A Simpler Lambda Calculus

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

Abstract

Closure calculus is simpler than pure lambda-calculus as it does not mention free variables or index manipulation, variable renaming, implicit substitution, or any other meta-theory. Further, all programs, even recursive ones, can be expressed as normal forms. Third, there are reduction-preserving translations to calculi built from combinations of operators, in the style of combinatory logic. These improvements are achieved without sacrificing three fundamental properties of lambda-calculus, being a confluent rewriting system, supporting the Turing computable numerical functions, and supporting simple typing.

CCS Concepts • **Theory of computation → Equational logic and rewriting**; **Functional constructs**; *Type theory*;

Keywords lambda calculus, closures, closure calculus

ACM Reference Format:
Barry Jay. 2019. A Simpler Lambda Calculus. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '19), January 14–15, 2019, Cascais, Portugal.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3294032.3294085

1 Introduction

Lambda calculus [3] can be seen as a theory of intensional functions, in which the algorithm for producing a value from a given input x is represented by a λ-abstraction $\lambda x.t$ where the term t describes the operations that are to be performed on the input x. Since it is Turing complete, it can be a foundation for programming languages. Since it supports higher-types, it can be a foundation for theorem-provers. Since it is a confluent rewriting system, any term can be replaced by its value, which supports aggressive partial evaluation.

PEPM '19, January 14–15, 2019, Cascais, Portugal

ACM ISBN 978-1-4503-6226-9/19/01…$15.00
https://doi.org/10.1145/3294032.3294085

Nevertheless, pure λ-calculus has three weaknesses. First, it requires an elaborate meta-theory of variable renaming. This makes it difficult to reason about programs or to implement them. Second, the representations of recursive functions, including those used to demonstrate Turing completeness, lack normal forms. This is troublesome for partial evaluation, as special care is required to avoid useless unfolding of fixpoints. Third, it is not combinatory, in the sense that the known translations to combinatory logic [8] do not preserve reduction. There is a disconnect between the λ-calculus model of computation and combinatory logic, even though they are both Turing-complete, confluent typed rewriting systems.

This paper introduces a new variant of λ-calculus, *closure calculus*, which is a Turing-complete, confluent, typed rewriting system that repairs all of these weaknesses. As such, it may be a better foundation for programming and program analysis, though this will require further investigation.

Although the definition of closure calculus is quite elementary, it is incompatible with traditional thinking about λ-calculus, which may make it difficult to engage. For example, the free variables of a term can be defined in the usual syntactic manner, but then a term may have free variables which are immune to substitution. This is not a problem for the formalism, which makes no reference to free variables anyway, but it is a shock to experts. What is the point of a free variable that cannot vary? Something must be wrong! Of course, such variables are useless, but every foundational calculus contains some useless terms, so one cannot object on this score. Another natural reaction is to try and see closure calculus as an outgrowth from other variants of λ-calculus. Since it uses *explicit substitutions* [1, 17], why create yet another explicit substitution calculus? There is nothing new here! However, our purpose is completely different. Such calculi were invented so that the timing of substitutions could be controlled by the evaluation strategy, not to eliminate the weaknesses identified above. In particular: none of them eliminate all meta-theory; none of them have been used to represent recursive programs as normal forms; and none of them support reduction-preserving translations to combinations. Closure calculus achieves all these properties by being simpler, not more complicated. Its simplicity emerges from a new way of thinking about computation that has developed, without fanfare, over the past twenty years.

The fundamental problem with λ-calculus is that it is an *extensional theory* of *intensional functions*. More precisely, its functions are intensional since two λ-abstractions f and g may be distinct even though they have the same input-output behaviour, i.e. are extensionally equivalent. However, the resulting theory is extensional since f and g cannot be distinguished by any context; they are observationally equivalent. In particular, neither equality of closed normal forms nor substitution is λ-definable.

Recent work has developed calculi that support intensionality, i.e. the ability to query internal structure directly. Pattern calculus [11, 15] generalises λ-abstraction to pattern-matching functions, which can query data structures in a uniform manner. *SF*-calculus [14, 16] can define queries that apply to arbitrary normal forms, both data and functions, and this in an axiomatic manner. λSF-calculus [12] adds to this native support for λ-abstraction but there is a blowout in the meta-theory required to identify when an abstraction is ready to be queried. Closure calculus emerges from the desire to eliminate this meta-theory, to produce an account of λ-abstraction that is compatible with the intensionality of *SF*-calculus. In this sense, the third goal, of producing a reduction-preserving translation from a λ-calculus to a calculus of combinations, was the driver for this work. Without this background, the inspiration for closure calculus may be obscure but even so, the benefits for extensional computation are clear.

Typing closure calculus is fairly straightforward, once substitutions are typed. This is by judgements of the form $\Gamma \vdash \sigma : \Delta$ where Γ is the type context for the free variables in the substitution σ and Δ is the type context that describes the types of the bindings in σ. It follows that type contexts must be types. Traditionally, a type context is given by a sequence or tuple of type assignments $x_i : U_i$ but then types would contain term variables, which is not very appealing. Instead, by using indices to represent term variables, a type context becomes a product $U_0 * U_1 \ldots U_i * \ldots U_n * \mathbf{1}$ of the types of the indices.

The main technical contributions of this paper are to introduce closure calculus and show that it supports:

1. λ-abstraction
2. equational reasoning (confluent rewriting with no meta-theory for free variables etc.)
3. numerical computation (Turing complete)
4. recursive programs in normal form
5. simple typing
6. translations to combinations (as in combinatory logic).

The structure of the paper is as follows. Section 1 is the introduction. Section 2 introduces closure calculus. Section 3 develops its basic properties. Section 4 translates closure calculus to calculi of combinations. Section 5 discusses related work. Section 6 considers partial evaluation and program manipulation. Section 7 draws conclusions. All named theorems have been verified in Coq.

$$
\begin{array}{rcl}
Jt & \rightharpoonup & J@t \\
Rtu & \rightharpoonup & Rt@u \\
(r@t)u & \rightharpoonup & (r@t)@u \\
(\lambda st)u & \rightharpoonup & (u,s)t \\
It & \rightharpoonup & t \\
(u,s)J & \rightharpoonup & u \\
(u,s)(Rt) & \rightharpoonup & st \\
(u,s)(r@t) & \rightharpoonup & (u,s)r((u,s)t) \\
(u,s)\lambda rt & \rightharpoonup & \lambda((u,s)r)t \\
(u,s)I & \rightharpoonup & I \\
(u,s)(r,t) & \rightharpoonup & ((u,s)r,(u,s)t)
\end{array}
$$

Figure 1. Reduction rules of closure calculus

2 Closure Calculus

Closure calculus has *terms* r, s, t, u given by

$$r, s, t, u, v \quad ::= \quad J \mid Rt \mid r@t \mid \lambda st \mid I \mid u, s \mid tu \ .$$

Instead of having a separate class of variables, *indices* are used to indicate variable binding. For example, J is the zeroth index, which is bound by the immediate enclosing λ. Then RJ is bound by the second enclosing λ etc. This adapts de Bruijn's technique in two ways: indices are constructed much as any other terms are; and there is no meta-theory for lifting and lowering indices. For convenience, we may use the natural number i to represent $R^i J$ so that 0 is J and 1 is RJ etc. A *frozen* or *tagged application* $r@t$ never reduces until thawed. A *closure* λst binds the variable 0 in t. Then s is its *environment* or *substitution* which may supply values for other indices in t. The constant I is the identity function, and also is the empty substitution. The *pair* u, s is also a *substitution* which binds J to u. Applications are as usual. We may write $[u_0, u_1, \ldots, u_n]$ for $(u_0, (u_1, \ldots, (u_n, I)))$. Then u_i is the value of the ith variable. Like application, tagged application associates to the left, but binds more tightly than ordinary applications. As usual, abstraction binds as far to the right as possible. Pairing associates to the right.

The *reduction rules* of the calculus are given in Figure 1. Applications of J or Rt or a tagged application may be tagged. The β-reduction rule adds the argument u to the context s and applies this substitution to the body t of the closure. Applications of substitutions (u, s) query the internal structure of their argument. If the argument is J then return u. If the

argument is of the form Rt then apply s to t. In this manner $[u_0, u_1, \ldots u_i, \ldots, u_n](R^i J)$ reduces to u_i. If it is a tagged application $r@t$ then thaw the application and apply the substitution to each of r and t. If it is a closure λrt then apply the substitution to r 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, but this presumption need not be enforced. Applications of pairs to pairs are performed pointwise. The *reduction relation* $l \longrightarrow r$ is obtained by applying a reduction rule to a subterm of l. The reflexive, transitive closure of $\longrightarrow$ is denoted $\longrightarrow^*$.

2.1 Remarks

It is unusual to combine the forms for indices, terms and substitutions into a single syntactic class. However, the theory becomes simpler, no harm comes from their mixing, and it may prove to be useful. The various distinctions can be enforced by a type system, as in Section 3.6.

Closures shadow any free variables in their body, in that substitutions never act on the body, so there is no possibility of scope violations, and so no need to ever rename variables, or lift indices in de Bruijn style. Variable names remain stable throughout a computation!

Although all of the mechanics of computation work smoothly, e.g. there are no critical pairs of reduction rules, this whole approach is incompatible with traditional thinking about variables. For example, what are we to make of the following reduction

$$(v, s)(\lambda I(RJ)) \longrightarrow \lambda((v, s)I)(RJ) \longrightarrow \lambda I(RJ) \ ?$$

By the standard calculations, J is free in $\lambda I(RJ)$ so why is it not bound to v? The short answer is that the notions of free and bound variables are *not* part of the reduction machinery, so it is a problem for our explanations, not a problem for the calculus itself. The practical answer is that $\lambda I(RJ)$ is poor style, and should be replaced by $\lambda(J, I)(RJ)$ in which the environment binds the free variable J to itself and so represents all free variables in the body of the closure. Then we have

$$(v, s)\lambda(J, I)(RJ) \longrightarrow^* \lambda(v, I)(RJ)$$

in which the value v of J awaits access to the closure body. The long answer is that types can enforce this good style by supporting our intuitions about free and binding occurrences.

A frozen application is never a redex. Freezing can only be eliminated by applying a substitution, which removes the tag. For example, we have the reduction sequence

$$\begin{array}{rcl} [u](01) & \longrightarrow & [u](J@(RJ)) \\ & \longrightarrow & [u]J\ ([u](RJ)) \\ & \longrightarrow & u([u](RJ)) \\ & \longrightarrow & u([\,]J) \\ & \longrightarrow & uJ\ . \end{array}$$

A more traditional approach would achieve this result without freezing but the extra cost does not arise in practice, since the useless evaluation of the body $J(RJ)$ will generally proceed before any substitution is performed.

3 Basic Properties

3.1 A Confluent Rewriting System

Closure calculus is a rewriting system, in that reductions can be applied to any sub-term.

Theorem 3.1. *Reduction is confluent.*

Proof. There are no critical pairs. That is, the application of a reduction rule may eliminate, or duplicate, some other redex $t\ u$ but it cannot break the redex, so that the subterms t and u appear separately. So confluence follows from general principles. □

3.2 Fixpoint Functions

Turing's fixpoint operator is defined by

$$\begin{array}{rcl} \omega & = & \lambda[]\lambda[J]J@(RJ@RJ@J) \\ Y & = & \lambda[\omega]J@(RJ@RJ@J) \end{array}$$

Note that Y is a value, an irreducible term.

Lemma 3.2 (fixpoint_property)**.** *For all terms f of closure calculus, Yf reduces to $f(Yf)$.*

It is built into this fixpoint property that such fixpoint functions Yf are never strongly normalizing. However, closure calculus provides the fine control necessary for fixpoints to preserve normal forms.

Define

$$\begin{array}{rcl} \omega_2 & = & \lambda[]\lambda[J]\lambda[J, RJ] \\ & & RJ@(R(RJ))@(R(RJ))@(RJ))J \\ Y_2 & = & \lambda[\omega_2]\lambda[J, RJ] \\ & & RJ@(2R(RJ)@R(RJ)@RJ)J\ . \end{array}$$

Note that $\omega_2\omega_2$ reduces to Y_2 which is normal.

Lemma 3.3. *For all terms f and x of closure calculus, $Y_2 f$ is strongly normalizing if f is, and $Y_2 fx$ reduces to $f(Y_2 f)x$.*

Proof. $Y_2 f$ reduces to

$$\lambda[f, \omega_2]RJ@R(RJ)@R(RJ)@RJ)J$$

which is a value. Further,

$$\begin{array}{lcl} Y_2 f x & \longrightarrow & [x, f, \omega_2] \\ & & (RJ@(R(RJ)@R(RJ)@(RJ))J) \\ & \longrightarrow & f\ (\omega_2\omega_2 f)\ x \\ & \longrightarrow & f\ (Y_2 f)\ x \end{array}$$

as required. □

3.3 Arithmetic

It follows that closure calculus supports arithmetic, including all of the μ-recursive functions [18]. Curiously, this is *not* achieved using the Church numerals but is achieved using the Scott numerals. Let us consider them in turn.

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

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

Then the predecessor of zero reduces to zero via

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

by *substituting under the lambda*, which is *not* possible in closure calculus.

The Scott numerals of pure λ-calculus [20] represent zero and successor by $\text{zero} = \lambda x.\lambda y.x$ and $\text{succ} = \lambda n.\lambda x.\lambda y.yn$ respectively, in the spirit of algebraic data types. Then the predecessor can be represented by

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

so that application to some numeral n is able to apply n immediately, without waiting for other arguments or substituting under the λ. 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.

For example, we can define addition by the normal form

$$\begin{array}{lcl} \text{plus} & = & Y_2(\lambda[](\lambda[0]1@(\lambda 0)@ \\ & & \lambda[0,1]\lambda[0,1,2]3@1@(\text{succ}@0)))\ . \end{array}$$

3.4 No Meta-Theory

Although closure calculus has many reduction rules, compared to the single rule of pure λ-calculus, there is absolutely no meta-theory required. Neither variable renaming nor index lifting is required. Nor are there references to free variables, renaming (α-conversion), implicit substitution, or index manipulation.

$$\frac{}{U * \Gamma \vdash J : U}$$

$$\frac{\Gamma \vdash u : U}{T * \Gamma \vdash Ru : U}$$

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

$$\frac{\Gamma \vdash \sigma : \Delta \quad U * \Delta \vdash t : T}{\Gamma \vdash \lambda\sigma t : U \to T}$$

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

$$\frac{}{\Gamma \vdash I : \mathbf{1}}$$

$$\frac{\Gamma \vdash u : U \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash u, \sigma : U * \Delta}$$

$$\frac{\Gamma \vdash \sigma : \Delta \quad \Delta \vdash t : \tau}{\Gamma \vdash \sigma t : \tau}$$

Figure 2. Type derivation for closure calculus

3.5 Normal Forms

The *normal forms* n are given by BNF

$$n ::= J \mid Rn \mid n@n \mid \lambda nn \mid I \mid n, n$$

in which all term forms are supported except (untagged) application.

Theorem 3.4 (closure_progress)**.** *Every term t is either a normal form, or reduces.*

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

It follows that all μ-recursive functions can be represented by normal forms [13], as they are given by applicaitons of R.

3.6 Typing

This section provides a simple type system for closure calculus. The main point of interest is to type the environments by type contexts. The *simple types* T and *type contexts* Γ and *general types* τ are given by

$$\begin{array}{rcl} T, U & ::= & X \mid U \to T \\ \Gamma, \Delta & ::= & \mathbf{1} \mid U * \Gamma \\ \tau & ::= & T \mid \Gamma\ . \end{array}$$

The types consist of *type variables* $X, Y \ldots$ and *function types* $U \to T$. The *type contexts* Γ or Δ are of the form $U_0 * \ldots U_n * \mathbf{1}$.

In this context the index i has type U_i. That is, there are no indices, or variables, for substitutions. The substitutions will be typed by type contexts. The separation of simple types from type contexts is used to ensure that indices cannot be instantiated by substitutions, which seems to be important for subject reduction. The type derivation rules are given in Figure 2, using judgements of the form $\Gamma \vdash t : \tau$. A closure $\lambda\sigma t$ has type $U \to T$ in context Γ if σ has type Δ in context Γ and t has type T in a context $U * \Delta$. Thus, all free variables in the body of a closure must be bound in this augmented environment. The application σt of a substitution σ to a term t is typed by some type τ in some context Γ as follows. σ must have type given by some type context Δ while t has type τ in context Δ. The tuples (I and pairs) are typed in the obvious manner.

Theorem 3.5 (Subject_reduction)**.** *If* $\Gamma \vdash t : \tau$ *and* $t \longrightarrow t'$ *then* $\Gamma \vdash t' : \tau$.

Proof. The proof is by routine case analysis on the structure of the type derivation. □

4 Translating to Combinations

The remaining property of interest is to produce reduction-preserving translations to combinations of operators.

4.1 Abstraction Calculus

The simplest translation of closure calculus to combinations is to *abstraction calculus*, which is here introduced. Its operators

$$O ::= J \mid R \mid H \mid A \mid I \mid B$$

are inspired by closure calculus. The operators J and R are for zero and successor. H is used to freeze, or hold its arguments. A is for abstraction by closures. I is unchanged. B is used to create bindings or pairs. 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 above. For example, the translation of λIJ is

$$AIJ\ .$$

Each operator has an *arity* given by 1, 2, 3, 3, 1 and 3 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. Then the terms M and N are then the *left-* and *right-hand components* of P. The *lambda forms* are those terms of the form

$$J \mid RM \mid HMN \mid AMN \mid I \mid BMN$$

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 forms that are factorable is given by the sum of the arities, which is 13.

$$\begin{array}{rcl}
JM & \rightharpoonup & HJM \\
RMN & \rightharpoonup & H(RM)N \\
HMNP & \rightharpoonup & H(HMN)P \\
AMNP & \rightharpoonup & BPMN \\
IM & \rightharpoonup & M \\
BMNJ & \rightharpoonup & M \\
BMN(RP) & \rightharpoonup & NP \\
BMN(HPQ) & \rightharpoonup & BMNP(BMNQ) \\
BMN(APQ) & \rightharpoonup & A(BMNP)Q \\
BMN(BPQ) & \rightharpoonup & B(BMNP)(BMNQ) \\
BMNO & \rightharpoonup & O \quad (O \neq J) \\
BMN(PQ) & \rightharpoonup & BMNP(BMNQ) \\
 & & (PQ \text{ is a true compound})
\end{array}$$

Figure 3. Reduction rules of abstraction calculus

The reduction rules are given in Figure 3. As written, there are 12 reduction rules, but when the side-conditions are eliminated, there are 13 rules for B making a total of $5 + 13 = 18$ rules.

Theorem 4.1. *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 4.2 (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. □

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

$$\begin{array}{rcl}
[J] & = & J \\
[Rt] & = & R[t] \\
[s@t] & = & H[s][t] \\
[\lambda\sigma t] & = & A[\sigma][t] \\
[I] & = & I \\
[v, \sigma] & = & B[v][\sigma] \\
[s\ t] & = & [s]\,[t]\ .
\end{array}$$

Indeed, the only significant difference between the two calculi is that abstraction calculus allows partially applied operators, such as R or HH which are not from closure calculus.

Theorem 4.3. *(*CLOSURE_TO_ABSTRACTION-_PRESERVES_REDUCTION*) The translation above from closure calculus to abstraction calculus preserves reduction, in the sense that each reduction rule in closure calculus translates to a reduction rule of abstraction calculus.*

Proof. Routine case analysis. □

It follows that the abstraction calculus represents all μ-recursive functions, since the interpretation of recursion within closure calculus is carried over to the abstraction calculus.

Theorem 4.4. *(*CLOSURE_TO_ABSTRACTION-_PRESERVES_NORMAL*) The translation above from closure calculus to abstraction calculus preserves normal forms.*

Proof. Routine case analysis. □

Unlike the closure calculus, the types of the operators of abstraction calculus must consider more than one type context at a time, so instead of typings of the form $\Gamma \vdash t : T$ and $\Gamma \vdash \sigma : \Delta$ we will push the context Γ to the right of the turnstile, to get $t : \Gamma \to T$ and $\sigma : \Gamma \to \Delta$. Then function types of such types are able to type the operators. That is, the simple types T and type contexts Γ are as before, but now the general types τ are given by

$$\tau ::= \Gamma \to T \mid \Gamma \to \Delta \mid \tau \to \tau \ .$$

The different function types are distinguished by the nature of their argument and resut types.

The operators of abstraction calculus have types of the form

$$\begin{array}{rcl} J & : & U * \Gamma \to U \\ R & : & (\Gamma \to U) \to (T * \Gamma \to U) \\ H & : & (\Gamma \to U \to T) \to (\Gamma \to U) \to \Gamma \to T \\ A & : & (\Gamma \to \Delta) \to (U * \Delta \to T) \to (\Gamma \to U \to T) \\ I & : & \Gamma \to \mathbf{1} \\ B & : & (\Gamma \to U) \to (\Gamma \to \Delta) \to \Gamma \to U * \Delta \end{array}$$

Then there are four rules for typing applications, plus two weakening rules:

$$\frac{M : \Gamma \to U \to T \quad N : \Gamma \to U}{MN : \Gamma \to T}$$

$$\frac{M : \tau_1 \to \tau \quad N : \tau_1}{MN : \tau}$$

$$\frac{M : \Gamma \to \Delta \quad N : \Delta \to U}{MN : \Gamma \to U}$$

$$\frac{M : \Gamma \to \Delta_1 \quad N : \Delta_1 \to \Delta}{MN : \Gamma \to \Delta}$$

$$\frac{M : \mathbf{1} \to U}{M : \Gamma \to U}$$

$$\frac{M : \mathbf{1} \to \Delta}{M : \Gamma \to \Delta} \ .$$

Theorem 4.5 (Subject_reduction)**.** *If $t : \tau$ and $t \longrightarrow t'$ then $t' : \tau$.*

Proof. The proof is by routine case analysis on the structure of the type derivation. Unfortunately, the weakening rules create many more cases, but the proof has been verified in Coq. □

Theorem 4.6. *If $\Gamma \vdash t : \tau$ in closure calculus then $[t] : \Gamma \to \tau$ in abstraction calculus.*

Proof. The proof is by induction on the structure of the type derivation. □

4.2 *SF*-Calculus

Although the translation above is compact, abstraction calculus is not exactly axiomatic, as its operators have been carefully chosen to support the mechanics of abstraction and substitution. An axiomatic approach is supported by *SF*-calculus, which needs only three reduction rules (or seven, once side-conditions have been eliminated) to support all intensional computation. In particular, F is able to decompose compounds into their components, so that *SF*-calculus supports a generous class of pattern-matching functions, rich enough to define the reduction rules of closure calculus. Indeed, there is a reduction-preserving translation from closure calculus to *SF*-calculus.

Recall [14] that terms of *SF*-calculus are combinations of the two operators S and F where S is taken from combinatory logic, and F performs factorisation. Each operator will have arity three, so that terms of the form SM, SMN, FM and FMN will turn out to be head normal. Call these four forms the *compounds*. The reduction rules of *SF*-calculus are then given by three conditional rules

$$\begin{array}{rcll} SMNP & \rightharpoonup & MP(NP) & \\ FOMN & \rightharpoonup & M & (O \text{ is an operator}) \\ F(PQ)MN & \rightharpoonup & NPQ & (PQ \text{ is a compound.}) \end{array}$$

It is crucial to the soundness of the calculus that there are applications PQ which are not compounds, such as $PQ = FFFF$ which reduces to F.

Since we can define $K = FF$ and $I = SKK$ it follows that *SF*-calculus is at least as expressive as *SKI*-calculus, or traditional combinatory logic [9]. In particular, it supports a form of abstraction which is enough to represent the extensional operators J, R, H, A and I of abstraction calculus. However, the intensional operator B is naturally represented

as a pattern-matching function

$$\begin{aligned} &BMN = \\ &\mid J \to M \\ &\mid Rx \to Nx \\ &\mid Hxy \to BMNx(BMNy) \\ &\mid Axy \to A(BMNx)y \\ &\mid Bxy \to B(BMNx)(BMNy) \\ &\mid xy \to (BMNx)(BMNy) \\ &\mid x \to x \end{aligned}$$

Such pattern-matching is beyond the ability of extensional calculi, such as combinatory logic, but is easily represented in SF-calculus, in the sense that the encoding of B is given by a combination that contains thousands of copies of S and F. With enough work, we obtain the following result.

Theorem 4.7. *There is a translation from closure calculus to SF-calculus that preserves multi-step reduction.*

Although there is a formal proof in Coq of a similar result (i.e. for an earlier variant of closure calculus) it has not yet been adapted for the current version of closure calculus.

5 Related Work

5.1 Pure λ-Calculus

The terms of pure λ-calculus in de Bruijn notation can be translated to closure calculus as follows. If n is the largest free index of t then λt translates to $\lambda[0, 1, 2, \ldots, n-1]t'$ where n is the. However, closure calculus does not preserve all β-reductions of pure λ-calculus since substitutions do not pass under λs. Conversely, closure calculus terms can be translated to pure λ-terms, by removing tags and making substitutions implicit, but removing tags may introduce non-termination. If the goal were merely to produce a more practical version of pure λ-calculus then these conflicts would be a serious concern. However, closure calculus is proposed as a potential *improvement* over pure λ-calculus, for the reasons given in the introduction.

5.2 Explicit Substitution Calculi

There is a substantial body of work devoted to explicit substitution calculi[1]. As summarized by Delia Kesner [17] the goal of these calculi is to provide an intermediary between pure λ-calculus and its implementations that retains the good properties of λ-calculus while offering fine control over substitution in a way that is useful in practice. The main properties of interest were confluence, strong normalization and simulation, in the sense that β-reduction of pure λ-calculus is supported. Unfortunately, it has not been possible to achieve all of the goals. For example, λes [17] supports all three good properties, but many of the reduction rules are now conditional on free variable restrictions, which seems impractical.

The first challenge for explicit substitutions is to treat the application $\sigma(\lambda x.t)$ of a substitution σ to an abstraction $\lambda x.t$. Most approaches allow substitution under the λ but then variable renaming is usually required. An alternative is to combine the substitution and abstraction into a *closure*, which could be written written $\lambda[\sigma]x.t$. This avoids variable renaming but creates other difficulties. First, when a substitution ρ is applied to this closure, previous approaches yields a new closure $\lambda[\rho \circ \sigma]x.t$ where $\rho \circ \sigma$ is the *composition* of σ and ρ. In particular, if x is *not* in the domain of σ then $(\rho \circ \sigma)x$ is ρx. In practice, such compositions can quickly grow large. In theory, composition of substitutions corrupts the account of arithmetic since non-trivial substitution into a numeral cannot be eliminated! That is, when a non-trivial substitution σ is applied to a numeral, expressed as a closure $\lambda[I]x.t$ with empty environment I then $\lambda[\sigma \circ I]x.t$ may perhaps reduce to $\lambda[\sigma]x.t$ but not to the numeral $\lambda[I]x.t$.

Closure calculus resolves the situation as follows. When a substitution ρ is applied to a closure $\lambda\sigma t$ then ρ is not composed with σ but is *applied* to σ, to get $\lambda(\rho(\sigma))t$. Here, if $R^i J$ is not in the domain of σ then $\rho(\sigma)$ $(R^i J)$ is simply $R^i J$. Hence, ρ does not act on t except through σ which can be thought of as the environment of the closure. Now ρI reduces to I and so arithmetic is supported without any need for variable renaming.

The other key question is how to apply a substitution σ to an application $t\ u$. Traditionally, the answer was obvious, namely $\sigma\ t\ (\sigma\ u)$. This approach probably has the properties we seek but it has two deficiencies. The minor deficit is that it creates critical pairs of reductions if $t\ u$ is a redex, which complicates reasoning. The major deficit is that it complicates the translation of abstractions into combinations. To avoid these defects, we introduce a second form of application, a *tagged application* $t@u$ as well as the usual application $t\ u$ (similar to the markings of λ_ζ-calculus introduced by Munoz [22]). Now tagged applications are never redexes, and so are ready to have substitutions act upon them. On the other hand, untagged applications will always reduce, and so are never ready to have substitutions act on them.

Closest to closure calculus is the λ_ζ-calculus of Munoz [22]. They both support confluence and strong normalization but not β-reduction, avoid adding any side-conditions, avoid composition of substitutions, and use tagging, or marks to control substitution into applications. However, λ_ζ-calculus still substitutes under abstractions, so that the meta-theory for re-indexing is still required, and there is no translation to combinations either.

The explicit substitution calculus of Fernandez et al [7] avoids variable renaming while substituting under λ by requiring that the substitutions map variables to closed terms only. This eliminates a considerable part of the meta-theory, but retains the need to calculate free variables of terms, which can be a burden.

Typed explicit substitution calculi generally avoid the challenge of typing substitutions themselves. For example, in

typed λes-calculus, the type derivation rule for the application of a substitution requires that the substitution be atomic, in that it maps a single variable to a single value. No provision is made for typing environments that hold values for several variables, or for composite substitutions etc. Here, we have taken the radical step of typing a substitution by a type context, using the rule

$$\Gamma \vdash \sigma : \Delta \ .$$

It is not yet clear how this typing might correspond to a logic, according to the Curry-Howard Correspondence [6, 10].

6 Partial Evaluation and Program Manipulation

The simplicity of closure calculus suggests that it will be better as a core language than any form of pure λ-calculus. This suggestion has not yet been confirmed, though our initial experiments with naive interpreters (joint work with Xuanyi Chew) suggest that interpreters become smaller and simpler, and that execution uses fewer, quicker steps. So perhaps partial evaluation and program manipulation will also improve. Certainly, it will be convenient that recursive programs have normal forms.

At first glance, it might seem that partial evaluation is no longer required. Given a program p with static argument s simply reduce $p\ s$ to normal form. In practice, there is no risk of non-termination since p will be of the form $\lambda[]\lambda[0]v$ where v is a value, so $p\ z$ will reduce to the normal form $\lambda[s]v$. However, there is no substitution of s into v so this sort of partial evaluation is trivial. For example, if v is the exponential of index 0 by index 1 and s is the number three, then partial evaluation should produce $\lambda I(J * J * J)$ where $*$ is the multiplication of natural numbers. So the traditional partial evaluation is still non-trivial, but is no longer achieved by applying reduction rules according to some strategy.

That said, evaluation strategy only goes so far; there are many other program manipulations that are not realised by partial evaluation. In general, program analysis requires the program to be viewed as a data structure, a syntax tree, which can be queried and updated. This intensional programming is not supported in closure calculus, or any other variant of λ-calculus, but requires a calculus that supports intensional computation in a general manner. SF-calculus has the necessary expressive power, but the huge combinations necessary to represent substitutions make it impractical. Instead, it is a simple matter to add factorisation to abstraction calculus, though typing is not quite trivial. Now programs will remain small, but can be analysed within their own language by means of factorisation.

The traditional starting point for this approach is *self-interpretation* There are many examples of self-interpreters for untyped λ-calculus [2, 19–21, 23]. Now there are even *typed* self-interpreters [5] for λ-calculus, and *Jones-optimal* partial evaluators [4]. Such analyses begin by *quoting* the program, to convert an executable into a syntax tree. Quotation makes typing a challenge, which is why this work is so impressive. However, by working with an intensionally complete calculus, such as SF-calculus, there is no need for quotation: the very same term is both a data structure and an executable, so that there is the possibility of dynamic program analysis. Much remains to be done.

7 Conclusions

Closure calculus is a confluent, typed, higher-order rewriting system whose λ-abstractions can represent the μ-recursive numerical functions. That is, closure calculus has the key properties that make λ-calculus so attractive. Further, it dispenses with the meta-theory of terms that burdens pure λ-calculus. There is no meta-theory for substitution, variable renaming (α-conversion) or even for the classification of variables as free, or bound, etc. In this sense, it is superior to pure λ-calculus.

It also compares well to other explicit substitution calculi. These calculi aspire to the good properties of pure λ-calculus while being closer to implementations, but none has quite succeeded. Closure calculus has the good properties of the other explicit substitution calculi, and is closer than them to implementation, at least in so far as meta-theory is to be avoided. Its only "failing" is that it does not preserve some useless reductions of pure λ-calculus (that do not contribute to finding normal forms). However, if we accept the conclusion above, that closure calculus is superior to λ-calculus, then the ability to simulate pure λ-calculus is no longer a goal.

Of course, there is a huge literature on λ-calculus, pure and applied, demonstrating many fine properties that have not been touched on here, so it is a little early to declare a winner. However, initial results on typing and implementation seem promising.

The ability to define recursive programs in normal form implies that evaluation can be quite aggressive, so that all values are normal forms, not just head normal forms. In turn, this stability may make it easier to analyse programs.

We have seen how to translate closure calculus into abstraction calculus and SF-calculus in a manner that preserves reduction. This solution to a problem that has been open since the field was created is perhaps the strongest theorem to imply the superiority of closure calculus over pure λ-calculus, and further evidence of the superiority of SF-calculus over traditional combinatory logic, or SKI-calculus.

Much remains to be done, in both theory and practice. One of the more interesting theoretical questions is to find, or create, a logic that corresponds to closure calculus. Traditionally, substitution corresponds to cut-elimination, so such a logic may require cut-elimination to meet side-conditions on the structure of the proofs.

The practical work is to build interpreters and compilers for functional languages that use closure calculus as their core. By adding factorisation, it should be possible to manipulate programs in the source language, even to perform dynamic optimisation.

Finally, the discovery of closure calculus and its translation to *SF*-calculus is a win for intensional computation. That the true complexity of λ-abstraction is measured by abstraction calculus, and defined away in *SF*-calculus, demonstrates the power of intensional computation, even in the representation of extensional functions.

Acknowledgments

Thanks to participants at the Shonan meeting 115 on intensional and extensional aspects of computation, and to Xuanyi Chew, Thomas Given-Wilson, Roger Hindley, Delia Kesner, Randy Pollack, Masahiko Sato, Jose Vergara and the anonymous referees for their feedback.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques 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.

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

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

[4] Matt Brown and Jens Palsberg. Jones-optimal partial evaluation by specialization-safe normalization. *Proc. ACM Program. Lang.*, 2(POPL): 14:1–14:28, December 2017. ISSN 2475-1421. doi: 10.1145/3158102. URL http://doi.acm.org/10.1145/3158102.

[5] Matt Brown and Jens 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.

[6] Haskell Brooks Curry, Robert Feys, and William Craig. *Combinatory Logic*. North-Holland Amsterdam, 1972.

[7] 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.

[8] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008. ISBN 0521898854, 9780521898850.

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

[10] W.A. Howard. The formulae-as-types notion of construction. In R. Hindley and J. Seldin, editors, *To H.B. Curry: essays in Combinarory Logic, lambda calculus and Formalisms*. Academic Press, 1980.

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

[12] Barry 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).

[13] Barry Jay. Recursive programs in normal form (short paper). In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '18, pages 67–73. ACM, 2018.

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

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

[16] 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.

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

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

[19] Stephen C. Kleene. λ-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

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

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

[22] Cesar A. Munoz. Confluence and preservation of strong normalisation in an explicit substitutions calculus. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 440–7, Washington, DC, USA, 1996. IEEE Computer Society.

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