Recursive Programs in Normal Form

author hidden

for reviewing purposes

Abstract

Recursive programs can now be expressed as normal forms within some rewriting systems, including traditional combinatory logic, a new variant of lambda-calculus called closure calculus, and recent variants of combinatory logic that support queries of internal program structure. In all these settings, partial evaluation of primitive recursive functions, such as addition, can reduce open terms to normal form without fear of non-termination. In those calculi where queries of program structure are supported, program optimizations that are expressed as non-standard rewriting rules can be represented as functions in the calculus, without any need for quotation or other meta-theory.

1. Introduction

Rewriting theory (Terese 2003) is an appealing foundation for partial evaluation and program analysis: partial evaluation may be expressed as reduction; and optimizations can be written as additional rewriting rules. One of the key difficulties with this approach is that recursive programs, even primitive recursive ones, may be represented by terms that have infinite reduction sequences. For example, in pure λ-calculus (Barendregt 1984) we may define addition of natural number as a fixpoint function $\mathsf{plus} = Yf$ where Y has the property that Yf reduces to $f(Yf)$. It follows that $\mathsf{plus}\ 3\ 4$ has, in addition to the normal form 7, some infinite reduction sequences, that continuously unfold Y while ignoring its arguments.

Standard evaluation strategies can avoid some of these difficulties. For example, suppose that the first argument of plus is static, say 2, and the second argument y is dynamic. Then left-most outermost reduction will reduce $2+y$ to $\mathsf{succ}\ (\mathsf{succ}\ y)$ (the second successor of y) as desired. The difficulties arise when the the first argument x is dynamic and the second argument is static, say 3. Now, partial evaluation of $x+3$ should have no effect, but left-most, outermost reduction will not terminate, since the fixpoint plus is still present, ready to be unfolded. Traditionally, *binding time analysis* (Jones et al. 1993) is required to avoid this unfolding.

Surprisingly, the lack of normal forms for recursive functions is an artifact of the pure λ-calculus: there are other rewriting systems in which all recursive functions can be expressed as normal forms. Indeed, this can be done in traditional combinatory logic, the SKI-calculus (Curry and Feys 1958; Hindley and Seldin 1986).

To be sure, the traditional account of recursion in combinatory logic is derived from that of λ-calculus approach, and inherits its weaknesses. However, combinatory logic supports much finer control over reduction than pure λ-calculus, which can be exploited to control fixpoint functions, as follows. Define

$$\begin{array}{rcl} A & = & (S(S(KS)(S(K(S(KS)))(S(S(KS)(S(KK) \\ & & \quad (S(KS)K)))(KK))))(K(KI))) \\ \omega_2 & = & S(K(SI))(S(KA)(S(KA)(SAI))) \\ Y_2 & = & A(A\omega_2\omega_2)\ . \end{array}$$

Now A has the property that AMN reduces to $S(KM)(KN)I$ which is a normal form if M and N are, but then

$$AMNP \longrightarrow^* MNP$$

for any combinator P. That is, A delays the application of M to N until a second argument P is supplied. It follows that ω_2 and the fixpoint function $Y_2 f$ have normal forms whenever f does. Nevertheless, the fixpoint property holds when the fixpoint function is given an argument, since

$$\begin{array}{l} Y_2 f \text{ has a normal form but} \\ Y_2 f x \longrightarrow^* f(Y_2 f)x \end{array}$$

for any f and x. On the right-hand side, the fixpoint function has no argument, and so cannot be reduced unless some reduction of f supplies it with an argument. In this manner, all reduction sequences of $3+4$ terminate. Similarly, if x is a variable and n is a normal form then $x+n$ has a normal form, which eliminates these difficulties for partial evaluation.

Actually, these results can be duplicated within λ-calculus, provided that the reduction rules are adapted to provide the desired fine control. The particular solution adopted here is the *closure calculus*, introduced in the accompanying paper, and recalled in Section 2. It has various interesting properties, but the key point for controlling fixpoints is that abstraction binds a non-empty sequence of variables, so that substitution is not performed until all bound variables are given values. More precisely, abstraction is given by *closures* of the form

$$\lambda x, xs[\sigma].t$$

where x, xs is a non-empty sequence of variables, reminiscent of *multi-variate* λ-calculus (Pottinger 1990) that are bound in the body t, and σ is the *environment*, reminiscent of *explicit substitution calculi* (Abadi et al. 1990; Kesner 2007). Each binding of a variable, say x, to an argument u is added to σ which may act on t only when all variables in xs take values.

Although this simplifies partial evaluation, the process of program analysis and optimization is still a challenge, as λ-calculi and combinatory logic are *extensional*, in that a function cannot query the internal structure of its argument u, except by the trial and error process of applying u to other values. Even equality of closed normal forms is not definable, which is a telling limitation (Jay and Vergara 2017). The traditional solution is to *quote* the program, to produce a syntax tree, which can then be analysed using specialized algorithms. Traditionally, this seemed unavoidable: how else could one analyse a term without a normal form? Now that programs have normal forms, the weakness of the extensional calculi is more obvious. The solution is to perform program manipulations within *intensional* calculi (Jay 2017b).

The simplest of the intensional calculi is SF-calculus, which consists of combinations of the operators S (from combinatory logic) and the *factorisation operator* F which is used to reveal the internal structure of combinations in normal form. By the way, we call them *combinations* rather than *combinators* since Curry and Feys (Curry and Feys 1958) were careful to reserve the latter nomenclature for combinations that correspond to λ-terms, and F is not one of those. Since K can be represented by FF and I by SKK, it follows that combinatory logic can be represented as a proper subsystem of SF-calculus. Thus, all programs can be represented as normal forms and all Turing-computable functions of natural numbers are definable. Further, there is an SF-term which converts normal forms to their Gödel numbers, and so all (Turing computable) program analyses are definable as SF-terms.

So, in theory, the challenge of program analysis has been solved. But this is far from showing how to do program analysis in practice. In particular, no-one wants to do program analysis on Gödel numbers! Indeed, few would want to perform program analysis on combinations, without access to λ-abstractions or closures. So the next challenge is to integrate abstraction and factorisation within a single calculus.

One solution is to simply combine pure λ-calculus and SF-calculus, by showing how to factorise abstractions in λSF-calculus (Jay 2016, 2017a). This works, but requires a large amount of meta-theory to determine when an abstraction is ready to be factorised. The accompanying paper shows how to represent closure calculus as a subsystem of SF-calculus, in that every reduction step of closure calculus can be represented by a sequence of reductions in SF-calculus. That is, the traditional abstraction can be represented by

$$\lambda x.t \quad \equiv \quad \mathsf{lam}\ (\mathsf{var}\ x)\ t$$

where var is used to convert a term x to a variable whose name is x, and lam forms the abstraction with respect to this name. In this manner, the process of abstraction in λ-calculus with its meta-level rules for substitution and variable renaming is explained by the axiomatic reduction rules of SF-calculus.

Unfortunately, the combination lam is huge, requiring around one hundred thousand operators. This does not bode well for program analysis, if the goal is to speed up program execution. One solution is to make the combinations var and lam into operators, which eliminates all code expansion. This might turn out to be best, but it does make it difficult to analyse closures. In this paper we will develop an intermediate position, in which new operators are introduced for some common, or expensive combinations, without privileging abstraction.

$FIESKA$-calculus has six operators, namely: the familiar S, K and I of combinatory logic; the operator A corresponding to the combinator A above; the factorisation operator F; and an equality operator E that is used for deciding equality of variable names, and for building pattern-matching functions.

The $FIESKA$-combination for the identity abstraction

$$\mathsf{lam}\ (\mathsf{var}\ x)\ (\mathsf{var}\ x)$$

requires 3383 operators, which is much better than one hundred thousand, but still impractical. Interestingly, it turns out that a simple extensional optimization (a variant of the η-rule) is enough to collapse the identity abstraction to I! More generally, this optimization can be used to eliminate some useless bindings from the environments of closures.

Now let us return to the challenge of program manipulation. An important class of optimizations can be represented as non-standard rewriting rules

$$p \Rightarrow s$$

in which p is a normal form. Then the corresponding optimization function

$$\mathsf{update}\{p, s\}$$

can be defined by pattern-matching, in the style of *pattern calculus* (Jay and Kesner 2009; Jay 2009). For example, the optimization

$$x + 0 \Rightarrow x\ .$$

yields $\mathsf{update}\{x + 0, x\}$ which is the recursive pattern-matching function given by the pseudo-code

```
let rec upd =
| $x + 0 \Rightarrow x$
| $y\ z \Rightarrow (\mathsf{upd}\ y)\ (\mathsf{upd}\ z)$
| $y \Rightarrow y$
```

Note that the pattern $y\ z$ does not match arbitrary applications, but only those which are *compounds*, i.e. partially applied operators, as will be recalled in the body of the paper. For example we have

$$\mathsf{update}\{x + 0, x\}(\mathsf{pair}\{y + 0, z + 0\}) \longrightarrow^* \mathsf{pair}\{y, z\}\ .$$

where $\mathsf{pair}\{y, z\}$ is an abstraction for pairing. This example shows how the optimization function traverses the program structure looking for opportunities to optimize.

The contributions of this paper to partial evaluation and program manipulation can be summarized as follows. It shows how to:

1. partially evaluate $2 + y$ to $\mathsf{succ}\ (\mathsf{succ}\ y)$ by reduction of both to normal form;
2. partially evaluate $x + 3$ to $x + 3$ by reduction of both to normal form; and
3. optimize $\mathsf{pair}\{y + 0, z + 0\}$ to $\mathsf{pair}\{y, z\}$ by applying a term of $FIESKA$-calculus.

These results will generalize from plus to arbitrary primitive recursive functions.

All named lemmas, theorems and corollaries have been verified in Coq, as shown in the files accompanying this paper.

The sections of the paper are as follows. Section 1 is the introduction. Section 2 recalls closure calculus and its use in partial evaluation to normal form. Section 3 introduces $FIESKA$-calculus. Section 4 shows how to optimize programs in $FIESKA$-calculus. Section 5 considers related work. Section 6 draws conclusions.

2. Closure Calculus

A full account closure calculus can be found in the accompanying paper. Instead, we will simply state the term forms and reduction rules, point out some of its novel features, state some theorems that have been verified in Coq, and present the evidence for its ability to reduce partial evaluation to normalization.

Assume given an unbounded collection of *variables* $x, y, z, \ldots$ whose equality is decidable. Finite sequences of variables are comma-separated, with the empty sequence being nil. The *terms* of closure calculus are given by the BNF

$$s, t, u, \sigma ::= x \mid s, t \mid t\ u \mid \lambda x, xs.[\sigma].t \mid I \mid x \mapsto t :: t \mid [\sigma]t\ .$$

The terms forms are, respectively: variables, tagged applications, applications, closures, the empty substitution, substitutions that are *extensions* and the application of a substitution to a term. Note that, although the symbol σ is used to indicate the expectation that the term is either I or an extension, this expectation is not enforced; the substitutions are in the same syntactic category as the terms.

$$\begin{array}{rcll}
x\;t & \longrightarrow & x, t & \\
(s,t)\;u & \longrightarrow & (s,t), u & \\
(\lambda x, \mathsf{nil}[\sigma].t)u & \longrightarrow & [x \mapsto u :: \sigma]t & \\
(\lambda x, y, ys[\sigma].t)u & \longrightarrow & \lambda y, ys[x \mapsto u :: \sigma].t & \\
I\;t & \longrightarrow & t & \\
(x \mapsto u :: \sigma)t & \longrightarrow & (\sigma\;t)(x\;t)(u\;t) & \\
[y]x & \longrightarrow & x & \\
[s,t]x & \longrightarrow & x & \\
[\lambda y, ys[\sigma_2].t]x & \longrightarrow & x & \\
[I]x & \longrightarrow & x & \\
[x \mapsto u :: \sigma]x & \longrightarrow & u & \\
[y \mapsto u :: \sigma]x & \longrightarrow & [\sigma]x & (y \neq x) \\
[\sigma](s,t) & \longrightarrow & [\sigma]s([\sigma]t) & \\
[\sigma](\lambda x, xs[\sigma_2].t) & \longrightarrow & \lambda x, xs[[\sigma]\sigma_2].t & \\
[\sigma]I & \longrightarrow & I & \\
[\sigma](x \mapsto u :: \sigma_2) & \longrightarrow & x \mapsto [\sigma]u :: [\sigma]\sigma_2 &
\end{array}$$

Figure 1. Reduction rules of closure calculus

Tagged application and application are left-associative, application binds tighter than tagged application, λ-abstractions bind as far to the right as possible, extension is right-associative. We may write $\lambda x[\sigma].t$ for $\lambda x, \mathsf{nil}[\sigma].t$ and $\lambda x, xs.t$ for $\lambda x, xs[I].t$ and $\lambda x.t$ for $\lambda x, \mathsf{nil}[I].t$. Also, we may write $x \mapsto u$ for $x \mapsto u :: I$.

Tagging is used to indicate that an application is not a redex. For example, $x\;t$ reduces to x, t for any variable x. In this manner, we can ensure that no application is a normal form unless it is a tagged application. Of course, substitution for x may convert $x\;t$ to a redex, so there is a rule $[\sigma](s,t) \longrightarrow ([\sigma]s)([\sigma]t)$ by which substitution converts a tagged application to an (untagged) application.

Closures bind a non-empty sequence x, xs of variables, and carry an *environment* σ which is to be thought of as a substitution waiting to act up on the *body* t of the closure. When the closure is applied to some term u then the binding of x to u is added to σ to get σ'. If xs is empty then σ' is applied to t; otherwise a new closure is formed. It is good style to ensure that every free variable of t is either bound in x, xs or is in the domain of σ, but this is not essential for any of the theorems below, since the reduction rules do not rename variables or refer to free variables.

Application of a substitution σ_2 to a closure $\lambda x, xs[\sigma].t$ reduces by

$$[\sigma_2]\lambda x, xs.[\sigma].t \longrightarrow \lambda x, xs.[[\sigma_2]\sigma].t$$

where σ_2 acts on the environment σ only. It is worth noting that $[\sigma_2]\sigma$ is *not* the composition of the substitutions. In particular, $[\sigma_2]I$ reduces to I and not σ_2. Thus if a closure has an empty environment then explicit substitutions have no impact, and this without having to inspect the closure body.

The reduction rules of closure calculus are given in Figure 1. The rules are a little heavy, since both $\sigma\;t$ and $[\sigma]\;t$ are terms, and have distinct meanings. It is done to make the relationship to combinations as simple as possible. In particular, the rule for reducing the direct application of an extension has been chosen to align with its representation in combinations, where S is used for pairing.

The *one-step* reduction relation, also denoted $\longrightarrow$ is obtained by applying a reduction rule to a sub-term. The *multi-step* reduction relation $\longrightarrow^*$ is the reflexive, transitive closure of the one-step relation.

THEOREM 1 (simple_confluence). *Reduction of closure calculus is confluent.*

Proof. There are no critical pairs.

This confluence result is an improvement on traditional theory and practice. In theory, the use of explicit substitutions generally creates critical pairs, so that some effort is required to ensure confluence, including substitution into the body of an abstraction. In turn, this triggers variable renaming (α-conversion) which is here avoided. In practice, evaluation strategies are used to avoid substituting under the λ but then confluence is lost, which makes optimization harder.

The *normal forms* are given by the BNF

$$n ::= x \mid n, n \mid \lambda x, xs[n].n \mid I \mid x \mapsto n :: n\;.$$

THEOREM 2 (simple_progress). *Every term of closure calculus is either normal or contains a redex.*

Proof. Induction on the structure of the term.

It follows that all terms of the form $[\sigma]t$ are reducible, so that in normal forms, all explicit substitutions are environments of closures.

2.1 Fixpoints

Closure calculus supports the traditional fixpoint operator, given by

$$\begin{array}{rcl}
\omega_1 & = & \lambda w, f.f, (w, w, f) \\
Y_1 & = & \omega_1 \omega_1\;.
\end{array}$$

Unlike the usual fixpoint operators of pure λ-calculus, Y_1 has a normal form $\lambda f[w \mapsto \omega_1].f, (w, w, f)$. Nevertheless, it satisfies the usual fixpoint property:

LEMMA 1 (fixpoint_property). $Y_1\;f \longrightarrow^* f(Y_1 f)$ *for all terms* f.

Closure calculus also supports fixpoint functions that have normal forms until given sufficient arguments. For example, we have

$$\begin{array}{rcl}
\omega_2 & = & \lambda w, f, x.\; f, (w, w, f), x \\
Y_2 & = & \omega_2 \omega_2\;.
\end{array}$$

Now $Y_2\;f$ has normal form

$$\lambda x.[f \mapsto f :: w \mapsto \omega_2].\; f, (w, w, f), x\;.$$

LEMMA 2 (fixpoint2_property).

$$Y_2\;f\;u \longrightarrow^* f(Y_2 f)u$$

for all terms f and u.

2.2 Arithmetic

Given the existence of fixpoint functions, the development of arithmetic in closure calculus should be routine, but the inability to substitute into abstraction bodies creates difficulties for Church's account of numerals and arithmetic. Fortunately, these problems do not arise when using the Scott numerals (see, e.g. (Mogensen 1992)).

Define zero and successor by

$$\begin{array}{rcl}
\mathsf{zero} & = & \lambda x, y.x \\
\mathsf{succ} & = & \lambda n, x, y.yn\;.
\end{array}$$

Then define the *Scott* numeral of a natural number n by

$\mathsf{scott}\;n =$
$\mathsf{match}\;n\;\mathsf{with}$
$\mid 0 \Rightarrow \mathsf{zero}$
$\mid S\;n_1 \Rightarrow \lambda x, y[z \mapsto \mathsf{scott}\;n_1].\;y, z\;.$

Then we can define my_plus by

$$Y_2(\lambda p, n.\; n, (\lambda x.x), (\lambda x, y[p \mapsto p].\;(\mathsf{succ}\;(p, x, y))\;.$$

Note how the inner closure, whose body is $\mathsf{succ}\ (p, x, y)$ must bind p to itself in the environment, so that substitution of myplus for p can take effect.

LEMMA 3 (my_plus_scott). *For all natural numbers m and n we have*

$$\mathsf{my\_plus}\ (\mathsf{scott}\ m)\ (\mathsf{scott}\ n) \longrightarrow^* \mathsf{scott}\ (m+n)\ .$$

LEMMA 4 (optimize_plus_2). *For all terms M we have*

$$\mathsf{my\_plus}\ (\mathsf{scott}\ 2)\ M \equiv \mathsf{succ}\ (\mathsf{succ}\ M)$$

The following theorem is a little more interesting.

THEOREM 3 (my_plus_dynamic_nf). *If M is a variable or tagged application, and M and N are both normal forms, then the term* $\mathsf{my\_plus}\ M\ N$ *has a normal form.*

Proof. For any terms M and N, $\mathsf{my\_plus}\ M\ N$ reduces to an application of M. In general, there is no telling what the result will be, but if M is a variable or tagged application then the application of M becomes tagged, so that the result follows directly.

COROLLARY 1 (my_plus_dynamic_0_nf). *If M is a variable or tagged application, and M is a normal form, then the term* $\mathsf{my\_plus}\ M\ (\mathsf{scott}\ 0)$ *has a normal form.*

Thus, we may aggressively reduce applications of addition during partial evaluation: if the first argument is a Scott numeral then the addition reduces to an iterated successor; if the first argument is headed by a variable then the addition has a normal form.

Given the existence of fixpoints, the ability to define all μ-recursive functions reduces to the challenge of supporting all primitive recursive functions, especially the zero-test and predecessor. Now the booleans are given by

$$\begin{array}{rcl} \mathsf{tt} & = & \lambda x, y.x \\ \mathsf{ff} & = & \lambda x, y.y \end{array}$$

and the zero-test and predecessor are given by

$$\begin{array}{rcl} \mathsf{zero\_test} & = & \lambda n.n, \mathsf{tt}, \lambda x.\mathsf{ff} \\ \mathsf{my\_pred} & = & \lambda n.n, \mathsf{scott}\ 0, \lambda x.x\ . \end{array}$$

All behave as expected. Thus, we have strong evidence (not yet verified in Coq) that closure calculus is Turing complete.

3. $FIESKA$-Calculus

The *operators* (meta-variable O) of $FIESKA$-calculus are

$$O ::= S \mid K \mid I \mid A \mid F \mid E\ .$$

The *terms* of $FIESKA$-calculus are given by

$$p, q, r, s, t, u, v := x \mid O \mid t\ u\ .$$

The terms which are built from the operators by application only, without use of variables, are the *combinations*.

The reduction rules are given in Figure 2. The rules for S, K and I are standard. The rule for A was described in the introduction. The reduction of F branches according to the nature of its first argument. If this is an operator O then the second argument is returned. If this is a *compound* pq then p and q become the arguments of the third argument of F. It is central to the development that redexes can never be compounds, that a compound pq must be head normal. Of course, this cannot be the definition of a compound, since this would create a circularity. Fortunately, it is easy to characterise the compounds syntactically. An application pq is a compound if it is a partially applied operator. That is, every operator has an arity: S, K, I, A, F and E have arities $3, 2, 1, 3, 3$ and 2 respectively. If an operator is applied to at least one argument, but fewer arguments than its arity, then the result is a partial application. For example, Fxt is a compound, but $Fxtu$ is not.

$$\begin{array}{rcll} Sstu & \longrightarrow & su(tu) & \\ Ktu & \longrightarrow & t & \\ Iu & \longrightarrow & u & \\ Astu & \longrightarrow & stu & \\ FOtu & \longrightarrow & t & \\ F(pq)tu & \longrightarrow & upq & (pq \text{ is a compound}) \\ EOO & \longrightarrow & K & \\ EO_1O_2 & \longrightarrow & KI & (O_1 \neq O_2) \\ EO(pq) & \longrightarrow & KI & (pq \text{ is a compound}) \\ E(pq)O & \longrightarrow & KI & (pq \text{ is a compound}) \\ E(p_1q_1)(p_2q_2) & \longrightarrow & Ep_1p_2\,(Eq_1q_2)(KI) & \\ & & & (p_1q_1 \text{ and } p_2q_2 \text{ are compounds}). \end{array}$$

Figure 2. Reduction rules for $FIESKA$-calculus

The sum of the arities is 14 so the two conditional rewriting rules for F can be replaced by 14 unconditional rules. Similarly, the five conditional rewriting rules for E can be replace by $14 * 14 = 196$ unconditional rewriting rules. Of course, this would be tedious, but the point is that the side conditions to the reduction rules are harmless.

THEOREM 4 (Fieska_confluence). *Reduction of $FIESKA$-calculus is confluent.*

Proof. Since the operators and compounds are stable under reduction, it follows that there are no critical pairs.

3.1 Abstraction by Combinatory Analysis

Although we are soon going to give a combination for abstraction, its development is much easier to comprehend if we first introduce the traditional account of abstraction, using λ^*. Given a variable x then define $\lambda^* x.t$ by induction on the structure of t as follows.

$$\begin{array}{rcl} \lambda^* x.x & = & I \\ \lambda^* x.y & = & Ky \quad (y \neq x) \\ \lambda^* x.O & = & KO \\ \lambda^* x.tu & = & S(\lambda^* x.t)(\lambda^* x.u)\ . \end{array}$$

It follows that $(\lambda^* x.t)u$ reduces to $\{u/x\}t$ where $\{u/x\}t$ substitutes u for x in t by meta-theoretic means.

A curious property of $\lambda^* x.t$ is that is is always a normal form, even if t is a redex. For example, if tu is a redex, e.g. IK then $\lambda^* x.IK = S(KI)(KK)$ is a normal form.

In practice, there is much scope for optimizing λ^*. In the Coq implementation of $FIESKA$-calculus, λ^* is optimized in two ways. If x does not occur in t then $\lambda^* x.t$ is defined to be Kt and $\lambda^* x.tx$ is defined to be t. In this manner, some redexes are preserved, but many are still broken.

For example, we could have defined A in terms of S, K and I by

$$\lambda^* x.\lambda^* y.\lambda^* z.xyz\ .$$

This is a normal form, and when applied to s and t reduces to

$$\lambda^* z.stz$$

provided that z is not free in s or t. In turn, $\lambda^* z.stz$ is

$$S(S(\lambda^* z.s)(\lambda^* t))I$$

so that the application of s to t is delayed until some z is provided. Using the optimization above, this can be further simplified to

$$S(S(Ks)(Kt))I$$

since z is not free in neither s nor t. However, some care is required, since it will not do to replace $\lambda^*z.st$ with $K(st)$ since the goal was to break the application of s to t.

3.2 Pattern-Matching

Building on the account of abstraction, we can define pattern-matching functions as *extensions*

$$p \Rightarrow s \mid r \quad = \quad S(p \Rightarrow s)(Kr)$$

with *pattern* p and *body* s and *default function* r. [These are not to be confused with the extension used to build explicit substitutions.] In turn, the *case* $p \Rightarrow s$ is defined by induction on the structure of p. The details can be found in the original paper on SF-calculus (Jay and Given-Wilson 2011), or in the associated Coq proofs. In brief, when an extension $p \Rightarrow s \mid r$ is applied to an argument u then u is compared to p. If they match then the resulting substitution is applied to s. If matching fails then r is applied to u.

3.3 Fixpoint Operators

Now let us consider fixpoint operators. Much as before, we can define

$$\begin{aligned} \omega_2 &= \lambda^*x.\lambda^*f.f(A(A(Axx))f) \\ &= (S(K(SI))(S(KA)(S(KA)(SAI)))) \end{aligned}$$

and

$$Y_2 = A(A\omega_2\omega_2)\ .$$

Thus $Y_2f = A(A\omega_2\omega_2)f$ is head normal but

$$Y_2fx = A(A\omega_2\omega_2)fx \longrightarrow \omega_2\omega_2fx \longrightarrow f(Y_2f)x$$

as required. Similarly, for each k greater than 2 one may define Y_k as a fixpoint for functions f that take $k-1$ arguments.

3.4 Translating Closure Calculus to $FIESKA$-Calculus

In order to translate closure calculus to $FIESKA$-calculus, it is necessary to settle on a choice of term variables for closure calculus. For simplicity, assume that these are the natural numbers. Then the ith variable can be translated by

$$[i] = S^i(S)$$

in $FIESKA$-calculus. Similarly, the translation $[xs]$ of a sequence xs of variables is given by the sequence of the translations, which is defined by using S to represent both nil and cons.

There are $FIESKA$-combinations $\mathsf{tag}, \mathsf{var}, \mathsf{abs}, \mathsf{add}$ and act which can be used to define a translation $[t]$ of tagged λ-terms to $FIESKA$-combinations, as follows:

$$\begin{aligned} [i] &= \mathsf{var}\ [i] \\ [s,t] &= \mathsf{tag}\ [s]\ [t] \\ [t\ u] &= [t]\ [u] \\ [\lambda i, xs[\sigma].t] &= \mathsf{abs}\ [xs]\ (S\ [\sigma]\ [i])\ [t] \\ [I] &= I \\ [i \mapsto u :: \sigma] &= \mathsf{add}\ [\sigma]\ (\mathsf{var}\ (S^iK))\ [u] \\ [[\sigma]\ t] &= \mathsf{act}\ [t]\ [\sigma]\ . \end{aligned}$$

Then we may define lam so that

$$\mathsf{lam}\ v\ t \longrightarrow^* \mathsf{abs}\ S\ (SIv)t\ .$$

THEOREM 5 (lambda_to_FIESKA_preserves_reduction). *If $t \longrightarrow t'$ is a one-step reduction of closure calculus then there is a multi-step reduction $[t] \longrightarrow^* [t']$ of $FIESKA$-calculus.*

Proof. The proof is by induction on the structure of the reduction.

```
(A (A (A (A (A (S (K (S (K (A (S (K (S (K A))) A))))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K (A (S (K (S (K
A))) A))))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S (K (A (S (K (S (K A))) A))))) A))) (S A I)))))) (S (S
(K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (S F (K
I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S (K K) (S (K K) (S (K (S (K K)))
(S (K (S (K K))) (S (K (S (K K))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (S (K S)
(S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S (K
(S (K (S (K S))))))) (S (K (S (K (S (K (S (K (S (K A))))))))) (S (S (K S) (S (K K) (S (K S) (S (K (S (K S))) (S (K (S (K
K))) (S (K (S (K S))) (S (K (S (K K))) (S (K (S (K A))) A)))))))) (K (S (K K) (S (K (S (S (K S) (S (K (S (K S))) S)))) (S
(K K) K))))))))))) (K (K (K (K K))))))))))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S
(K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (S (K (S (S (K S) (S (K K)
(A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K
(A (S (K (S (K A))) A))) (S A I)))))) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S)))
(S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E
A))) (S (K K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S)
(S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S (K K) (S (K K)
(S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E (A (A (S (K
(S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S
(K A))) A))) (S A I)))))))))) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K (S (K K)))
(S (K (S (K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S
(K (S (K (S (K K))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K K))) (S (K S))))) K))))))))))))))))))) (K (S (K (S I)) K))))
(K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K
(K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S I))
K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K
(S (K (S I)) K)))) (S (K K) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K
S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S (K
K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E
(A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K
(A (S (K (S (K A))) A))) (S A I)))))))))) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K K) (S (K
(S (K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (S (K S) (S (K (S (A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K
A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (K (S (S (S
(S F (K I)) (K (S (S (K S) (S (S (S (S F (K I)) (K (S (S (K S) (S (S (S (E S) (K (K (S (S (S F (K I)) (K (S (S (K S) (S (S
(S (S F (K I)) (K (S (S (K S) (S (S (S (E S) (K (K (S (K K) (S (K K) (S (K K) (S (K (S (K K))) (S (K (S (K K))) (S (K (S
(K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S
(K (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (K K) (S (K S) E))) (K (S (K K) K))))))) (S (K K) (S (K K) (S (K (S (S (K
S) (S (K (S I)) K)))) (S (K K) K))))))))))))))))))) (S (K (S I)) K)) (K (K (K (K I)))))) (K (K (K I)))))) (S (K (S I)) K)) (K
(K (K (K I)))))) (K (K (K I)))))) (S (K (S I)) K))))) (S (K (S I)) K)) (K (K (K (K I)))))) (K (K (K I)))))) (S (K (S I)) K))
(K (K (K (K I)))))) (K (K (K I)))))) (S (K (S I)) K)) (K (K K))))))) (S (K K) (A (A (A (S (K (S (K A))) A) (A (S (K (S I))
(S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (S (K
(S (K A))) (S (K (S (K (A (A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I))))
(S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (S (K (S (K (S (K A))))) (S (S (K S) (S (K K) (S (K S)
(S (K K) A)))) (S (K (S (K A))) A)))))))) A))))))))) (S (K K) K)))))))))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I))))))))
(K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K
(S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (S (K K) (S (S (K S) (S (S (K S) (S (K (S
(S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S
(K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S (K K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S
(K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S
(S (K S) (S (K (S (E A))) (S (K K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S
(K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S
(K K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S
(S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E A))) (S (K K) (S (K K) (S (K (S
(K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S
(K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S
(K (S (K (S (K (S (K K))))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S (K (S (K (S (K K))))))) (S (S (K S) (S (K (S (K S))) (S
(K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K (S (S F (K I)))))))))) (S (K (S (K (S (K (S
(K K))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K
(S (K S))))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (S (K S) (S (K (S
(K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K (S (S F (K I)))))))))) (S (K (S
(K (S (K (S (K K))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K
(S (K (S (K (S (K S))))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (S
(K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K (S (E S)))))))))
(S (K (S (K (S (K (S (K K))))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S (K (S (K (S (K (S (K K))))))))) (S (K (S (K (S
(K (S (K (S (K K))))))))) (S (K (S (K (S (K (S (K (S (K K))))))))) (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K
(S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K
(S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K
(S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))))) (S (K (S (K (S (K (S
(K (S (K (S (K (S (K K))))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))))) (S (S (K S) (S (K (S (K S)))
(S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (K (S (K (S (K (S (K (S (K S))))))))) (S (K (S (K (S (K (S (K
(S (K (S (K S))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K K))))))))))) (S (K (S (K (S (K (S (K (S (K (S (K S)))))))))))
(S (K (S (K (S (K (S (K (S (K (S (K (S (K A))))))))))))) (S (K (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K
(S (K (S (K K))))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K K)))))
(S (K (S (K (S (K A))))) (S (K (S (K A))) A)))))))))))) (S (K K) (S (K K) (S (K K) (S (S (K S) (S (K (S (K S))) (S (K (S
(K K))) (S (K (S (K S))) (S (K (S (K S)))))))) (K (K K))))))))))))))) (K (K (K (K (K (K K)))))))))))))))))))))))))) (K (K
(K (K (S (K (S I)) K)))))))))) (K (K (K (K (K (K (K (K I))))))))))))) (K (K (K (K (K (K (K I))))))))))))) (K (K (K (K (S
(K (S I)) K))))))))) (K (K (K (K (K (K (K (K I))))))))))))) (K (K (K (K (K (K (K I))))))))))))) (K (K (K (K (S (K (S I))
K))))))))))))))))))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K
(K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K
I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S I)) K))))
(K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K
(S I)) K))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K
I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (S (K K) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S
(K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S
(S (K S) (S (K (S (E S))) (S (K K) (S (K K) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S
(S (K S) (S (S (K S) (S (K (S (S F (K I)))) (S (K K) (S (S (K S) (S (K (S (K S))) (S (S (K S) (S (S (K S) (S (K (S (E S)))
(S (K K) (S (K K) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K K))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K)))))
(S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K K))))) (S (K (S (K (S (K (S (K K))))))) (S (K (S (K
(S (K (S (K K))))))) (S (K (S (K (S (K (S (K K))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K (S (K S))))) (S (K (S (K (S
(K K))))) (S (K (S (K (S (K S))))) (S (K (S (K (S (K (S (K S))))))) (S (S (K S) (S (K (S (K S))) (S (K (S (K K))) (S (K (S
(K S))) (S (K (S (K S)))))))) (K (K K))))))))) (S (K K) K)))))))))))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K
(K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K))))))) (K (S (K (S
I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I)))))))) (K (S (K (S I)) K)))) (K (K (K (K (K I)))))))) (K (K (K (K I))))))))
(K (S (K (S I)) K)))) (K (K K)))))))))))))) (S (K K) S))))))) S) (S I (A (A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A)
(S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (S (K (S (K
A))) (S (K (S (K (A (A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K
(S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (S (K (S (K (S (K A))))) (S (S (K S) (S (K K) (S (K S) (S (K
K) A)))) (S (K (S (K A))) A)))))))) A))) S))) (A (A (A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K
A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))))) (S (K (S (K A))) (S (K (S (K (A (A
(A (S (K (S (K A))) A) (A (S (K (S I)) (S (K A) (S (K (A (S (K (S (K A))) A))) (S A I)))) (S (K (S I)) (S (K A) (S (K (A
(S (K (S (K A))) A))) (S A I)))))) (S (K (S (K (S (K A))))) (S (S (K S) (S (K K) (S (K S) (S (K K) A)))) (S (K (S (K A)))
A)))))))) A))) S)).
```

Figure 3. The combination for the identity abstraction

3.5 Reducing the Overheads of Abstraction

There are many ways to represent closures as combinations. The most direct is to introduce operators for the term forms of closure calculus, so that the representation does not increase term size at all. The disadvantage of this approach is that it does not provide opportunities for optimization. At the other extreme, we could translate to SF-calculus. Now the combination for abstraction contains within it all of the pattern-matching necessary to perform substitutions, including the representation of E (at about two thousand operators) and of A (about twenty operators). The resulting combination requires many thousands of operators, probably more one hundred thousand! An order of magnitude reduction in size is achieved by targetting $FIESKA$-calculus, where the identity function, expressed as an abstraction, requires 3383 operators, as displayed in Figure 3. This is a lot less than one hundred thousand, but a lot more than one.

So it is an interesting diversion to eliminate the overheads that are introduced by the representation process. Specifically, our first goal is to optimize the identity abstraction to I. Of course, this optimization changes the internal structure of the term, in that, for example, the value of the size function will change. So, to put it more precisely, we are looking for optimizations that preserve the extensional behaviour of the program, by modifying its internal structure.

It turns out that a single optimization rule will suffice. Add to $FIESKA$-calculus the following variant of the η-rule. rule

$$(\eta)\ \frac{M\ x \longrightarrow^* N}{M \longrightarrow^* \lambda^* x.N}\ x \text{ not free in } M.$$

Of course, this breaks the confluence of reduction in $FIESKA$-calculus but should have no impact on extensional behaviour, e.g. the behaviour of the translations of terms from closure calculus. In particular, we have the following theorems.

THEOREM 6 (identity_abstraction_optimized). *Given the η-rule above, we have*

$$\llbracket \lambda x.x \rrbracket \longrightarrow^* I\ .$$

Proof. Routine calculation.

THEOREM 7 (first_projection_abstraction_optimized). *Given the η-rule above, we have*

$$\llbracket \lambda x, y.x \rrbracket \longrightarrow^* K\ .$$

Proof. Routine calculation.

THEOREM 8 (second_projection_abstraction_optimized). *Given the η-rule above, we have, for all M,*

$$\llbracket (\lambda x, y.y)M \rrbracket \longrightarrow^* I\ .$$

Proof. Routine calculation.

These results suggest that radical optimization of closures is possible in $FIESKA$-calculus. For example, $(\lambda x, y.y)M$ above reduces to $\lambda y[x \mapsto M].y$ so the effect of the optimization is to recognise that the binding of x to M is not required, so that the closure can be replaced by a combinator.

4. Program Manipulation

The general approach to program manipulation is illustrated by the following example, where terms of the form $\mathsf{my\_plus}\ x\ (\mathsf{scott}\ 0)$ are replaced by x. Actually, this is a little more delicate than it may appear, for reasons that will emerge below.

The optimization is done by means of general machinery, namely the $\mathsf{update}\{p, s\}$ function that is defined by

$$Y_2\ (\lambda^* u.\ p \Rightarrow s \mid (\lambda^* x.F\ x\ x\ (\lambda^* x_1.\lambda^* x_2.u\ x_1\ (u\ x_2))))\ .$$

It replaces instances of p by the corresponding instances of s, and otherwise recurses through the structure of (the normal form of) the argument.

The first delicate point is that the pattern p should be a normal form. Now $\mathsf{plus}\ x\ (\mathsf{scott}\ 0)$ is not a normal form, though it has one. If we abuse notation and designate this normal form by $x + 0$ then we can write the optimization as $\mathsf{update}\{x + 0, x\}$. The second delicate point is that update builds terms of $FIESKA$-calculus (since the pattern-matching uses F) but the variable x in $x + 0$ lives in closure calculus (since it must be translated to a var). So, to be more precise, we require

$$\mathsf{my\_plus\_zero\_r} = \mathsf{update}\{\llbracket x + \mathsf{scott}\ 0)\rrbracket, \llbracket \mathsf{scott}\ 0\rrbracket\}\ .$$

The third point is that matching may fail for some terms of the form $\llbracket \mathsf{my\_plus}\ t\ (\mathsf{scott}\ 0)\rrbracket$. Recall that if t is $\mathsf{scott}\ m$ then $\llbracket \mathsf{my\_plus}\ t\ (\mathsf{scott}\ 0)\rrbracket$ reduces to $\llbracket \mathsf{scott}\ m\rrbracket$ which will not match the pattern. Rather, matching will succeed if t is a variable, or is tagged. For example, if t is $\mathsf{my\_plus}\ y\ u$ for some variable y then t reduces to $y + u$ which is a tagged term, and now matching will succeed and replace $(y + u) + 0$ with $y + u$.

THEOREM 9 (my_plus_zero_r_basic). *For all normal forms t that are either variables or tagged applications, we have*

$$\mathsf{my\_plus\_zero\_r}\ \llbracket \mathsf{my\_plus}\ t\ (\mathsf{scott}\ 0)\rrbracket \longrightarrow^* \llbracket t \rrbracket\ .$$

We can represent the pair $\mathsf{pair}\{t, u\}$ by $\lambda f.f, t, u$. Now we also have

THEOREM 10 (my_plus_zero_r_basic4).

$$\mathsf{my\_plus\_zero\_r}\ \llbracket \mathsf{pair}\{\mathsf{my\_plus}\ x\ (\mathsf{scott}\ 0), \mathsf{my\_plus}\ y\ (\mathsf{scott}\ 0)\}\rrbracket$$

reduces to $\llbracket \mathsf{pair}\{x, y\}\rrbracket$ *for all variables x and y.*

In practical terms this is not very much. A single, almost trivial, optimization, with no machinery for combining optimizations into powerful strategies. More immediately, the construction of the update uses meta-theory, in that the extension construction is not within the calculus. However, there do not appear to be any conceptual barriers to a more practical development. The pattern-matching machinery can be used to build more complex optimizations. Also, by representing extensions within the calculus, it can support dynamic patterns that can be computed before matching commences. For now, we have established that optimization can be performed within the calculus, without any need for quoting.

5. Related Work

According to Jones, Gomard and Sestoft (Jones et al. 1993), the three main partial evaluation techniques are symbolic computation, unfolding function calls and program point specialization, of which our main contribution is to unfolding. The key challenge is to avoid unfolding recursive functions that depend on dynamic arguments. Traditionally, this relies on *binding time analysis* to determine which expressions depend upon dynamic arguments. In closure calculus, partial evaluation proceeds without any separate binding time analysis. This is possible because tags supply the required information. For example, consider the expression $x\ t\ u$. If x is a dynamic variable then the sub-expressions $x\ t$ and $x\ t\ u$ are also dynamic. In closure calculus, this is captured by the reductions

$$x\ t\ u \longrightarrow x, t\ u \longrightarrow x, t, u\ .$$

That is, any residual tagged terms can be considered dynamic. To put it another way, tagging was introduced to control the application of explicit substitutions, but is then available as an aid to analysis.

Traditionally, partial evaluation must consider programs in two ways, as both functions and data. In the λ-calculus tradition of self-interpretation (Kleene 1936; Reynolds 1972; Barendregt 1991; Mogensen 1992, 2000; Berarducci and Böhm 1993; Brown and Palsberg 2017), one begins with a λ-term (i.e. a function) and then quotes it, to produce a data structure, e.g. a Gödel number or an abstract syntax tree, perhaps expressed as another λ-term. Alternatively, one can consider a program to be a data structure, devoid of meaning, which acquires functionality through its interpretation with respect to some particular programming language. (Jones et al. 1993). These two-level approaches introduce overheads which can now be avoided. In particular, quotation of programs is now directly definable within the programming language itself. That is, programs are closed normal forms, whose internal structure is revealed by factorisation.

Of course, this approach to partial evaluation is in its earliest stage of development. In particular, there is not yet any partial evaluator as such, much less one that can be self-applied, to realize the Futamura projections (Futamura 1999).

That optimizations can be expressed as rewrite rules is fairly obvious. Given that programs have been quoted, to produce data structures, then it is straightforward to update their structure by using tree traversals, e.g. (Berezun and Jones 2017). The novelty of the approach adopted here is that the resulting optimization functions are definable in the source language, without any need for quotation.

6. Conclusions

A fundamental challenge for partial evaluation has always been to avoid useless evaluation, especially, to avoid unfolding recursions without limit. This is particularly important when the full evaluation does terminate, as when evaluating primitive recursive functions.

The first contribution of this paper is to provide new representations of recursive programs as fixpoints, in which reduction of primitive recursive expressions, such as addition of natural numbers, is guaranteed to terminate no matter what evaluation strategy is used. In these circumstances, there is no useless evaluation, and partial evaluation can be completely aggressive.

The basic idea is to support fixpoint functions whose recursion emerges only when given sufficient arguments. For example, the recursion of a program p of the form $Y_2 f$ emerges only when it is applied to some argument x. This application reduces to $f(Y_2 f)x$ or fpx in which the new copy of p cannot recurse unless reduction of fpx supplies p with an argument.

Such fixpoints have been given for three different rewriting systems, namely combinatory logic, closure calculus, and $FIESKA$-calculus. For example, in $FIESKA$-calculus, Y_2 is given by

$$\begin{aligned} Y_2 = A(&A(S(K(SI))(S(KA)(S(KA)(SAI)))) \\ &(S(K(SI))(S(KA)(S(KA)(SAI))))) \ . \end{aligned}$$

In every calculus, they rely on the existence of applications (perhaps tagged applications) which are closed normal forms. As is well known, these do not occur in pure λ-calculus.

In order to support such fixpoints in a λ-calculus, it has been necessary to modify the pure λ-calculus. The main change has been to allow abstraction to bind a sequence of variables, so that substitution can be delayed until more arguments are given. However, closure calculus goes much further than this. By tagging applications that are headed by variables, reduction of explicit substitutions can be controlled in such a manner that it is is no longer necessary to substitute into the body of abstractions, which then behave like the closures which are familiar from implementations of functional programming languages. In this manner, the theory becomes both simpler (no critical pairs, no renaming of variables) and closer to practice.

The second contribution of this paper is to program manipulation. Traditionally, partial evaluation and program analysis for rewriting systems have acted on the syntax trees of programs, obtained by quotation. However, this is no longer necessary. As noted above, programs now have normal forms so there is no need to "freeze" them by quotation. However, this exposes the computational limitations of λ-calculus and combinatory logic, as these extensional calculi are unable to query the internal structure of their terms without outside help. Instead, it is necessary to adopt an intensional rewriting system, such as SF-calculus or $FIESKA$-calculus. Now abstractions can be represented as combinations whose internal structure can be explored by factorisation, using F. Since the representations in SF-calculus are so large, we work with $FIESKA$-calculus instead. Representations are still quite large, but extensional optimization can be used to simplify, or eliminate closures so that the representations become quite small.

To illustrate the approach to program manipulation, we developed general machinery for converting an optimizing rule

$$p \Rightarrow s$$

into an optimizer

$$\mathsf{update}\{p, s\} \ .$$

For example, given the rewriting rule

$$\mathsf{my\_plus}\ x\ 0 \Rightarrow x$$

then we can update using the pattern $x + 0$ which is the normal form of $\mathsf{my\_plus}\ x\ 0$. When applied to a program, the update will traverse the program structure looking for sub-terms that match the pattern. At the time of writing, the conversion of the optimizing rule to the update term must be done by hand, but further automation can be expected. Similarly, the size of the representations makes optimization a time-consuming process, though this too is open to improvement.

These contributions suggest that, in future, partial evaluation and program manipulation will be conducted in the source programming language, without recourse to quotation or other meta-programming techniques, so that program analysis will become both simpler and more powerful.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy. Explicit substitutions. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, pages 31–46. ACM, 1990. URL `citeseer.nj.nec.com/abadi91explicit.html`.

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

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

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.

D. Berezun and N. D. Jones. Compiling untyped lambda calculus to lower-level code by game semantics and partial evaluation (invited paper). In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM 2017, pages 1–11, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4721-1. doi: 10.1145/3018882.3020004. URL `http://doi.acm.org/10.1145/3018882.3020004`.

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.

H. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Combinatory Logic*. North-Holland Publishing Company, 1958.

Y. Futamura. Partial evaluation of computation process, revisited. *Higher-Order and Symbolic Computation*, 12(4):377–380, Dec 1999. ISSN 1573-0557. doi: 10.1023/A:1010043619517. URL `https://doi.org/10.1023/A:1010043619517`.

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

B. Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.

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

B. Jay. Self-quotation in a typed, intensional lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 2017a. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII), to appear.

B. Jay. Beyond lambda-calculus: Intensional computation. `http://lambdajam.yowconference.com.au/slides /yowlambdajam2017/Jay-BeyondLambdaCalculus.pdf`, May 2017b. Presented at Yow LambdaJam, Sydney.

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

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

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.

N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall International, 1993.

D. Kesner. The theory of calculi with explicit substitutions revisited. In *International Workshop on Computer Science Logic*, pages 238–252. Springer, 2007.

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

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.

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

G. Pottinger. *A Tour of the Multivariate Lambda Calculus*, pages 209–229. Springer Netherlands, 1990. ISBN 978-94-009-0681-5. URL `https://doi.org/10.1007/978-94-009-0681-51-14`.

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.

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