Deconstructing Lambda-Calculus

Barry Jay

Centre for Artificial Intelligence
University of Technology Sydney
Barry.Jay@uts.edu.au

Abstract. Pure lambda-calculus supports functions that juggle their arguments but not those that query their internal structure. By contrast, SF-calculus can represent all such queries of closed normal forms but has given a poor account of lambda-abstraction, by either failing to preserve reduction or by adding significant additional machinery. This paper overcomes both of these limitations by representing the formation of variables and abstractions by the SF-combinations var and lam.
More precisely, there is a reduction-preserving translation to SF-calculus from a new variant of lambda-calculus, the *closure calculus* that is of independent interest. The latter is a confluent rewriting system that: is Turing complete; supports explicit substitutions without introducing any critical pairs; is closer to implementation practice; and has the surprising property that there is no need for variable renaming.

1 Introduction

Pure λ-calculus (or $\lambda\beta$-calculus) supports *extensional* functions, those that are able to apply their arguments to each other, but cannot directly query their internal structure. This is a significant limit on expressive power, since queries and other *intensional* functions are ubiquitous, both in practice and in theory. Examples include the standard database queries of, say, SQL, the pattern-matching functions of functional programming languages, and even the substitution function of λ-calculus itself. To represent these examples within λ-calculus requires additional machinery, either to specialize the queries to particular types, or to encode λ-terms as syntax trees.

Over the last decade or two, have developed calculi that improve upon λ-calculus with steadily increasing support for intensionality [11, 4, 9, 5, 8, 10, 6, 7]. Of these, the best mix of expressive power and simplicity is found in SF-calculus [8]. The operator S is taken from combinatory logic and the operator F is used to factorize combinations into their components. This is enough to support all intensional functions that are computable by a Turing machine, such as the equality of closed normal forms, generic queries that search and replace, and even program analyses that identify dead code, etc. In this sense, SF-calculus is *intensionally complete*.

The chief concern about SF-calculus has been the manner in which it represents abstraction. In the original paper, λ-abstraction was represented just as

in SKI-calculus, as follows. Create a language of terms t by adding a countable class of identifiers x, y, z to the combinations. Then define the abstraction $\lambda^* x.t$ just as in SKI-calculus, with $\lambda^* x.x$ defined to be the identity combinator I, etc. The main weakness of this approach is that, for every term t, the abstraction $\lambda^* x.t$ is a normal form, so that abstraction breaks every redex within the abstraction body t! It follows that abstraction does not preserve reduction; the ξ-rule [1] is broken. In practice, this implies that abstraction cannot be performed freely at any point during computation, but must be done statically, or controlled by staging [2]. This inability to support the ξ-rule greatly limits the utility of combinators as a source of optimization.

Recent work adds λ-abstraction to SF-calculus to get λSF-calculus [6, 7]. This contains both pure λ-calculus and SF-calculus as sub-systems, then adds rules for showing how to factorize abstractions. Unfortunately, the rules for deciding when abstractions are compounds are given by some complicated metatheory, so that the calculus is not very axiomatic. That is, the reduction rules are too complicated to appreciate them at a glance, much less to approve of them.

Looking back, the same criticism can be leveled at λ-calculus itself. The terms of pure λ-calculus are given by the BNF

$$s, t, u ::= x \mid t\ u \mid \lambda x.t$$

where x is an identifier whose value in t may vary through substitution $\{u/x\}$ of u for the identifier x, as introduced by the β-reduction rule

$$(\lambda x.t)u \longrightarrow \{u/x\}t\ .$$

Of course, this substitution is an intensional function, that queries the internal structure of t to determine if it is an identifier y, an application $t\ u$ or abstraction $\lambda y.s$. In the latter case it is necessary to avoid name capture: if y is x then it must be renamed (using α-conversion) by some "fresh" variable. Enough students struggle with these ideas that we can be sure that they are not axiomatic. It further illustrates the weakness of λ-calculus as a theory of functions, since, without extra machinery, it cannot even represent the functions required for its own operation!

Some of this burden can be transferred to the rewriting rules by introducing *explicit substitutions* σ so that the meta-computation $\{u/x\}t$ above can be replaced by the new term form $[x \mapsto u :: I]t$ where $\sigma = x \mapsto u :: I$ is an explicit substitution. There is a large family of such calculi [12] but most of them are able to both support equational reasoning (be confluent [14]) and avoid variable renaming (no α-conversion). Since we emphasise program analysis based on equational reasoning, we will restrict attention to the confluent calculi of explicit substitutions, and name them, collectively, as the $\lambda\sigma$-calculi.

This paper overcomes these limitations of λ-calculus by re-examining it from the viewpoint of intensional computation. This deconstruction leads to a new variant of explicit substitution calculus, the *closure calculus* or $\lambda\tau$-calculus, which is confluent and does not perform renaming. That done, it is routine to represent

the various term forms as applications of operators in *abstraction calculus*. For example, a λ-abstraction $\lambda x.t$ is now given by an application

$$\mathsf{lam}\ (\mathsf{Var}\ x)\ t$$

where $\mathsf{Var}\ x$ is a variable whose name is the term x, and lam is built using the abstraction operator Abs. Now the ξ-rule follows by construction. Finally, all of the operators of abstraction calculus can be represented as combinations, such as var and abs in SF-calculus, so that there is a reduction-preserving translation of $\lambda\tau$-calculus to SF-calculus. Thus, SF-calculus has all of the expressive power of SKI-calculus, and of λ-abstraction, and, in addition, supports the full range of intensional functions.

The measure of its simplicity is given by the size of the combination abs, which requires many thousands of operators. This is because it contains within it the machinery for performing substitutions, which in turn requires machinery for deciding when variables are equal, i.e. when $\mathsf{var}\ x$ and $\mathsf{var}\ y$ are equal. By the way, although it is tempting to call abs a *combinator*, Curry and Feys reserve this term for combinations of operators that correspond to λ-terms, so, following their lead, we call terms of SF-calculus *combinations*.

	N	ξ	σ	no α	F	axiomatic
SKI	✓	×	×	×	×	✓
$\lambda\beta$	✓	✓	×	×	×	×
$\lambda\sigma$	✓	✓	✓	?	×	×
$\lambda\tau$	✓	✓	✓	✓	×	×
λSF	✓	✓	✓	×	✓	×
SF	✓	✓	✓	✓	✓	✓

Fig. 1. Properties of Some Confluent Rewriting Systems

The differences between the calculi mentioned above are summarized in Figure 1. Each row describes a calculus: each column considers a property, as follows. The property **N** holds if the calculus is Turing complete. Property ξ holds if the ξ-rule holds. Property σ holds if substitution is represented explicitly in the term syntax. Property "no α" holds if there is no variable renaming, if names are fixed. Property F holds if the calculus supports factorisation. Property "axiomatic" holds if the reduction rules are axiomatic, in the sense that there are no side conditions of any kind. The table shows a steady improvement, from SKI-calculus to SF-calculus, which is the only example to have all these properties.

Most of the confluent calculi of explicit substitutions require variable renaming. An exception [3] achieves confluence by requiring that subsitutions be closed before substituting into abstractions. By contrast, we never substitute into abstraction bodies.

In terms of Figure 1, there are two main results in this paper. First, there is another confluent, Turing complete, λ-calculus, the closure calculus, that does

not require any variable renaming. Second, and mor important, there is an intensionally complete, axiomatic calculus, SF-calculus, which supports the ξ-rule for λ-abstraction.

The structure of the paper is as follows. Section 1 is the introduction. Section 2 introduces the closure calculus. Section 3 introduces the abstraction calculus. Section 4 translates the closure calculus to SF-calculus. Section 5 considers future work. Section 6 draws conclusions.

All proofs of named theorems have been verified in Coq, and are available on request.

2 Closure Calculus

Closure calculus is a variant of λ-calculus whose construction is motivated by the properties in Figure 1. It is to be a λ-calculus that is confluent, Turing complete, supports explicit substitutions and fixes variable names. The first subsection considers how to achieve these properties. The second subsection presents the formal development of the calculus, but may be read first, if preferred. The third subsection shows that closure calculus is Turing complete.

2.1 Motivation

In pure λ-calculus, substitution into a λ-abstraction may require variable renaming to avoid scoping problems. That is, when substituting u for x in $\lambda y.t$ it may be necessary to rename y by some fresh variable z that is neither x nor appears free in u. We can avoid this by equipping each abstraction with its own substitution σ so that abstractions are generalized to closures such as

$$\lambda\, ys\, [\sigma].t\ .$$

where ys is a (non-empty) list of bound variables.

Now consider substitution of u for x in this closure or, more generally, the application of some other substitution σ_2 to this closure. The natural solution is to produce $\lambda ys.[\sigma_2 \circ \sigma].t$ where $\sigma_2 \circ \sigma$ is a composition of substitutions that await their chance to act on t.

The difficulty with this approach is it cannot support the representation of arithmetic. This is because numerals are represented by abstractions with empty substitutions which, when composed with σ_2 produce non-empty substitutions, that are not numerals.

Instead, the application of σ_2 above will reduce to

$$\lambda ys.[\sigma_2\sigma].t$$

where the substitution σ_2 is applied to the substitution σ only, and never to t. To support this, it is convenient for the substitutions to be terms, in the same syntactic class as the closures. In particular, if σ is the empty substitution, represented by the identity operator I, then $\sigma_2 I$ will reduce to I so that numerals

will be stable under substitution. All other substitutions are given by *extensions* of the form $x \mapsto u :: \sigma$. Application of σ_2 to this reduces to $x \mapsto \sigma_2 u :: \sigma_2\sigma$.

Of course, this solution seems peculiar from a traditional standpoint. For example, the substitution of u for x in $\lambda y.x$ will have no effect! That is, there are unbound variables, such as x above, which cannot be acted upon by substitution. This certainly messes with our intuitions about free variables etc. However, it turns out that such concepts are not required for the development.

Now consider the application of a substitution σ to an application $t\ u$. The natural solution is to apply σ to each of t and u. However, this will clash with our earlier decisions. For example, suppose that t is $\lambda x.x\ y$. Then $\sigma(t\ u)$ reduces to both $\sigma(u\ y)$ and to $(\lambda x.x\ y)(\sigma u)$. In turn these reduce to $\sigma\ u\ (\sigma\ y)$ and $\sigma\ u\ y$. Thus, σ may or may not be applied to y.

The solution is to drop this reduction rule, i.e. to reduce applications such as $(\lambda x.x\ y)\ u$ before applying substitutions. Informally, we will require that abstractions act on head normal forms only. This works well for closed terms, but there is a problem when substitutions are applied to terms of the form

$$x\ t_1\ \ldots t_k\ .$$

Such terms are indeed head normal in pure λ-calculus, but we do not want to introduce a family of reduction rules, indexed by the number k of arguments that a variable takes.

The solution is to introduce a new term form, the *tagged application*

$$t, u$$

and the reduction rule

$$\sigma(t, u) \longrightarrow (\sigma t)(\sigma u)\ .$$

Then $x\ t_1\ \ldots\ t_k$ is handled by adding just two more reduction rules

$$\begin{aligned} x\ u &\longrightarrow x, u \\ (s, t)\ u &\longrightarrow ((s, t), u)\ . \end{aligned}$$

Thus $\sigma(x\ t_1\ \ldots\ t_k)$ reduces to $(\sigma x)(\sigma t_1)\ldots(\sigma t_k)$ in $k + k$ steps.

In this manner, the substitution machinery is both closer to implementation practice and simpler in theory, since there are no critical pairs. Also, using $k + k$ reductions within the calculus seedms liek an improvement over using k steps of meta-theory.

2.2 Formalities

Suppose given a countable class of *names* $x, y, z, \ldots$ whose equality is decidable. Let xs denote a comma-separated list of names, with empty list given by nil. The *terms* of the *closure calculus* or $\lambda\tau$-calculus are given by the BNF

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

consisting of *variables*, *tagged applications* s, t and *applications* $t\ u$, *closures* $\lambda x, xs[\sigma].t$, the *identity operator* I and *extensions* $x \mapsto u :: \sigma$. Like application, tagged application associates to the left, but binds less tightly than ordinary applications. As usual, abstraction binds as far to the right as possible. The interpretation of the abstraction $\lambda x.t$ of pure λ-calculus is given by the closure $\lambda x, \mathsf{nil}[I].t$ with the singleton sequence x, nil of bindings and I playing the role of identity substitution.

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

Fig. 2. Reduction rules of closure calculus

The *reduction rules* of the calculus are given in Figure 2. They can be understood as follows. Applications of variables may be tagged to indicate head normality. Applications of tagged terms may be tagged too. Thus, $x\ u_1\ u_2\ \ldots\ u_k$ reduces to $x, u_1, u_2, \ldots, u_k$ in k steps. When an abstraction is applied to an argument then the substitution is extended with a new case. If there are no bindings remaining then the resulting substitution is applied to the abstraction body. Otherwise, the result is the obvious abstraction. Note that this differs from the *multi-variate* λ-calculus [13], in that the latter delays β-reduction until all bound variables in the list are given values. I is the identity operator, used to represent the empty substitution. The rules for applying an extension $x \mapsto u :: \sigma$ are intensional, in that they must query the structure of the argument. If it is a variable y then there are two cases. If y is x then return u else apply σ to y. If it is a tagged application (s,t) then apply the extension to each of s and t to get s' and t' and apply s' to t' without tagging. If it is a closure $\lambda y, ys[\sigma_2].t$ then apply the extension to σ_2 only, leaving the body t of the closure untouched. This conforms to the usual treatment of closures on the presumption that all free variables in the body are in the domain of the substitution. If the argument is I then return

σI. If it is an extension $y \mapsto v :: \sigma_2$ then apply the outer extension to v and to σ_2.

Since extensions never act on the body of a closure, there is no possibility of scope violations, and so no need to ever rename variables.

Theorem 1 (tagged_confluence). *Reduction is confluent.*

Proof. There are no critical pairs.

The normal forms are given by the following BNF.

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

Theorem 2 (tagged_progress). *Every term t is either a normal form, or reduces.*

Proof. A straightforward induction on the structure of the term suffices.

Closure calculus is **not** directly comparable to the pure λ-calculus. On the one hand, its substitutions cannot act on redexes. On the other hand, it supports more structure, such as explicit substitutions, that is not easily represented in pure λ-calculus. However, these differences are no great concern. First, closure calculus has the core properties of interest. That is, it is a confluent rewriting system, and the translation from pure λ-calculus preserves normal forms. Further, in its use of closures and control over reduction, it is much closer to the implementation of functional programming languages than pure λ-calculus. The remaining significant issue is its support for arithmetic.

2.3 Arithmetic

The closure calculus supports arithmetic, including all of the μ-recursive functions. The most important point is that the usual accounts of fixpoint hold. We can define

$$\begin{aligned} \omega &= \lambda x, f[I].f, (x, x, f) \\ Y &= \lambda f[x \mapsto \omega :: I].f, (x, x, f) \ . \end{aligned}$$

In the traditional accounts, the fixpoint combinator is defined to be $\omega\omega$ which does not have a normal form. Here, however, $\omega\omega$ reduces to the normal form Y. Using similar techniques, one can represent all recursive functions as normal forms, with the possibility of non-termination arising only when the functions are applied to enough arguments. This approach was first developed in λSF-calculus [6].

Lemma 1 (fixpoint_property). *For all terms f of closure calculus, Yf reduces to $f(Yf)$.*

Although support for fixpoints is the usual focus of interest, there remain some subtleties in the representation of the primitive recursive functions, arising from the need to eliminate substitutions. This proves to be problematic for the Church numerals but not for the Scott numerals. Let us consider them in turn.

The Church numerals of pure λ-calculus represent zero and successor by $\lambda f.\lambda x.x$ and $\lambda n.\lambda f.\lambda x.f(nfx)$ respectively. Then the predecessor can be represented by

$$\mathsf{pred} = \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u) \; .$$

Then the predecessor of zero reduces to

$$\lambda f.\lambda x.(\lambda f.\lambda x.x)(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda u.u)$$

and thence to zero again. However, the initial β-reduction substitutes for n within the abstraction by f and x which is not possible in closure calculus.

The Scott numerals of pure λ-calculus represent zero and successor by $\mathsf{zero} = \lambda x.\lambda y.x$ and $\mathsf{succ} = \lambda n.\lambda x.\lambda y.yn$ respectively. Then the predecessor can be represented by

$$\mathsf{pred} = \lambda n.n \; \mathsf{zero} \; (\lambda x.x)$$

so that application to some numeral n is able to apply n immediately, without waiting for other arguments. In this manner, the zero test, the predecessor, and other primitive recursive functions, such as addition, and minimization can all be defined and shown to behave correctly.

3 Abstraction Calculus

Abstraction calculus is built to represent the syntax of the closure calculus, both its variables and terms, as combinations of operators. The terms of the *abstraction calculus* are given by

$$M, N, P, Q ::= O \mid MN$$

where MN is the application of M to N as usual, and the *operators* (meta-variable O) are given by

$$S \mid K \mid I \mid E \mid \mathsf{Tag} \mid \mathsf{Var} \mid \mathsf{Ext} \mid \mathsf{Abs} \; .$$

These operators consist of the traditional combinators S, K and I of combinatory logic, plus operators E for equality, Tag for tagging, Var for variables, Ext for extensions, and Abs for abstractions. Every operator has an *arity* given by $3, 2, 1, 2, 3, 2, 4$ and 5 respectively. The *compounds* are the partial applications $P = MN$ of an operator, i.e. where the operator is applied to fewer arguments than its arity. The terms M and N are then the *left-* and *right-hand components* of P. The *lambda forms* are those terms of the form

$$\mathsf{Tag} \; Q \; R, \; \mathsf{Var} \; M, \; \mathsf{Ext} \; MNP, \; \mathsf{Abs} \; MNP$$

which will be used to represent terms of λ-calculus. All other compounds are *true compounds*. The *factorable forms* are the operators and the compounds. The number of possible terms form that are factorable is given by the sum of the arities, which is 22.

The reduction rules are given in Figure 3. As written, there are 18 reduction rules, but when the side-conditions are eliminated, there are 22 rules for Ext and for Abs and 22 * 22 rules for E making a a total of $5+22+22+484 = 533$ rules!

$$
\begin{aligned}
SMNP &\longrightarrow MP(NP)\\
KMN &\longrightarrow M\\
IM &\longrightarrow M\\
EOO &\longrightarrow K\\
EON &\longrightarrow KI \quad (N \neq O \text{ is factorable})\\
EMO &\longrightarrow KI \quad (M \text{ is compound})\\
E(P_1Q_1)(P_2Q_2) &\longrightarrow EP_1P_2(EQ_1Q_2)(KI)\\
&\qquad (P_1Q_1 \text{ and } P_2Q_2 \text{ are compounds})\\
\mathsf{Tag}\ MNP &\longrightarrow \mathsf{Tag}(\mathsf{Tag}\ M\ N)P\\
\mathsf{Var}\ MN &\longrightarrow \mathsf{Tag}(\mathsf{Var}\ M)N\\
\mathsf{Ext}\ MNP(\mathsf{Tag}\ Q\ R) &\longrightarrow \mathsf{Ext}\ MNPQ(\mathsf{Ext}MNPR)\\
\mathsf{Ext}\ MNP(\mathsf{Var}\ Q) &\longrightarrow EMQN(P(\mathsf{Var}\ Q))\\
\mathsf{Ext}\ MNP(\mathsf{Ext}\ M_2N_2P_2) &\longrightarrow \mathsf{Ext}\ M_2(\mathsf{Ext}\ MNPN_2)(\mathsf{Ext}\ MNPP_2)\\
\mathsf{Ext}\ MNP(\mathsf{Abs}\ M_2N_2P_2) &\longrightarrow (\mathsf{Abs}\ M_2(\mathsf{Ext}\ MNPN_2)P_2)\\
\mathsf{Ext}\ MNPO &\longrightarrow PO\\
\mathsf{Ext}\ MNP(QR) &\longrightarrow \mathsf{Ext}\ MNPQ(\mathsf{Ext}MNPR)\\
&\qquad (QR \text{ is a true compound})\\
\mathsf{Abs}\ OMNPQ &\longrightarrow \mathsf{Ext}\ MQNP\\
\mathsf{Abs}\ (OR)MNPQ &\longrightarrow \mathsf{Ext}\ MQNP \qquad (OM \text{ compound.})\\
\mathsf{Abs}\ (R_1R_2R_3)MNPQ &\longrightarrow \mathsf{Abs}\ R_3\ R_2(\mathsf{Ext}\ MQN)P\\
&\qquad (R_1R_2R_3 \text{ compound.})
\end{aligned}
$$

Fig. 3. Reduction rules of abstraction calculus

Theorem 3 (abstraction_confluence). *Reduction of the abstraction calculus is confluent.*

Proof. There are no critical pairs.

The normal forms are given by the factorable forms whose sub-terms are all normal.

Theorem 4 (abstraction_progress). *Every term of abstraction calculus is either a normal form, or reduces.*

Proof. A straightforward induction on the structure of the term suffices.

Consider the variant of closure calculus where the variables are indexed by natural numbers, i.e. are of the form x_i for some natural number i, built from Zero and Succ. Then there is translation $[\![-]\!]$ from variables to terms of abstraction calculus given by

$$\begin{aligned}[\![\mathsf{Zero}]\!] &= K \\ [\![\mathsf{Succ}\ i]\!] &= S[\![i]\!] \ .\end{aligned}$$

Then sequences of variables are translated by

$$\begin{aligned}[\![x_i, \mathsf{nil}]\!] &= [\![i]\!] \\ [\![x_i, y, ys]\!] &= S\ [\![i]\!]\ [\![y, ys]\!] \ .\end{aligned}$$

Then the terms of closure calculus can be translated to the abstraction calculus by

$$\begin{aligned}[\![x_i]\!] &= \mathsf{Var}\ [\![i]\!] \\ [\![s, t]\!] &= \mathsf{Tag}\ [\![s]\!]\ [\![t]\!] \\ [\![s\ t]\!] &= [\![s]\!]\ [\![t]\!] \\ [\![\lambda x, xs[\sigma].t]\!] &= \mathsf{Abs}\ [\![xs]\!]\ [\![x]\!]\ [\![\sigma]\!]\ [\![t]\!] \\ [\![I]\!] &= I \\ [\![x \mapsto u :: \sigma]\!] &= \mathsf{Ext}\ [\![x]\!]\ [\![u]\!]\ [\![\sigma]\!] \ .\end{aligned}$$

Theorem 5 (tagged_to_abstraction). *The translation above from closure calculus to abstraction calculus preserves reduction, in the sense that a single step $M \longrightarrow N$ in the closure calculus yields a multi-step reduction $[\![M]\!] \longrightarrow^* [\![N]\!]$ in the abstraction calculus.*

Proof. Routine case analysis.

It follows that the abstraction calculus is Turing complete, since the interpretation of recursion within closure calculus is carried over to the abstraction calculus.

4 Interpreting Closure Calculus in SF-Calculus

The intensional operators of abstraction calculus, namely E and Ext and Abs support the minimum amount of intensionality required to represent abstraction. Even so, the resulting system has 533 reduction rules! The expressive power can

be increased, to support full intensionality, by adding yet one more operator, namely a factorisation operator F. However, the number of reduction rules will then swell to over 700! These numbers can be reduced by shrinking the collection operators to just S and F. Now there are only 6 factorable forms, for a total of 7 reduction rules. In particular, factorisation by F allows the test for equality to be reduced to a test for equality of operators, which can be defined in terms of S and K alone, since S and F are extensionally separable. On the other hand, the algorithmic information contained in the reduction rules of abstraction calculus must be captured within terms of SF-calculus. Each of the operators $\mathsf{Tag}, \mathsf{Var}, \mathsf{Ext}$ and Abs yields a combination of S and F whose size is several thousand operators long!

However, the resulting translation of abstraction calculus to SF-calculus does not quite preserve reduction, since the various term forms must reduce somewhat before the intensional computation begins. Hence, we shall begin with an interpretation of closure calculus, in the directory "SF-for-lambda".

Theorem 6 (tagged_to_SF_preserves_reduction). *The translation from closure calculus to SF-calculus preserves multi-step reduction.*

Theorem 7 (tagged_to_SF_preserves_normal). *The translation from closure calculus to SF-calculus preserves normal forms.*

The traditional statement of the problem of the ξ-rule presumed that variables form a syntactic class that is separate to that of the terms, but here the variables are given by combinations of the form $\mathsf{var}\ x$ where var is a combination and x is the name of the variable. In translating from λ-calculus, variables are injectively mapped to normal forms, whose equality can be decided. However, there is no barrier to the computation of names, which will be considered in future work.

5 Future Work

The translations to SF-calculus from closure calculus or abstraction calculus to SF-calculus above are not very practical, since the representation of abs contains approximately fifty thousand operators! From one point of view, this is a clear demonstration of just how complicated λ-abstraction is, even in the closure calculus. On the other hand, this account is heavy compared to existing implementations, which is unsatisfying.

One way forward is to implement some intermediate calculus between closure calculus and SF-calculus that contains a few operators as primitives, and perhaps even λ-abstraction as a primitive, too.

Having expressed abstractions in combinatory form, it is natural to consider program analysis and optimisation in this setting. The first point to notice is that complete analysis of terms requires that they be in normal form, so a general analysis of programs requires that all programs be in normal form. Using a Universal Turing machines, this is quite natural, as programs are sequence of

symbols, just like any other data. However, in the traditions of the λ-calculus, recursive programs are given by fixpoint functions, which never have normal forms. This also holds for the traditional accounts of fixpoints in combinatory logic. However, as noted earlier, closure calculus, is able to represent fixpoint functions as normal forms! Indeed, this is also true of traditional combinatory logic, with non-termination arising only when fixpoint functions are fully applied, in a manner analogous to the Turing model of recursion. So our first task will be to explain this new combinatory approach to recursion.

That done, our primary task will be to optimize the translations of abstractions the translations. Preliminary work suggests that little effort is required to optimise the translation of $\lambda x.x$ to something smaller, namely I.

That variable names are fixed suggests new approaches to program debugging, since the programmer will be able to track a variable name through all stages of evaluation.

That a variable is now given by applying the combination var to the variable's name x means that variable names are now first-class citizens, that we may compute with names. This suggests all sorts of interesting possibilities. For example, imperative programming is distinguished by its ability to compute names, such as the next index into an array, or the result of some pointer arithmetic. Similar remarks may apply to both lower-level languages, that compute with registers, or higher-level languages, that compute names by using paths through some hierarchy, or ontologies.

6 Conclusions

The main result of this paper is that λ-abstractions can be defined as applications of a combination abs of the elementary operators of SF-calculus to some arguments. Unlike the traditional account in SKI-calculus, this approach to abstraction preserves reduction of the term being abstracted, preserves the ξ-rule, so that λ-abstraction is now a first-class operation on combinations. Also, there is no need for any meta-theory for performing substitutions, or renaming variables. Rather, all of the complexity of λ-calculus has been expressed within the combinations used in its translation.

In order to achieve this, it was first necessary to recast the λ-calculus in a manner that eliminated the traditional meta-theory that performs substitutions and variable renaming. In closure calculus, the intensional computation of substitution is broken down into small steps that can be expressed as reduction rules. Then renaming is avoided by using closures, so that substitution is delayed until all bound variables have acquired values. The resulting calculus is subtly different from pure λ-calculus, but the core requirements, of supporting abstraction, being confluent, and supporting numerical computation (being Turing complete) are all met.

That done, it is relatively simple to represent the term forms of closure calculus as operators, in abstraction calculus. The main difference is that the variables no longer form a separate class, but are given by terms of the form

Var x where x is a normal form that represents the variable name. Equality of names is decided by the equality operator E.

The final step is to represent the operators of abstraction calculus in terms of the primitives S and F. This can be done as the SF-calculus is intensionally complete, and so able to capture the intensionality required to decide equality, perform substitutions, etc.

As well as resolving a long-standing problem, of the status of the ξ-rule in combinational settings, of clarifying the relationship between combinations and abstraction, this account suggests new directions in program analysis and optimisation.

References

1. H.B. Curry and R. Feys. *Combinatory Logic*, volume 1 of *Combinatory Logic*. North-Holland Publishing Company, 1958.
2. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, May 2001. URL: `http://doi.acm.org/10.1145/382780.382785`.
3. M. Fernandez, I. Mackie, and F-R. Sinot. Closed reduction: explicit substitutions without α-conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.
4. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):911–937, November 2004.
5. Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009.
6. Barry Jay. Programs as data structures in λSF-calculus. *Electronic Notes in Theoretical Computer Science*, 325:221 – 236, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).
7. Barry Jay. Self-quotation in a typed, intensional lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 2017. The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII), to appear.
8. Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *Journal of Symbolic Logic*, 76(3):807–826, 2011.
9. Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.
10. Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, pages 247–58, 2011.
11. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.
12. Delia Kesner. The theory of calculi with explicit substitutions revisited. In *International Workshop on Computer Science Logic*, pages 238–252. Springer, 2007.
13. Garrel Pottinger. *A Tour of the Multivariate Lambda Calculus*, pages 209–229. Springer Netherlands, 1990. URL: `https://doi.org/10.1007/978-94-009-0681-51-14`.
14. Terese. *Term Rewriting Systems*, volume 53 of *Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.