda-calculus using no more than the types of System F. Even the quotation function used to access program syntax can be defined and typed, as can its inverse: the calculus supports typed self-quotation. Thus, one may freely alternate between program analysis and program execution. Proofs of all results have been verified in Coq.

Keywords: lambda-calculus, combinators, self-interpretation, type theory

1 Introduction

The default implementation of a programming language is written in another, lower level, language in which programs can be analysed. In turn, this may prove to be an intermediate language that, through a sequence of translations, is eventually implemented in, say, machine code. To avoid this chain, it is routine to provide a *self-interpreter* (see [Kleene(1936),Reynolds(1972),Barendregt(1991),Mogensen(1992),Mogensen(2000)] and [Berarducci and Böhm(1993),Song et al.(2000),Jay and Palsberg(2011)] and [Brown and Palsberg(2015),Brown and Palsberg(2016),Brown and Palsberg(2017)]), in which the language is implemented in itself, and so can analyse its own programs. Thus program syntax has two interpretations in the calculus. The standard one yields a reducible term, with all possible evaluation strategies in play. The novel one is *quotation* which represents a program as a data structure, open to analysis. The simplest form of analysis merely unquotes the program, to recover the standard interpretation. More generally, analysis can be used to optimise, or to impose an

[1] Email:Barry.Jay@uts.edu.au

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

evaluation strategy, before converting to an executable. Let us consider some of the desirable properties that quotation should satisfy.

First, there should be no limits on the nature of the analyses that can be done. It is not enough to support a single self-interpreter, or a small suite of analyses. For example, it should be possible to decide syntactic equality of programs, or to perform a dead-code analysis.

Second, opportunities for optimisation are maximized when the language is represented by a confluent calculus, e.g. a λ-calculus [Barendregt(1984)], so that one is free to manipulate the evaluation strategy.

Third, the calculus should be typed. This provides a sanity check on the whole approach. In particular, it is common to model each stage of interpretation by rising one level in a type hierarchy. This is inherently difficult, and often limits the nature of the analyses that are possible. To put it another way, if the analysis is to be done in the source language, all of the usual arguments for typed programming languages also apply here.

Fourth, analysis should be *dynamic*, and not merely static. For example, in *staged computation* [Shields et al.(1998),Davies and Pfenning(2001)] the results from computation in one stage may be analysed in the next. More generally, this is required for dynamic program analysis. Despite its significance, this challenge is generally considered too hard in the context of self-interpretation, because it requires that quotation be definable within the language itself.

From the viewpoint of λ-calculus, it is not even clear that such *self-quotation* make sense. After all, quotation cannot act on reducible terms, since reduction changes the nature of the syntax produced by quotation. The solution is to identify *programs* with *closed normal forms*, whose reduction begins only when they are applied to arguments. Even in this restricted domain, however, the task is beyond pure λ-calculus because it is inherently *extensional*: there is no pure λ-term that can reveal the internal structure of closed normal λ-abstractions in a uniform manner.

The solution is to add *intensionality* to the λ-calculus in the form of combinators that are able to query the internal structure of terms. The simplest such system is SF-calculus [Jay and Given-Wilson(2011)], whose *factorisation* operator F is able to expose the components of compound terms. Recent work [Jay(2016a)] introduces the λSF-calculus, in which abstractions can be factored, just as combinators can. This paper continues this development, to introduce an *intensional* λ-calculus that

- is confluent
- is strongly typed
- defines program equality, and
- defines self-quotation, and self-interpreters.

Despite this expressive power, the actual machinery is comparatively lightweight. The types are given by those of System F [Girard et al.(1989)], with coercions used to represent type abstraction and instantiation. The terms are given by adding the following operators to the λ-calculus:

$$O ::= S \mid K \mid A \mid Y \mid E \mid G \mid DS \mid DK \mid DA \mid DY \mid DE \mid DG \,.$$

The operators S and K are standard. The operator A is new, with reduction

rule $AMNP \longrightarrow MNP$. That is, A is a ternary identity operator: given three arguments it behaves as the identity. It could be defined in terms of S and K, but plays an important role in controlling evaluation, especially of fixpoints.

The fixpoint operator Y is not standard. Its reduction rule $YM \longrightarrow M(AYM)$ inserts a copy of A to prevent unwanted reductions. We could have added an operator I but instead let us define it as SKK since, for any P and Q we have $SKPQ \longrightarrow KQ(PQ) \longrightarrow Q$ which shows that SKP is an identity function for any term P. Together, S, K, A and Y are the *extensional* operators, so called because they do not query the internal structure of their arguments. Their types are easily guessed from their reduction rules. The remaining operators are *intensional*; typing them is a little more of a challenge.

The operators E tests for equality of operators. Its principal type is

$$\mathsf{Ty}[E] = \forall X.\forall Y.\forall Z.X \to Y \to Z \to Z \to Z$$

which is equivalent to $\forall X.\forall Y.X \to Y \to \mathsf{Bool}$ where Bool is defined to be $\forall Z.Z \to Z \to Z$. Note that the operators being compared may have different types, so that, for example, ESK has the same type as its reduct KI.

The operator G is a variant of the factorisation operator from SF-calculus. Its sole reduction rule is

$$GMP \longrightarrow MP\rceil\lceil P \quad (P \text{ is factorable})$$

where $P\rceil$ and $\lceil P$ are the *left* and *right components* of P. If P is, say, SP_1P_2 then its components are SP_1 and P_2. The complete account of factorable forms is made more complicated by the need to factor abstractions. The term M in the rule must be sufficiently polymorphic to handle the components of P, no matter what their types prove to be. Hence the principal type of G is

$$\mathsf{Ty}[G] = \forall X.\forall Y.(\forall Z.(Z \to X) \to Z \to Y) \to X \to Y\ .$$

Thus, if $P : X$ in the rule above has components $P_1 : Z$ and $P_2 : Z \to X$ then $MP_1P_2 : Y$ provided that $M : \forall Z.(Z \to X) \to Z \to Y) \to X \to Y$.

The remaining operators are the *case operators* corresponding to the basic operators we have already met. Each one has a pair of reduction rules. For example, we have

$$\begin{aligned} &DS\,MNS \longrightarrow M \\ &DS\,MNP \longrightarrow NP \quad (P \neq S \text{ is factorable})\ . \end{aligned}$$

It represents a pattern-matching function that maps S to M and applies N to anything else. Its type is

$$\mathsf{Ty}[DS] = \forall X.\mathsf{Ty}[S] \to (\forall Y.Y \to Y) \to X \to X$$

Note the use of the principal type of S in the principal type of DS. It is tempting to replace the family of case operators with a single operator D whose applications yield DS, etc. However, the type of D would require a means of referencing the principal types of the operators that excludes other types. While this is possible, it would require additions to the type system.

Factorisation supports a divide-and-conquer approach to program analysis that is quite direct. For example, self-quotation can be described informally as a pattern-matching function that maps operators to themselves, and compounds of the form

MN to $A(\mathsf{quote}\ M)(\mathsf{quote}\ N)$. As a term of the calculus, it is given by

$$\mathsf{quote} = AY(\lambda q.\lambda p.Eppp(G(\lambda x.\lambda y.A(qx)(qy))p)\ .$$

This is a closed normal form, since the application of A blocks the reduction of the fixpoint operator Y until an argument has been supplied. E is used to check if p is an operator. If not then G is used to factorise p and recurse. Further, it has type

$$\mathsf{quote} : \forall X.X \to X\ .$$

All proofs in the paper have been verified using the Coq proof assistant [Barras et al.(1997)].

The structure of the paper is as follows. Section 1 is the introduction. Section 2 introduces the compounds. Section 3 introduces the reduction rules of the calculus. Section 4 introduces the types of the calculus. Section 5 shows how to define and type equality of programs. Section 6 defines quoting and unquoting. Section 7 discusses verification in Coq. Section 8 considers related work. Section 9 draws conclusions.

2 Factorable Forms

This section develops the machinery necessary to describe the reduction rules given in the next section. At issue is that the reduction rules for the intensional operators have side-conditions requiring that arguments be factorable forms, i.e. either operators or *compounds*. The latter include some abstractions, whose right components are given by *star abstraction*. So the order of business will be: star abstraction; components; and then compounds.

2.1 Star Abstraction

The syntax of terms was given in Section 1. The *star abstraction* $\lambda^*x.M$ of M with respect to x is defined by

$$\begin{aligned}
\lambda^*x.x &= I \\
\lambda^*x.y &= Ky \quad (y \neq x) \\
\lambda^*x.O &= KO \quad (O \text{ an operator}) \\
\lambda^*x.\lambda y.M &= A(\lambda x.\lambda^*y.M) \\
\lambda^*x.MN &= S(\lambda x.M)(\lambda x.N)\ .
\end{aligned}$$

This definition modifies the traditional definition of $\lambda^*x.M$ for combinators M (see, e.g. [Hindley and Seldin(1986)]) in two ways. First, when the body is an application MN the result uses $\lambda x.N$ instead of $\lambda^*x.N$. To see why this is necessary, consider $\lambda^*x.S(KN_1N_2)$. Now $S(KN_1N_2)$ is not a redex, so it is safe to separate S from KN_1N_2 but $\lambda^*x.KN_1N_2$ breaks the redex KN_1N_2. Thus a recursive call to λ^*x would here be unsafe. Second, there needs to be a rule for λ^*x when the body is an abstraction $\lambda y.M$. The result is $\lambda x.\lambda^*y.M$ and not $\lambda^*x.\lambda^*y.M$ since keeping track of information requires that only one abstraction is eliminated at a time, namely, the innermost one.

Here are some simple examples of star abstraction. In SKI-calculus, the λ-abstraction $\lambda x.\lambda y.y$ can be represented by

$$\lambda^*x.\lambda^*y.y = \lambda^*x.I = KI$$

where λ^* is used to convert abstractions into combinators in the traditional manner. In λSF-calculus, $\lambda x.\lambda y.y$ is already a closed normal form. However, its factorisation will introduce $\lambda^* x.\lambda y.y$ which is calculated as follows:

$$\lambda^* x.\lambda y.y = \lambda x.\lambda^* y.y = \lambda x.I \ .$$

This has eliminated the innermost abstraction, just like the first step in the calculation of $\lambda^* x.\lambda^* y.y$ in SKI-calculus. A second factorisation exposes

$$\lambda^* x.SKK = S(\lambda x.SK)(\lambda x.K) \ .$$

Further factorisation eliminates the remaining abstractions to produce the combinator

$$S(S(KS)(KK))(KK)$$

which when applied to terms M and N reduces to N, just like the original abstraction. Of course, it is much bigger than the original term, as it does not take advantage of the standard optimisation, in which $\lambda^* x.I$ takes advantage of the fact that x is not free in I to produce KI. Optimizing this has been addressed in λSF-calculus [Jay(2016a)].

2.2 Components

For convenience, the definitions of components are given for arbitrary terms, whether they prove to be compounds or not.

The *left component* $M\rceil$ and *right component* $\lceil M$ of a term M are defined as follows

$$\begin{array}{rcll} (\lambda x.M)\rceil & = & \mathsf{abs\_left} & \qquad \lceil(MN) = N \\ (MN)\rceil & = & M & \qquad \lceil(\lambda x.M) = \lambda^* x.M \\ M\rceil & = & KM \quad \text{(otherwise)} & \qquad \lceil M = M \quad \text{(otherwise.)} \end{array}$$

where $\mathsf{abs\_left} = I$ is used as the left component of an abstraction. The key point about $\mathsf{abs\_left}$ is that it cannot be the left component of an application to some N since $\mathsf{abs\_left}\ N = I\ N$ is a redex. In general, words in sans-serif, such as $\mathsf{abs\_left}$ may be used to name particular terms of the calculus, as well as the meta-variables M and N, etc. Perhaps surprisingly, the components of an operator O are defined to be KO and O. It will follow that GMO reduces to $M(KO)O$. If, as is usual, this is not desirable, then operators must be excluded by first using E to identify them, as in the example of quote introduced earlier.

2.3 Compounds

In combinatory calculi, the compounds are exactly the partially applied operators. For example, in SF-calculus, the compounds are all terms of the form SM or SMN or FM or FMN. These forms are compounds in λSF-calculus, too, as are all the other partially applied operators, such as those of the form KM and EM and $DSMN$. All other compounds will be abstractions $\lambda x.M$ whose decomposition is safe because either M is already factorable, or outermost reduction in M awaits the instantiation of x.

In turn, this requires keeping track of which variables require values in order for head reduction to proceed, i.e. which variables are *active* in a term. More generally,

```
Inductive status_val :=
| Reducible : status_val
| Lam : nat -> status_val     (* the active variable *)
| Active : nat -> status_val  (* the active variable *)
| Ternary_op (* S, A *)
| Binary_op0 (* K, 0 eager arguments *)
| Binary_op2 (* E, 2 eager arguments *)
| Binary_op1 (* G, 1 eager argument  *)
| Ternary_op1 (* DO *)
| Unary_op (* Y *)
| Lazy2 (* need two args *)
| Lazy1 (* need one arg *)
| Eager2 (* DOM *)
| Eager (* EO *)
.
```

Fig. 1. Status Values in Coq

```
Fixpoint status (M: lamSF) :=
 match M with
 | Ref i => Active i
 | Op Sop => Ternary_op
 | Op Aop => Ternary_op
 | Op Kop => Binary_op0
 | Op Eop => Binary_op2
 | Op Gop => Binary_op1
 | Op Uop => Binary_op1
 | Op Yop => Unary_op
 | Op _ => Ternary_op1
 | Abs M1 =>
    match status M1 with
    | Reducible => Reducible
    | Lam 0 => Lazy1
    | Lam (S n) => Lam n
    | Active 0 => Lazy1
    | Active (S n) => Lam n
    | _ => Lazy1
    end
 | App M1 M2 =>
 match status M1 with
  | Reducible => Reducible
  | Lam _ => Reducible
  | Active n => Active n
  | Ternary_op => Lazy2
  | Binary_op0 => Lazy1
  | Binary_op2 =>
     match status M2 with
     | Reducible => Reducible
     | Lam n => Active n
     | Active n => Active n
     | Ternary_op
     | Binary_op0
     | Binary_op2
     | Binary_op1
     | Ternary_op1
     | Unary_op => Eager
     | _ => Reducible
     end
  | Binary_op1 => Eager
  | Ternary_op1 => Eager2
  | Unary_op => Reducible
  | Lazy2 => Lazy1
  | Lazy1 => Reducible
  | Eager2 => Eager
  | Eager =>
     match status M2 with
     | Lam n => Active n
     | Active n => Active n
     | _ => Reducible
     end
 end
end
```

Fig. 2. Term status in Coq

every term has a *status* which keeps track of all the information needed to determine if a term is factorable or not. This is complicated by the interplay between free variables and the intensional operators, as the latter are eager in the arguments that they query. To avoid transcription errors and other issues, it seems safest to give the definitions exactly as it appears in the Coq implementation. The *status values* are given in Figure 1. `Reducible` indicates that there is a head reduction. `Lam n` denotes a λ-abstraction whose active variable has de Bruijn index `n`. `Active n` denotes a term whose active variable is `n` but is not an abstraction. Rather it, it is either the nth variable, or an application. The other status values are used to characterise the factorable forms in what should be a self-explanatory manner.

The definition of the status of terms is given in Figure 2. To save space, the program has been broken into three columns, to be read successively. For example, the second column begins with a case for `App M1 M2` in the pattern-matching against `M` begun in the first column.

The status of an operator is one of `Ternary_op, Binary_op0, Binary_op2,`

$$
\begin{array}{rcl@{\qquad}rcll}
(\lambda x.M)N & \longrightarrow & \{N/x\}M & EOO & \longrightarrow & K & \\
SMNP & \longrightarrow & MP(NP) & EOP & \longrightarrow & KI & (P \neq O \text{ is factorable}) \\
KMN & \longrightarrow & M & EP & \longrightarrow & K(KI) & (P \text{ is a compound}) \\
AMNP & \longrightarrow & MNP & GMP & \longrightarrow & MP\rceil\lceil P & (P \text{ is factorable}) \\
YM & \longrightarrow & M(AYM) & DOMNO & \longrightarrow & M & \\
 & & & DOMNP & \longrightarrow & NP & (P \neq O \text{ is factorable})
\end{array}
$$

Fig. 3. Reduction Rules

`Binary_op1` or `Ternary_op1`. Now a term is a *compound* if its status is one of `Lazy2, Lazy1, Eager2` or `Eager`. It is *factorable* if it is a compound or an operator.

Although this is all hard to take in, it is worth noting that all closed, head normal forms of the pure λ-calculus are compounds. Here are some examples of compounds. The body of $\lambda x.xy$ has x active. The body of $\lambda x.\lambda y.x$ has x active. The body of $\lambda x.\lambda y.y$ is a compound. The body of $\lambda x.G$ is factorable. The body of $\lambda x.Gx$ is factorable. The body of $\lambda x.GMx$ has x active, since G is an intensional operator that needs to know the value of x to reduce. The body of $\lambda x.\lambda y.GM(GNx)$ has x active. Exy has x active. EOy has y active. The status of $E(SK)y$ is `Reducible`.

It follows that if M is factorable and $M \longrightarrow N$ then $M\rceil \longrightarrow N\rceil$ and $\lceil M \longrightarrow \lceil N$. That is, no redexes are broken by taking components of factorable forms. To put is another way, there is a derived reduction rule

$$(\xi)\ \frac{M \longrightarrow N}{\lambda^* x.M \longrightarrow \lambda^* x.N}\ (\lambda x.M \text{ is factorable.})$$

More important from a formal point of view is that the side-conditions on the reduction rules be decidable. Here, the side-conditions will depend upon equality of operators with factorable forms, and on being a factorable form. The latter is given by a function in Coq, and so is guaranteed to terminate.

3 Reduction

The reduction rules of intensional λ-calculus are given in Figure 3. The corresponding congruence is also denoted by $\longrightarrow$, whose reflexive, transitive closure is $\longrightarrow^*$.

3.1 Basic Results

Theorem 3.1 (confluence_lamSF_red) *Reduction in λSF-calculus is confluent.*

The *normal forms* are defined to be the variables, operators, abstractions of normal forms, and applications MN in which M and N are both normal and the status of MN is not `Reducible`.

Theorem 3.2 (irreducible_iff_normal) *A term is irreducible if and only if it is a normal form.*

A *program* is a closed normal form. If the goal is to analyse an open normal form, then it will be necessary to first bind all of its free variables.

Theorem 3.3 (programs_are_factorable) *All programs are factorable forms.*

Hence, any closed term of the form GMP must reduce. This is a form of progress result.

3.2 Unstar

As an example of the machinery in use, let us show how to reverse star abstraction, how to convert $\lambda^* x.M$ to $\lambda x.M$. The pattern-matching account is given by

$$\begin{array}{l}\mathsf{let\ rec\ unstar}\ x = \\ \quad \mathsf{match}\ x\ \mathsf{with} \\ \quad \mid O \Rightarrow O \\ \quad \mid Ax \Rightarrow \mathsf{abs\_A\ unstar}\ x \\ \quad \mid Kx \Rightarrow \mathsf{abs\_K}\ x \\ \quad \mid Sxy \Rightarrow \mathsf{abs\_S}\ x\ y\end{array}$$

where $\mathsf{abs\_S} = \lambda g.\lambda f.\lambda x.gx(fx)$ and $\mathsf{abs\_K} = \lambda x.\lambda y.x$ and $\mathsf{abs\_A} = \lambda u.\lambda x.\lambda y.u(x\ y)$. For example, the case $\mid Kx \Rightarrow \mathsf{abs\_K}\ x$ updates the operator K with the term $\mathsf{abs\_K}$ that has the same type. Similarly, S is updated by $\mathsf{abs\_S}$ and A by $\mathsf{abs\_A}$. The corresponding term of the calculus is

$$\mathsf{unstar} = AY(\lambda u.G(DA\ (\mathsf{abs\_A}\ u)(DK\ \mathsf{abs\_K}\ (G(DS\ \mathsf{abs\_S}\ I)))))$$

Theorem 3.4 (unstar_star) *The term* $\mathsf{unstar}\ (\lambda^* x.M)$ *reduces to* $\lambda x.M$.

3.3 Extensionality

Mathematically, two functions f and g are extensionally equivalent if they have the same graph. For unary functions, this means that $f\ x = g\ x$ for all x. In λ-calculi, extensionality is captured by adding the η-reduction rule

$$\lambda x.f\ x \longrightarrow f \quad \text{if } x \text{ is not free in } f.$$

When added to the basic λ-calculus, with just the β-rule, we get the $\lambda\beta\eta$-calculus, which is confluent. Define $=_{\beta\eta}$ to be the equivalence relation on λ-calculus induced by β-reduction and η-reduction. However, adding the η-rule to λSF-calculus is unsound, as it changes the status of terms.

A more useful relation is obtained by excluding the rules for the intensional operators (in the second column of Figure 3), to get the equivalence relation $\equiv_e$. Define terms M and N to be *extensionally equivalent* if $M \equiv_e N$. For example, we have the following theorem.

Theorem 3.5 (star_equiv_abs) $\lambda^* x.M \equiv_e \lambda x.M$ *for all terms* M.

4 Typing

There are three key challenges to typing an intensional λ-calculus:

(i) Type the factorisation operator, in this case G.

(ii) Factorise type abstractions and applications.

(iii) Type the equality operator E and the case operators.

In addressing these challenges we aim to keep the type system as simple as possible.

First consider GMP of type T where $P : U$ is factorable. There must be a type Z such that $\lceil P : Z$ and so $P \rceil : Z \to U$. Hence, M must have type $(Z \to U) \to Z \to T$.

$$\frac{}{\forall X.T \prec \{U/X\}T} \qquad \frac{}{X \prec X} \qquad \frac{X \notin U}{\forall X.U \to T \prec\!\!\prec U \to \forall X.T} \qquad \frac{}{X \prec\!\!\prec X}$$

$$\frac{U_1 \prec T_1 \quad T_2 \prec U_2}{T_1 \to T_2 \prec U_1 \to U_2} \qquad \frac{T \prec U}{\forall X.T \prec \forall X.U} \qquad \frac{T_2 \prec\!\!\prec U_2}{T_1 \to T_2 \prec\!\!\prec T_1 \to U_2} \qquad \frac{U \prec\!\!\prec T}{\forall X.U \prec\!\!\prec \forall X.T}$$

Fig. 4. Type instantiation ($\prec$) and pushing ($\prec\!\!\prec$)

Further, there is no way to determine the type Z in advance. Hence, M must be polymorphic, of quantified type $\forall Z.(Z \to U) \to Z \to T$ so that

$$G : (\forall Z.(Z \to U) \to Z \to T) \to U \to T\ .$$

More generally, we have

$$G : \forall X.\forall Y.(\forall Z.(Z \to X) \to Z \to Y) \to X \to Y.$$

The inherent need for polymorphism, and the quantification of type variables within a function type, means that neither simple types nor those of the Hindley-Milner system are expressive enough: the type system must be at least as expressive as System F. Indeed, this is enough.

Second, there is the challenge of representing type generalization and instantiation. In the original System F [Girard et al.(1989)], these changes to the types are made explicit in the terms. For example, if $t : T$ then $\Lambda X.t : \forall X.T$ and then $(\Lambda X.t)U : \{U/X\}T$. However, if such operations are made explicit, then it is not clear how to factorise a type application of the form tU. In particular, if it is deemed to have components t and U then the type U must also be a term in its own right. This is an intriguing possibility, but would take us far beyond the simplicity of System F. Instead, let us make the type-level operations implicit, so that type coercions can modify the type of a term without change to the term itself. The basic machinery for doing this is already well known, but requires some small modifications for our purposes.

For example, consider a compound $PQ : T$ where Q has type U and P has type $U \to T$. Further, suppose the type is generalized to be $PQ : \forall X.T$. Now, we have $Q : \forall X.U$ and so factorisation requires that $P : (\forall X.U) \to (\forall X.T)$. The solution is to perform the following coercions:

$$\dfrac{\dfrac{\dfrac{\vdash P : U \to T}{\vdash P : (\forall X.U) \to T}\ \text{(contravariance)}}{\vdash P : \forall X.(\forall X.U) \to T}\ \text{(generalization)}}{\vdash P : (\forall X.U) \to (\forall X.T)}\ \text{(push).}$$

The first uses contravariance of function types and type variable instantiation. The second generalizes the type variable X. The third step is not quite standard, as it involves *pushing* a quantifier across a function type, using the rule,

$$\forall X.(U \to T) \prec\!\!\prec U \to \forall X.T \quad (X \notin U)$$

where X is not free in U. The details of the rules for *instantiation* $T \prec T'$ a type T to a type T', and for *pushing* $T \prec\!\!\prec T'$ a type T to a type T' are given in Figure 4.

Typing judgments are of the form $\Gamma \vdash t : T$ where Γ is a context, and t is a term and T is a type. In turn, a context Γ is a collection of distinct term variables with types $x_i : U_i$. The type derivation rules are given in Figure 5. All contexts therein are assumed to be well-formed. There is one derivation rule for each term form,

$$\frac{}{\Gamma \vdash x : U}\ (x : U \in \Gamma) \qquad \frac{}{\Gamma \vdash O : \mathsf{Ty}[O]} \qquad \frac{\Gamma \vdash t : U \to T \quad \Gamma \vdash u : U}{\Gamma \vdash t\ u : T} \qquad \frac{\Gamma, x : U \vdash t : T}{\Gamma \vdash \lambda x.t : U \to T}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T'}\ T \prec T' \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : T'}\ T \nprec T' \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash t : \forall X.T}\ (X \notin \Gamma)$$

Fig. 5. Type Derivation

$$\begin{aligned}
\mathsf{Ty}[S] &= \forall X.\forall Y.\forall Z.(X \to Y \to Z) \to (X \to Y) \to (X \to Z) \\
\mathsf{Ty}[K] &= \forall X.\forall Y.X \to Y \to X \\
\mathsf{Ty}[A] &= \forall X.\forall Y.(X \to Y) \to X \to Y \\
\mathsf{Ty}[Y] &= \forall X.(X \to X) \to X \\
\mathsf{Ty}[E] &= \forall X.\forall Y.X \to X \to Y \to Y \to Y \\
\mathsf{Ty}[G] &= \forall X.\forall Y.(\forall Z.(Z \to X) \to Z \to Y) \to X \to Y \\
\mathsf{Ty}[DO] &\quad \forall X.\mathsf{Ty}[O] \to (\forall Y.Y \to Y) \to X \to X\ .
\end{aligned}$$

Fig. 6. Operator Types

plus three rules for implicitly manipulating types, by instantiation, pushing, or by generalizing from a type T to its quantification $\forall X.T$. All that remains is to type the operators. Each operator O has a designated type $\mathsf{Ty}[O]$ as given in Figure 6.

The expressive power of type derivation is illustrated by the following theorems.

Theorem 4.1 (unstar_type) $\mathsf{unstar} : \forall X.X \to X$.

Theorem 4.2 (reduction_preserves_derivation) *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : T$.

Proof. Consider $(\lambda x.t)u \longrightarrow \{u/x\}t$. This preserves typing since substitution preserves typing. The rule for F applied to an abstraction preserves typing since star abstraction does. The other cases are routine. □

5 Equality

The expressive power the overall approach is illustrated by a couple of key examples. This section presents an example of a query, namely equality. The next section considers some updates, namely quote and unquote.

It follows from Theorem 3.3 that equality can be decided by comparing operators and compounds. The algorithm is as follows. Atomic equality is decided by E. Operators and compounds are never equal. Compounds are equal if their components are. Informally, equality is given by the following, nested pattern-matching function:

$$\begin{array}{l}
\mathsf{let\ rec\ equal}\ x\ y = \\
\mathsf{match}\ x\ \mathsf{with} \\
\quad \mid O \Rightarrow EOy \\
\quad \mid x_1x_2 \Rightarrow \mathsf{match}\ y\ \mathsf{with} \\
\qquad\qquad \mid O \Rightarrow KI \\
\qquad\qquad \mid y_1y_2 \Rightarrow (\mathsf{equal}\ x_1\ y_1)\ (\mathsf{equal}\ x_2\ y_2)\ (KI)\ .
\end{array}$$

The corresponding term of the calculus is

$$\mathsf{equal} = AY(\lambda e.\lambda x.\lambda y.Exx(Exy)(G(\lambda x_1.\lambda x_2.Eyy(KI)(G(\lambda y_1.\lambda y_2.(ex_1y_1)(ex_2y_2)(KI))y))x))\ .$$

Theorem 5.1 (equal_type) equal *has type* $\mathsf{Ty}[E]$.

Theorem 5.2 (equal_programs) equal $M\ M \longrightarrow^* K$ *for all programs* M.

Theorem 5.3 (unequal_programs) equal $M\ N \longrightarrow^* KI$ *for all distinct programs* M *and* N.

Proof. The proof is by induction on the rank of M, as defined in the Coq implementation. The only case of interest arises when M is an abstraction and N is an application. Now the left component of N cannot be abs_left since any application of abs_left reduces, and so the left components of M and N cannot be equal. □

6 Quoting and Unquoting

The account of quotation given in Section 1 is a slight simplification, since it does not begin with the meta-function for quotation, here called mquote. As a pattern-matching function mquote is given by

$$\begin{array}{l}\mathsf{let\ rec\ mquote} = \\ \mid O \Rightarrow O \\ \mid \lambda x.M \Rightarrow A(\mathsf{mquote\ abs\_left})(\lambda^* x.M) \\ \mid MN \Rightarrow A(\mathsf{mquote}\ M)(\mathsf{mquote}\ N)\ .\end{array}$$

Of course, abs_left and $\lambda^* x.M$ are the components of $\lambda x.M$ so that, from the viewpoint of factorisation, the second and third cases above can be handled by a single line of code. Consequently, we have the following theorems.

Theorem 6.1 (quote_is_definable) *Quotation of programs is definable by* quote.

Theorem 6.2 (quote_type) quote *has type* $\forall X.X \to X$.

The latter theorem implies that no type information has been lost during quotation. The type is also very simple. However, using a quotation instead of the source program will not yield a type error. If this is a concern, the solution will be to make the system a little more complicated, e.g. by giving the operator A the type $[X \to Y] \to [X] \to [Y]$ where $[T]$ is a new type form, the expression type of T.

This theorem also illustrates that the type $\forall X.X \to X$ supports many useful programs, including all database updates, and so is richer than a unit type. Thus, datatypes must be introduced separately to function types, just as is done in practice. For example, given a type of natural numbers, it is routine to define a function that computes the size of a program, or its Gödel number.

The informal account of unquote is equally straightforward, being given by the pattern-matching function

$$\begin{array}{l}\mathsf{let\ rec\ unquote} = \\ \mid O \Rightarrow O \\ \mid A(\mathsf{quote\ abs\_left})N \Rightarrow \mathsf{unstar}(\mathsf{unquote}\ N) \\ \mid AMN \Rightarrow (\mathsf{unquote}\ M)(\mathsf{unquote}\ N)\ .\end{array}$$

There are two points of interest when unquoting a term of the form AMN. First, the generic equality (and, ultimately, E) is used to decide if unquote M is abs_left. Second, DA is used to replace the application of A with an application of I.

```
Theorem confluence_lamSF_red
Theorem irreducible_iff_normal
Theorem programs_are_factorable
Theorem unstar_star
Theorem star_equiv_abs
Theorem unstar_type
Theorem reduction_preserves_derivation
Theorem equal_type
Theorem equal_programs
Theorem unequal_programs
Theorem quote_is_definable
Theorem quote_type
Theorem unquote_type
Theorem unquote_quote
```

Fig. 7. Theorems verified in Coq

Theorem 6.3 (unquote_type) unquote *has type* $\forall X.X \to X$.

Theorem 6.4 (unquote_quote) *For all programs* M, unquote (quote M) *reduces to* M.

7 Verification in Coq

All proofs in the paper have been verified using the Coq proof assistant [Barras et al.(1997)], as supplied in the supplementary materials [Jay(2017)]. For ease of reference, the same theorem names are used in both places.

8 Related Work

8.1 Self-Interpretation

First, note that till now, no self-interpreter of any kind has supported self-quotation. The closest prior work concerns self-interpretation for typed, confluent calculi. Jay and Palsberg [Jay and Palsberg(2011)] created self-interpreters for a typed combinatory calculus based on SF-calculus. The lack of first-class λ-abstractions was its primary limitation, here overcome. Also, its approach to typing equality was more general than here, but at the cost of introducing some obscurity to the type system. By introducing case operators to handle updates, the system has been simplified without losing any of the motivating examples. The most recent work on typed self-interpretation is by Brown and Palsberg [Brown and Palsberg(2015),Brown and Palsberg(2016),Brown and Palsberg(2017)]. They construct typed self-interpreters for System Fω. This is a good example of the use of a type hierarchy in which the quotation of programs at one level at typed at the next level. By contrast, the system here has a flat type hierarchy.

8.2 Partial Evaluation

The identification of programs with closed normal forms suggests new approaches to partial evaluation of λ-calculi, both static and dynamic [Jones et al.(1993)]. A running theme in the partial evaluation of typed languages is to eliminate the overheads associated with type tags [Pašali et al.(2002)] in the quest for *Jones optimality* [Taha et al.(2001),Danvy and Martínez López(2003)]. The analogue of tags in our setting are the uses of A and abs_left in the definition of quote. There is a minimal sense in which these carry type information, the same sense in which a λ does, but

overall, the types barely figure in the representation. Perhaps the interpretation overhead can be completely eliminated by applying unquote so that, in this sense, Jones optimality will follow from this more general principle.

8.3 Staged Evaluation

Staged evaluation [Shields et al.(1998),Davies and Pfenning(2001)] allows each stage of program execution to analyse the syntax of subsequent stages. Since programs can now be identified with data structures, the divisions between stages are no longer fundamental, as one may quote and unquote at leisure.

8.4 Higher-Order Abstract Syntax

Intensionality and quotation play a role in higher-order abstract syntax (see, e.g. [Schürmann et al.(2001)]) but the relationship to intensional λ-calculus has yet to be explored.

8.5 Program Analysis

The above examples illustrate how analyses built from queries and updates can be performed upon closed normal forms, without the need for any meta-level operations such as quotation, and this in a language whose type system is completely standard. The question, then, is whether it is legitimate to identify the programs with the closed normal forms. To completely resolve the issue will require the development of an actual programming language, as opposed to a calculus. Nevertheless, the evidence to hand is suggestive.

First consider the restriction to closed terms. This restriction will not be very controversial, but if open terms are of interest they can be handled by analyzing the closed terms obtained by abstracting with respect to the free variables. This reduces to the problem discussed above.

Now consider the restriction to normal forms, or strongly normalizing terms. It is instructive to compare with the extension of System F obtained by adding a fixpoint operator. In the standard account, any term containing an application of the fixpoint operator will have infinite reduction sequences. However, in System F extended with our A and Y we can define recursive functions that do not have infinite reduction sequences until applied to arguments. Indeed, by adding suitable intensional operators, it should be possible to support a generous class of queries and updates within a strongly normalising calculus, similar to the *query calculus* [Jay(2009)]. In this manner, a self-enactor may be a program, and so may be self-applied.

8.6 Foundations of Computing

This work continues the development of confluent calculi that are more expressive than λ-calculus, including *pattern calculus* [Jay(2004),Jay and Kesner(2009),Jay(2009)], the combinatory calculus SF-calculus [Jay and Given-Wilson(2011),Jay and Palsberg(2011)] and the untyped λSF-calculus [Jay(2016a)]. The various pattern calculi support queries that are

uniformly applied to arbitrary data structures, but not to λ-abstractions. The SF-calculus supports queries that are uniformly applied to arbitrary closed normal forms. The λSF-calculus extends this approach to a λ-calculus. It most clearly exposes the limitations of pure λ-calculus, its inability to support intensional computations as well as extensional ones. Since this conflicts with the traditional account of the nature of computation, we have been especially careful, by verifying the results in Coq [Barras et al.(1997)], and pinpointing the source of the traditional error, within conflicting accounts of λ-definability [Jay and Vergara(2017)].

9 Conclusions

The great strength and weakness of λ-calculus is that is *extensional*. Hiding the body of a λ-abstraction supports the separation of concerns, supports modular program development, and allows for the representation of data structures as higher-order polymorphic functions in, say, System F. However, intensionality is central to program analysis: we must be able to look under the λ to analyse a program. All of the difficulties of self-interpretation in λ-calculus, especially as manifested in typed calculi, spring from this source.

It follows that the solution is to add intensionality from the beginning, starting with a factorisation operator, say G, so that programs can be analysed through a general divide-and-conquer approach. Typing introduces a delicate challenge: how to exploit type information obtained during analysis. Following the database approach, we restrict our attention to two sorts of queries, queries and updates. Queries are easily typed using the operator E, while updates are typed by using the case operator DO associated to each basic operator O. This approach has been illustrated by accounting for program equality (a query), and for the updates that quote and unquote programs.

All the theorems have been verified in Coq [Jay(2017)].

In this manner, all program analyses based on queries and updates can be typed, and this using no more than the types of System F. The ability to define quotation as a term, to support self-quotation, means that program analysis can be performed dynamically, after, say, some profiling.

Acknowledgments Thanks to the anonymous referees for their helpful suggestions.

References

[Barendregt(1984)] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984. revised edition.

[Barendregt(1991)] H. Barendregt. Self-interpretations in lambda calculus. *J. Functional Programming*, 1 (2):229–233, 1991.

[Barras et al.(1997)] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. URL https://hal.inria.fr/inria-00069968. Projet COQ.

[Berarducci and Böhm(1993)] A. Berarducci and C. Böhm. *A self-interpreter of lambda calculus having a normal form*, pages 85–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993. ISBN 978-3-540-47890-4.

[Brown and Palsberg(2015)] M. Brown and J. Palsberg. Self-representation in girard's system u. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 471–484, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9.

[Brown and Palsberg(2016)] M. Brown and J. Palsberg. Breaking through the normalization barrier: A self-interpreter for f-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 5–17, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2.

[Brown and Palsberg(2017)] M. Brown and J. Palsberg. Typed self-evaluation via intensional type functions. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 415–428. ACM, 2017.

[Danvy and Martínez López(2003)] O. Danvy and P. E. Martínez López. *Tagging, Encoding, and Jones Optimality*, pages 335–347. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Davies and Pfenning(2001)] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001. ISSN 0004-5411.

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

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

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

[Jay(2009)] B. Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.

[Jay(2016a)] B. Jay. Programs as data structures in λSF-calculus. *Electronic Notes in Theoretical Computer Science*, 325:221 – 236, 2016a. ISSN 1571-0661. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

[Jay(2017)] B. Jay. Typed LambdaFactor Calculus repository of proofs in Coq, January 2017. https://github.com/Barry-Jay/typed-lambdaFactor.

[Jay and Given-Wilson(2011)] B. Jay and T. Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.

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

[Jay and Palsberg(2011)] B. Jay and J. Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, pages 247–58, 2011.

[Jay and Vergara(2017)] B. Jay and J. Vergara. Conflicting accounts of λ-definability. *Journal of Logical and Algebraic Methods in Programming*, 87:1 – 3, 2017. ISSN 2352-2208.

[Jones et al.(1993)] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall International, 1993.

[Kleene(1936)] S. C. Kleene. λ -definability and recursiveness. *Duke Math. J.*, 2(2):340–353, 06 1936.

[Mogensen(1992)] T. 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.

[Mogensen(2000)] T. Æ. Mogensen. Linear-time self-interpretation of the pure lambda calculus. *Higher-Order and Symbolic Computation*, 13(3):217–237, 2000. ISSN 1573-0557.

[Pašali et al.(2002)] E. Pašali, W. Taha, and T. Sheard. Tagless staged interpreters for typed languages. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 218–229, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

[Reynolds(1972)] J. 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.

[Schürmann et al.(2001)] C. Schřmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1):1 – 57, 2001. ISSN 0304-3975.

[Shields et al.(1998)] M. Shields, T. Sheard, and S. Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 289–302, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3.

[Song et al.(2000)] F. Song, Y. Xu, and Y. Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171 – 181, 2000. ISSN 0304-3975.

[Taha et al.(2001)] W. Taha, H. Makholm, and J. Hughes. *Tag Elimination and Jones-Optimality*, pages 257–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.