Abstraction as a Combinator

Barry Jay[1]

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

Abstract

Pure λ-calculus is defined using meta-functions for substitution and for controlling the scope of variables. This paper shows how to eliminate this meta-theory while simultaneously reducing the gap between the theory and its implementation. Then there is a translation from the λ-calculus to a combinatory calculus, called L-calculus, which preserves both normalisation and the size of the term. Finally, there is another such translation from the λ-calculus to a combinatory calculus, called T-calculus, but no such translation in the opposite direction. The proof of this, and many other results in the paper, have been verified in Coq. The implications for the foundations of computation are discussed.

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Although λ-calculus [3] and combinatory logic (SKI-calculus) [11] are both Turing complete, in that they can both define all of the general-recursive numerical functions, λ-calculus appears to be more expressive than combinatory logic, in that there is a structure-preserving translation from SKI-calculus to pure λ-calculus but none in the opposite direction. More precisely, although the well known conversion of an abstraction $\lambda x.t$ to a combinator $\lambda^* x.t$ preserves β-reduction, it manages to break *every* redex in t and so does not even preserve normal forms. That is, abstraction does not preserve reduction; the ξ-rule [8] is broken. As well as breaking redexes, the size of the terms grows exponentially in the number of nested abstractions.

This paper resolves the situation through three main results. First, it introduces the *delayed substitution λ-calculus* or $\delta\lambda$-calculus, which, by delaying substitution until a variable is found, avoids the need for any meta-function for substitution. There is an invertible translation between pure λ-calculus and $\delta\lambda$-calculus that preserves the relation between terms and their normal forms (if they have one).

Second, it provides a translation from $\delta\lambda$-calculus to a new combinatory calculus, the *L-calculus*, that preserves reduction, and this without any increase in term size.

Third, there is a reduction-preserving translation from L-calculus to another new combinatory calculus, the *T-calculus*, which has additional properties. In particular, since it supports *factorisation*, in the style of SF-calculus [17], we can prove that there is *no* translation from T-calculus to λ-calculus or SKI-calculus that preserves normalisation. In this sense, T-calculus is *more expressive* than λ-calculus.

Of course, the standard theory *does* produce a translation from T-calculus to λ-calculus that encodes each combinator by the Church numeral of its Gödel number. However, this encoding, like $\lambda*$ above, breaks *every redex*, and in particular does not preserve normalisation. The relative expressive power of the different calculi will be characterised in terms of *extensional* versus *intensional* computations in Section 6, once the machinery and theorems have been established.

The structure of the paper is as follows. Section 1 is the introduction. Section 2 introduces $\delta\lambda$-calculus. Section 3 introduces $\delta\lambda B$-calculus in which variables are represented

by de Bruijn indices. Section 4 introduces L-calculus. Section 5 introduces T-calculus. Section 6 discusses intensional computation in general. Section 7 draws conclusions. The proofs are managed as follows. Where the calculi use an unspecified collection of variables, the proofs are done on paper. Also, the proof of confluence of T-calculus is done on paper. Proofs of all other theorems are named and verified in [16] using the Coq theorem-prover [4].

2 Delayed Substitution λ-Calculus

Recall [3] that the *pure* λ-calculus has terms given by the BNF

$$r, t, u ::= x \mid \lambda x.t \mid tu$$

where x is a variable, taken from some unspecified, countable collection. Its sole reduction rule is the β-reduction rule

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

where $\{u/x\}t$ is the *substitution* of u for x in t. In turn, the meta-function of substitution is defined with due regard to the scope of abstractions, and the need to rename some bound variables by α-conversion. The η-reduction rule

$$\lambda x.tx \longrightarrow x \quad (x \text{ not free in } t)$$

is *not* included by default.

Here are some general definitions that will apply to all the reduction systems in this paper. The relation $\longrightarrow$ will also be used to represent the one-step reduction relation obtained from the reduction rule by closure under the term forming operations. The relation $\longrightarrow^*$ represents the reflexive, transitive closure of the one-step relation.

An *irreducible term* is a term that does not reduce. They may also be called the *normal forms* in anticipation of a characterisation of the irreducible terms. Two terms are *Kleene equal* if they have the same normal form. A translation $[-]$ from the terms of one applicative rewriting system to another such is a *Kleene homomorphism* if it preserves application, and Kleene equality. If, further it preserves $\longrightarrow^*$ then it is a *reduction homomorphism*. Now let us return to particulars.

The proof of confluence for reduction in pure λ-calculus is non-trivial because substitution into a redex tr performs substitution into t and r separately to produce terms t' and r'. So some work is required to show that $t'r'$ is also a redex, and that substitution preserves reduction.

Delayed substitution λ-calculus will avoid introducing a meta-level function of substitution by converting the β-reduction rule into a family of reduction rules that delay substitution until a variable is found. Further, to avoid breaking redexes, substitution into an application tr blocked if it is (or could reduce to) a redex. In terms of pure λ-calculus, this is to require that t is headed by a variable. However, since we are trying to avoid side-conditions, let us introduce a new syntactic class of *sequences of terms* s to represent the arguments of a variable.

Define the *delayed substitution λ-calculus*, or *$\delta\lambda$-calculus* as follows. The BNFs for the *terms* (meta-variables r, t, u) and the *sequences of terms* (meta-variable s) are given by:

$$\begin{aligned} r, t, u &::= x\langle s\rangle \mid \lambda x.t \mid tu \\ s &::= \varepsilon \mid s, t\ . \end{aligned}$$

$$
\begin{array}{rcll}
x\langle s\rangle u & \longrightarrow & x\langle s,u\rangle & \\
(\lambda x.x)u & \longrightarrow & u & \\
(\lambda x.y)u & \longrightarrow & y & (y \neq x) \\
(\lambda x.z\langle s,t\rangle)u & \longrightarrow & (\lambda x.z\langle s\rangle)u((\lambda x.t)u) & \\
(\lambda x.\lambda y.t)u & \longrightarrow & \lambda y.(\lambda x.t)u & (y \text{ is not free in } u)
\end{array}
$$

Figure 1 Reduction rules of delayed substitution λ-calculus

We may write $x\langle t_1, \ldots t_k\rangle$ for $x\langle \varepsilon, t_1, \ldots t_k\rangle$ or write x for $x\langle\varepsilon\rangle$. Free variables are defined in the usual manner. Its reduction rules are in Figure 1. The first rule allows a variable to collect all of its arguments into a sequence. Then the fourth rule can recognise a term headed by a variable without imposing any new side conditions.

The restriction on β-reduction implicit in these rules is really quite mild. For example, the standard construction of fixpoints becomes applications of

$$Y = (\lambda x.\lambda f.f\langle x\langle x, f\rangle\rangle)(\lambda x.\lambda f.f\langle x\langle x, f\rangle\rangle)$$

still has the property that Yf reduces to $f(Yf)$ using delayed substitution.

▶ **Theorem 1.** *Reduction in the delayed substitution λ-calculus is confluent.*

Proof. There are no critical pairs [24]. ◀

The curious thing about delayed substitution λ-calculus is that reduction is *not* stable under substitution. For example, substitution of a redex for the variable y converts the redex $(\lambda x.y)u$ into a non-redex. Although quite against tradition, this is, on balance, a good thing. First, since substitution no longer plays a central role, the loss of this property is of no great consequence. For example, it is no longer needed to prove confluence. Second, if we are to analyse programs within the calculus then it is essential to be able to distinguish term variables from, say, abstractions.

Now let us formalize the relationship between pure λ-calculus and delayed substitution λ-calculus. There are translations $[-]$ from delayed substitution λ-calculus to pure λ-calculus and back, defined as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
[x\langle\rangle] & = & x & [x] & = & x\langle\rangle \\
[x\langle s,t\rangle] & = & [x\langle s\rangle]\ [t] & [\lambda x.t] & = & \lambda x.[t] \\
[\lambda x.t] & = & \lambda x.[t] & [tu] & = & \text{match } [t] \text{ with} \\
[tu] & = & [t]\ [u] & & & \quad \mid x\langle s\rangle \Rightarrow x\langle s, [u]\rangle \\
& & & & & \quad \mid \_ \Rightarrow [t]\ [u]
\end{array}
$$

The main point is that the translations easily break down and build up sequences of terms. Clearly, these translations are inverses. Equally clearly, they preserve irreducibility. However, neither of these translations preserves reduction. From pure λ-calculus to delayed substitution λ-calculus this is by design, but there is the following weaker result.

▶ **Lemma 2.** *If $(\lambda x.t)u$ is a term of pure λ-calculus and t is normal then $[(\lambda x.t)u]$ reduces by delayed substitution to $[\{u/x\}t]$.*

Proof. The proof is by induction on the structure of t. If t is headed by a variable x then apply induction on the number of arguments to which x is applied. If t is an abstraction

$\lambda y.t_1$ then t_1 is also normal. By choosing y to be not free in u we have

$$\begin{array}{rcl} [(\lambda x.\lambda y.t_1)u] & = & (\lambda x.\lambda y.[t_1])[u] \\ & \longrightarrow & \lambda y.(\lambda x.[t_1])[u] \\ & \longrightarrow^* & \lambda y.[\{u/x\}t_1] \end{array}$$

The first reduction is by a reduction rule; the second reduction is by induction with respect to t_1. The result follows by applying the definitions. ◀

Conversely, delayed substitution reduction is more refined than β-reduction, which leads to the following result.

▶ **Theorem 3.** *If $t \longrightarrow t'$ by delayed substitution then $[t]$ and $[t']$ have a common reduct in pure λ-calculus.*

Proof. The proof is by induction on the size of t. ◀

▶ **Corollary 4.** *If t has normal form n in delayed substitution λ-calculus then $[t]$ has normal form $[n]$.*

Proof. Apply the theorem, and then observe that $[n]$ is irreducible in pure λ-calculus. ◀

The converse of the corollary is also true, but harder to prove.

▶ **Theorem 5.** *If t has normal form n in pure λ-calculus then $[t]$ has normal form $[n]$.*

Proof. By the standardisation theorem, left-most, outermost reduction of t yields n. The proof is by induction on the length k of this reduction sequence. If k is 0 then the result is immediate so assume that k is positive and that the result holds for all numbers that are strictly less than k.

Now proceed by induction on the structure of t. If t is a variable or abstraction then the result is immediate, so suppose that t is an application t_1u. If t_1 is not in normal form then left-most, outermost reduction proceeds by reducing t_1 to its normal form, n_1. Since t_1 is a sub-term of t and its reduction to n_1 takes no more than k steps, it follows that there is an intensional reduction from $[t_1]$ to $[n_1]$. So, without loss of generality, $t_1 = n_1$ is itself irreducible.

Now proceed by induction on the structure of n_1. If n_1 is a variable or application then left-most, outermost reduction of n_1u reduces u to normal form, and the result follows since u is a sub-term of t, so suppose that n_1 is an abstraction $\lambda x.t_2$. It follows that the first reduction yields $\{u/x\}t_2$. Since t_2 is a normal form, it follows that there is an intensional reduction from $[(\lambda x.t_2)u]$ to $[\{u/x\}t_2]$ by Lemma 2. Finally, apply induction with respect to k to produce the required delayed substitution reduction from $[\{u/x\}t_2]$ to $[n]$. ◀

▶ **Corollary 6.** *The translations from delayed substitution λ-calculus to pure λ-calculus and back again are inverse Kleene homomorphisms.*

Proof. Apply the preceding theorems. ◀

▶ **Corollary 7.** *There is a Kleene homomorphism from SKI-calculus to delayed substitution λ-calculus.*

Proof. Recall that there is a reduction homomorphism from SKI-calculus to pure λ-calculus given by

$$\begin{aligned}[S] &= \lambda x.\lambda y.\lambda z.xz(yz)\\ [K] &= \lambda x.\lambda y.x\\ [I] &= \lambda x.x\ .\end{aligned}$$

The required homomorphism is the composite of this and the homomorphism from pure λ-calculus to delayed substitution λ-calculus. Note that this is not a reduction homomorphism, since the reduction of $S(K(SII))(K(SII))I$ to $(SII)(SII)$ translates to a sequence of β-reductions whose final step

$$(\lambda y.(\lambda x.xx)(\lambda x.xx))(\lambda z.z) \longrightarrow (\lambda x.xx)(\lambda x.xx)$$

cannot be performed using delayed substitution. ◀

The delayed substitution calculus does not appear to have been studied previously. The reduction rules of the delayed substitution λ-calculus only ever perform *needed reductions* [2] and so must be related to some evaluation strategy of pure λ-calculus. However, unlike most strategies, its reduction rules are confluent. It is also reminiscent of *explicit substitution calculi*, introduced by Abadi et al [1] and surveyed by Kesner [21]. Such approaches replace β-reduction with a rule of the form

$$(\lambda x.M)N \longrightarrow [x := N]M\ .$$

The increased flexibility compared to pure λ-calculus arises from the potential for interaction between substitutions, either to accumulate them into an environment, or to permute them. From this point of view, delayed substitution λ-calculus supports a limited form of permutation through its β-rule for abstractions of abstractions. However, explicit substitution calculi are generally free to perform the substitution at any time, without requiring the body M to be a weak head normal form.

Another possible relation is to the work on *linear head reductions* [9, 22]. In this approach, β-reduction substitutes for the head variable only. Like the delayed substitution calculus, only one variable is handled at a time, but the linear head reduction requires side conditions to identify the head variable, which are avoided here.

Finally, *lambda abstraction algebras* [23] take quite a different approach to representing lambda-abstraction, both in the way of accounting for abstraction and, more generally, in constructing algebras for rewriting systems, rather than developing the systems themsleves.

3 de Bruijn Indices

The next step is to eliminate the other side-conditions in Figure 1, on variable occurrence. This can be done by replacing variables with de Bruijn indices [10] where the natural number i is used to represent the ith variable. We could use de Bruijn indices in three different ways. First, simply specify that the variables of λ-calculus are actually natural numbers, but make no other changes. For example, the last reduction rule in Figure 1 becomes

$$(\lambda j.\lambda k.t)u \longrightarrow \lambda k.(\lambda j.t)u \quad (k \text{ is not free in } u).$$

Second, we can remove the side-condition by requiring that abstraction always binds the variable 0, as is usual in the de Bruijn style, but then additional machinery for managing

$$
\begin{array}{rcll}
i\langle s\rangle u & \longrightarrow & i\langle s,u\rangle & \\
i\langle\,\rangle \uparrow (j,k) & \longrightarrow & i\langle\,\rangle & (i<k) \\
i\langle\,\rangle \uparrow (j,k) & \longrightarrow & (i+j)\langle\,\rangle & (i\geq k) \\
i\langle s,t\rangle \uparrow (j,k) & \longrightarrow & i\langle s\rangle \uparrow (j,k) & (t\uparrow(j,k)) \\
(\lambda i.t) \uparrow (j,k) & \longrightarrow & \lambda i.(t\uparrow(j,i+k)) &
\end{array}
\qquad
\begin{array}{rcll}
(\lambda j.i\langle\,\rangle)u & \longrightarrow & i\langle\,\rangle & (i<j) \\
(\lambda j.i\langle\,\rangle)u & \longrightarrow & \mathsf{pred}\ i\langle\,\rangle & (i>j) \\
(\lambda j.i\langle\,\rangle)u & \longrightarrow & u\uparrow(j,0) & (i=j) \\
(\lambda j.i\langle s,t\rangle)u & \longrightarrow & (\lambda j.i\langle s\rangle)u((\lambda j.t)u) & \\
(\lambda j.\lambda k.t)u & \longrightarrow & \lambda k.(\lambda(j+k+1).t)u &
\end{array}
$$

Figure 2 Reduction rules for delayed substitution calculus with de Bruijn indices

indices is required. For example, the last reduction rule in Figure 1 becomes

$$(\lambda\lambda t)u \longrightarrow \lambda(\lambda(\mathsf{swap}[t]))(\mathsf{lift1}\ u)$$

where lift1 is used to raise the indices in u by 1 to compensate for being within the scope of an additional binder, and swap is used to swap the indices 0 and 1 in t. However, this approach is inefficient, since it is better, and usual, to delay all index manipulations for as long as possible. The third, and prefered, approach is to name the bound variable, as in the first approach, but to use lifting to eliminate the side-condition. Now the last reduction rule in Figure 1 becomes

$$(\lambda j.\lambda k.t)u \longrightarrow \lambda k.(\lambda(j+k+1).t)u$$

where the lifting of u (by $j+k+1$) is performed lazily, using a new term form $u\uparrow(j,k)$ that raises by j all indices that are at least k. Lifting is introduced by the rule

$$(\lambda j.j)u \longrightarrow u\uparrow(j,0)\ .$$

The term forms for *delayed substitution λ-calculus with de Bruijn indices* are given by the BNF

$$
\begin{array}{rcl}
t,u & ::= & i\langle s\rangle \mid t\uparrow(j,k) \mid \lambda j.t \mid tu \\
s & ::= & \varepsilon \mid s,t
\end{array}
$$

where i,j and k are natural numbers. The reduction rules are given in Figure 2, where pred i is the predecessor of i. . They can be viewed as a formalisation of the usual properties of lifting and substitution in the de Bruijn style.

▶ **Theorem 8** (dlb_confluence). *Reduction in the delayed substitution λ-calculus with de Bruijn indices is confluent.*

Proof. There are no critical pairs. ◀

Define the *terms* t_n and *sequences* s_n *in normal form* by the BNFs

$$
\begin{array}{rcl}
t_n & ::= & i\langle s_n\rangle \mid \lambda j.t_n \\
s_n & = & \varepsilon \mid s_n,t_n\ .
\end{array}
$$

When a single BNF is used to describe both the terms and sequences of terms, as in the Coq implementation then the *well-formed terms* are exactly the terms and sequences of terms described in this section.

▶ **Theorem 9** (irreducible_iff_normal). *Every well-formed term is irreducible if and only if it is a normal form.*

$$
\begin{array}{rcl@{\qquad}rcl}
QZ & \longrightarrow & Z & Visu & \longrightarrow & Vi(Psu) \\
Q(Ni) & \longrightarrow & i & RUjk & \longrightarrow & U \\
HZj & \longrightarrow & j & R(Pst)jk & \longrightarrow & P(Rsjk)(Rtjk) \\
H(Ni)j & \longrightarrow & Hi(Nj) & R(Vis)jk & \longrightarrow & V(Diki(Hij)(Hij))(Rsjk) \\
DZZstu & \longrightarrow & u & R(Lit)jk & \longrightarrow & Li(Rtj(N(Hik))) \\
DZ(Nj)stu & \longrightarrow & s & Lj(ViU)u & \longrightarrow & Dij(ViU)(V(Qi)U)(RujZ) \\
D(Ni)Zstu & \longrightarrow & t & Lj(Vi(Pst))u & \longrightarrow & Lj(Vis)u(Ljtu) \\
D(Ni)(Nj)stu & \longrightarrow & Dijstu & Lj(Lit)u & \longrightarrow & Li(L(N(Hij))tu)
\end{array}
$$

Figure 3 Reduction rules for L-calculus

Usually, the relationship between the pure λ-calculus and its de Bruijn representation is treated informally since the correspondences are quite clear, and the de Bruijn representations are seen as merely a convenience that makes implementation easier. We will follow the same approach here. Complete formalisation would proceed by providing translations between the delayed substitution λ-calculus and its de Bruijn version and showing how they preserve structure. In this case, it is not especially hard work, but it is delicate, especially now that lifting is brought within the calculus. There appear to be uninvertible homomorphisms in each direction, but the details have not been fully explored.

At this point, the remaining side conditions to reduction (in Figure 2) involve numerical computations only, as the usual comparisons of variable names are replaced by comparisons of natural numbers, with variable scope handled by the functions of addition, the predecessor function and comparisons.

4 Abstraction as a Combinator

[7]

The next step is to create a combinatory calculus in which all of the side-conditions are eliminated by introducing operators for the numerical functions. The *combinators* of the *combinatory calculus for abstraction* or L-calculus are given by

$$t, u ::= O \mid tu \ .$$

where the operators O are given by

$$O ::= U \mid P \mid Z \mid N \mid Q \mid H \mid D \mid V \mid R \mid L \ .$$

U and P are used to build tuples. Z and N are zero and next number. Q, H and D perform predecessor, addition and comparisons, respectively. The discriminator D takes *five* arguments, namely two indices i, j to be compared, and three terms s, t and u. It reduces to s, t or u according to whether i is less than j or greater than j or equal to j. Then V constructs variables, R performs lifting or *raising* and L performs λ-abstraction.

Most of the operators can be motivated by giving the translation $[-]$ from delayed substitution λ-calculus with de Bruijn indices, given by

$$
\begin{array}{rcl@{\qquad}rcl}
[\langle\rangle] & = & U & [i\langle s\rangle] & = & V\ [i]\ [s] \\
[\langle s, t\rangle] & = & P\ [\langle s\rangle]\ [t] & [\lambda j.t] & = & L\ [j]\ [t] \\
[0] & = & Z & [tu] & = & [t]\ [u]\ . \\
[Ni] & = & N\ [i] & &
\end{array}
$$

For example, the projection function $\lambda x.\lambda y.\lambda z.x$ translates to

$$LZ(LZ(LZ(V(N(NZ))U)))$$

where L is the abstraction operator, that thrice binds the variable indexed (in de Bruijn's style) by Z (for zero), while $V(N(NZ))U$ is the variable (V) indexed by 2 (or $N(NZ)$) and having no arguments, as indicated by the unit value U. Note that there is no expansion in the size of the term, on the understanding that variables require explicit construction. This compares favourably with the traditional conversion $\lambda^* x.t$ of an abstraction $\lambda x.t$ to an SKI-combinator, which both hides the structure and grows exponentially in the number of nested abstractions. In this case it yields

$$S(S(KS)(S(KK)(KK)))(S(KK)I) \ .$$

The reduction rules are given in Figure 3. They exactly mirror the reduction rules and meta-level operations of the delayed substitution λ-calculus with de Bruijn indices.

▶ **Theorem 10** (abstraction_combinator_confluence)**.** *Reduction in the combinatory calculus for abstraction is confluent.*

▶ **Theorem 11** (translation_preserves_seq_red)**.** *The translation from delayed substitution λ-calculus with de Bruijn indices to L-calculus preserves $\longrightarrow^*$.*

▶ **Corollary 12.** *The translation from delayed substitution λ-calculus with de Bruijn indices to L-calculus is a reduction homomorphism.*

Proof. It is straightforward to show that the translation preserves irreducibility. ◀

Of course, the arithmetic operators Q, H and D can be defined by recursion, and so could be defined in terms of the traditional operators of combinatory logic, namely S, K and I. However, the need to represent recursion explicitly would make the reduction rules very long, and long rules are never axiomatic. Rather, the recursion that is implicit in, say, comparing indices i and j has been hard-wired into the rules for the discriminator D. In the other direction, the operators S, K and I can be modeled as λ-abstractions but, as with intensional λ-calculus, reduction is not always preserved.

L-calculus is not the first account of λ-calculus using intensional operators. Curien's *categorical combinatory logic* [5, 6] interprets λ-abstraction by an operator Λ that performs currying, and substitution by an operator $\circ$ for composition. These operators satisfy equations that, like L, examine the internal structure of their arguments. The main advantage of this approach over L-calculus is that it supports β-reduction exactly, without compromise. The disadvantages are as follows. First, the freedom to either substitute immediately, or to reduce the target of the substitution, creates a large number of critical pairs. Second, although the translations preserve β-equality, the equations of categorical combinatory logic cannot be ordered to produce a confluent rewriting system. Third, although the translation is much more efficient than previous efforts (being $O(nlogn)$), the translation to L-calculus does not increase term size at all. Another point of difference between the approaches is their orthogonal uses of tupling. For categorical combinators, tuples are used to represent the environment, so that variables becomes projections, application takes a pair of arguments, and abstraction is given by currying. In delayed substitution calculus and L-calculus, tuples are used to represent sequence of arguments to a variable, so that the head variable can be identified directly. Of course, such tuples can be created by reversing the process of currying, but this will hide the head variable, not expose it to analysis.

	Z	N	P
Z	$I = \|\|\|$	$Qi = \|\|(\|i)\|$ $Hij = \|\|(\|i)(\|j)$ $Rtjk = \|\|(\|t)(\|jk)$	$Ljt = \|\|(\|jt)$
N	$ViU = \|(\|i)\|$	$Sst = \|(\|s)(\|t)$	$Vi(Pst) = \|(\|i)(\|st)$
P	$Gst = \|(\|st)\|$	$Astu = \|(\|st)(\|u)$	$Dijst = \|(\|ij)(\|st)$

Figure 4 Representing operators as tallies

5 Tally Calculus

Tally calculus will support reduction homomorphisms from both SKI-calculus and L-calculus by supporting all of their operators, and a few others, as combinators built from a single operator, the *tally*, written $|$. Of course, these combinations are somewhat arbitrary, and give the appearance of, say, machine code, but the tally calculus is confluent, so that the representations of operators is similar to the representation of the pairing operator in λ-calculus by $\lambda x.\lambda y.\lambda f.fxy$. That is, it may not be practical, but it does serve to isolate the important features. A further benefit is that every irreducible term will have a meaning, so there are no "blocked" terms to worry about.

The representations proceed as follows. All of the constructors U, P, Z and N will be identified with the tally $|$. Since Z and U are nullary, N is unary and P is binary, this does not interfere with any of the ternary reduction rules for the tally. The other operators of L-calculus and of SKI-calculus are described in Figure 4, which is to be understood as follows.

The table describes the meanings of terms of the form $|uv$. The rows of the table describe the various possibilities for u, with Z meaning that u is $|$ and N meaning that u is of the form $|i$ and P meaning that u is of the form $|st$. Similarly, each column describes the status of the second argument v. For example, when both u and v are $|$ then the resulting term $|||$ is called I, and will behave as I usually does. Similarly, Ljt is given by $||(|jt)$. Note that this does not give an interpretation of L without arguments. When representing λ-abstractions this does not matter, but if L alone is required rather than Ljt then abstraction (using L again!) with respect to j and t will produce the desired combinator. Similar remarks apply to the other operators. Like most of the operations, L does not query its third argument. The exceptions arise with terms of the form $||(|i)$, whose nature is revealed only by examining the third argument, to yield a predecessor, addition or raise.

In this manner, all of the operators of L-calculus and SKI-calculus are already represented, except for K which is to come. Also, the operator D will become more powerful, while the combinators G and A are not yet introduced. Here A satisfies the rule

$$Astuv \longrightarrow |stu \ .$$

That is, $Astu$ will apply $|st$ to u only when supplied with a dummy parameter v. This sort of mechanism for delaying application is useful when building evaluation strategies. This use of a dummy parameter also generalizes K, which can now be defined by

$$Ku = A||u = |(|||)(|u) \ .$$

When applied to v it reduces to $|||u = Iu$ and so reduces to u.

$$
\begin{array}{rcl}
DZ(Ppq)stu & \longrightarrow & s \\
D(Ni)(Ppq)stu & \longrightarrow & s \\
D(Pmn)Zstu & \longrightarrow & t \\
D(Pmn)(Nj)stu & \longrightarrow & t \\
D(Pmn)(Ppq)stu & \longrightarrow & Dmpst(Dnqstu)
\end{array}
\qquad
\begin{array}{rcl}
Sstu & \longrightarrow & su(tu) \\
Astuv & \longrightarrow & |stu \\
It & \longrightarrow & t \\
Gst| & \longrightarrow & s \\
Gst(|u) & \longrightarrow & t|u \\
Gst(|uv) & \longrightarrow & t(|u)v
\end{array}
$$

Figure 5 Reduction rules for D, S, A, I and G

The combinator D now discriminates between arbitrary normal forms, not just indices. For example, we have $|$ is "less than" $|i$ and $|st$ for any i, s and t. It can be used to define equality by

$$Est = Dst(KI)(KI)K \ .$$

The combinator G will be used to factor compounds into their components, according to the following rules:

$$
\begin{array}{rcl}
Gst| & \longrightarrow & s \\
Gst(|i) & \longrightarrow & t|i \\
Gst(|uv) & \longrightarrow & t(|u)v \ .
\end{array}
$$

These rules are a slight variation on the rules for F in SF-calculus, as will be discussed in the following section. In this manner, every operation that has been discussed is represented in terms of the tally.

The interpretations given in the table have been chosen for a particular purpose, but others may prefer different interpretations. There are surely many different calculi with only one ternary operator. For example, there is no direct support for a fixpoint combinator Y in the table, which must be constructed from the other operators in the usual manner. However, one could as well use $|(|||)(|f)$ to represent Yf instead of Kf and then define K as an abstraction.

With these interpretations, the reduction rules of tally calculus are given in three groups, being the rules in Figure 3, the new rules for D, S, A, I and G in Figure 5 plus rules for mopping up in Figure 6. The latter are required because sorting has not been used to exclude non-standard applications, such $Q(Pst)$. The rule for mopping up come in two sets. First, where indices are expected, a pair Pst will be identified with Nt. Second, R and L leave $|$ unchanged, and commute with applications where no other rule has been given.

In all there are $16 + 11 + 14 = 41$ reduction rules. Of the 27 productive rules, 9 are for discrimination, with the other 18 shared between the other ten operators of Figure 4, averaging at under two rules per operation. Most of the rules for mopping up are used to commute raising and abstracting with other operators. One could halve the number of the latter rules by adding side-conditions, but this hardly seems worthwhile.

▶ **Theorem 13.** *Reduction of tally calculus is confluent.*

Proof. The reduction rules are all orthogonal [24] so there are no critical pairs. This proof has not yet been verified in Coq, because there are hundreds of cases to check. So some care is required to establish the orthogonality by hand. Examination of the reduction rules show that if a sub-term of a redex can be instantiated to be a redex, then the sub-term must

$$
\begin{array}{rcl@{\qquad}rcl}
Q(|st) & \longrightarrow & t & H(|si)j & \longrightarrow & Hi(|j) \\
R|jk & \longrightarrow & | & Lj|u & \longrightarrow & | \\
R(|t)jk & \longrightarrow & |(Rtjk) & Lj(|t)u & \longrightarrow & |(Ljtu) \\
R(||(|t)jk & \longrightarrow & ||(|(Rtjk) & Lj(||(|t)) & \longrightarrow & ||(|(Ljt) \\
RIjk & \longrightarrow & I & LjIu & \longrightarrow & I \\
R(Sst)jk & \longrightarrow & S(Rsjk)(Rtjk) & Lj(Sst)u & \longrightarrow & S(Ljsu)(Ljtu) \\
R(|(|st)u)jk & \longrightarrow & |(|(Rsjk)(Rtjk))(Rjnk) & Lj(|(|st)v)u) & \longrightarrow & |(|(Ljsu)(Ljtu))(Ljvu)
\end{array}
$$

Figure 6 Reduction rules for mopping up

be a meta-variable or the whole redex. Hence, it is enough to show that no term can be a redex with respect to two different instantiations of the rules. Now Figure 4 shows that the patterns for the eleven virtual operators do not overlap, so that it is enough to consider the rules associated to each virtual operator in isolation, which task is linear in the number of rules, rather than quadratic, and so is easily done. ◀

Define the *normal forms* to be the partial applications of the operators, as before, so that there are just three normal forms to consider.

▶ **Theorem 14.** *Every combinator of tally calculus is either a normal form or reduces.*

Proof. The proof is by a routine case analysis. ◀

▶ **Theorem 15.** *There is a reduction homomorphism from SKI-calculus to tally calculus, as per Figure 4.*

Proof. Since the translation of S is a normal form, β-reduction yields

$$[Sgfu] \longrightarrow^* S[g][f][u] \longrightarrow [g][u]([f][u]) = [gu(fu)]$$

as required. Similar remarks apply to K and I. ◀

▶ **Theorem 16.** *There is a reduction homomorphism from L-calculus to T-calculus, as per Figure 4.*

Proof. The proof is similar to that for translating SKI-calculus. ◀

▶ **Corollary 17.** *There is a reduction homomorphism from $\delta\lambda B$-calculus to T-calculus.*

Proof. Compose the homomorphisms involving L-calculus. ◀

▶ **Theorem 18.** *There is no normal homomorphism from T-calculus to SKI-calculus.*

Proof. Assume that there is such a homomorphism. Then there is another such to the pure λ-calculus, and another again to the pure λ-calculus plus η-contraction. It follows that the translation of S is η-equivalent to $\lambda x.\lambda y.\lambda z.Sxyx$ which is equivalent to $\lambda x.\lambda y.\lambda z.xz(yz)$. Similarly, the translation of K is equivalent to $\lambda x.\lambda y.x$ and the translation of I is equivalent to $\lambda x.x$. Further, it follows that the translation of SKX is equivalent to $\lambda x.x$ for every combinator X. Further, $GI(KI)(SKX)$ reduces to X. Hence the translation of X is equivalent to the translation of Y for every X and Y. However, the translations of K and I are *not* equivalent. Hence, there is no homomorphism. ◀

6 Intensional Completeness

Pure λ-calculus was created as a theory of functions, or at least as a theory of *extensional* functions. That is, an abstraction $\lambda x.t$ shows how the bound variable x may be used in t, i.e. be copied or deleted, but never queried, to determine its internal structure. Superficially, it appears to be of limited use, but it turns out that queries of natural numbers, such as "are you zero" or "what is your predecessor" can be defined extensionally, if rather inefficiently. In this manner, general queries, or *intensional* functions can be defined to the extent that the structures being queried can be converted to natural numbers, e.g. to their Gödel numbers. However, it is well known that Gödelization of λ-terms, even those in closed normal form, cannot be defined within λ-calculus, since then equality of λ-terms would be decidable. So it is a little curious that a theory of extensional functions should be defined using intensional meta-functions such as substitution.

From this point of view, it is no surprise that the extensional operators of SKI-calculus have not been able to capture the mechanics of β-reduction, but that the problem can be resolved by adding intensional operators, such as Curien's categorical combinators, or the operators L and T introduced here.

Intensionality has also appeared in *pattern calculus* [12] and *pure pattern calculus* [18, 13] generalize the pattern-matching of functional programming languages to support generic queries of data structures. Then SF-calculus [17, 19] extended the approach to support program analysis.

As well as the standard S-operator, it has a factorisation operator F which satisfies the following six reduction rules

$$\begin{array}{rcl@{\qquad}rcl} FSrt & \longrightarrow & r & FFrt & \longrightarrow & r \\ F(Sp)rt & \longrightarrow & tSp & F(Fp)rt & \longrightarrow & tFp \\ F(Spq)rt & \longrightarrow & t(Sp)q & F(Fpq)rt & \longrightarrow & t(Fp)q \end{array}$$

that show how to factorise partially applied operators. In the original paper, it was proved that there is *no way* to define F in SKI-calculus. This is consistent with traditional results about expressive power, provided the required encodings are taken into account [20]. The proof technique was adapted for tally calculus in Theorem 18. Further, SF-calculus was shown to support a powerful form of pattern-matching, to be *pattern complete*. However, this is somewhat complicated to define. An easier way of considering completeness (also suggested by Jacques Carette) is to observe that Turing completeness allows arbitrary queries of natural numbers, so if the calculus supports a means of converting terms (in closed normal form) to natural numbers, and back again, i.e. a Gödel function, then all queries can be reduced to numerical form.

Define a calculus to be *intensionally complete* if it is Turing complete and is able to define a Gödel function. It is routine to show that both SF-calculus and T-calculus are intensionally complete. It seems unlikely that the categorical combinators are intensionally complete.

Unfortunately, translation between intensionally complete calculi is non-trivial. For example, the translation of F into tally calculus cannot be $|$ and so must be an application. In turn, this application cannot a simple variant of G since G will treat the translation of F as an application, not as an operator. Rather, the behaviour of F in SF-calculus must be encoded as a pattern-matching function that acts on the translations of SF-terms, which is all rather complicated. Assuming this done, then a similar process must be applied in the opposite direction, which will not recover F from its translation, much as S cannot be recovered from its translation to pure λ-calculus.

The scale of the effort required can be seen in the construction of the Kleene homomorphism from $\delta\lambda$-calculus with de Bruijn indices to SF-calculus developed in previous work [15], where the combinator for abstraction was, if not astronomic, then genomic, comparable to the size of a genome. This explosion can be avoided by adding λ-abstraction with full β-reduction to SF-calculus to get λSF-calculus. It is non-trivial to factorise abstractions but it can be done to yield λSF-calculus [14]. Of course, λSF-calculus is also intensionally complete.

Summarising, although SF-calculus, λSF-calculus and T-calculus are all intensionally complete, none of them is superior to the others. SF has calculus has fewest reduction rules, but no native support for abstraction. λSF-calculus supports abstractions, but their factorisation blows out. Tally calculus supports abstraction without any combinatorial explosion, but has 41 reduction rules.

7 Conclusions

Pure λ-calculus turns out to be quite peculiar. Although it purports to be a general theory of functions, it accounts for extensional functions only. However, its β-reduction performs substitution by querying the the abstraction body, to determine if it is a variable, abstraction or application, in a manner that is quite beyond the abilities of λ-calculus itself. Indeed, it would not even be necessary if all functions were extensional. Further, this querying is so aggressive that it decomposes redexes into their constituents, so that serious effort is required to prove that reduction is stable under substitution. Since reduction queries its arguments, it is not surprising if a completely extensional calculus, such as SKI-calculus, has not been able to capture this behaviour.

However, once the issues are clear, it is straightforward to weaken the β-reduction rule so that substitution does not decompose any redexes, in $\delta\lambda$-calculus. This has all of the advantages of pure λ-calculus, e.g. in the ability to represent general recursion, has fewer disadvantages, e.g. fewer side-conditions, and is closer to programming practice, as illustrated by *explicit substitution calculi* [1, 21]. Then it is straightforward to replace the variables with de Bruijn indices in $\delta\lambda B$-calculus, so that the variable manipulations are expressed in terms of lifting, with the side-conditions expressed numerically. The next step is to translate $\delta\lambda B$-calculus to a combinatory calculus, the L-calculus, whose operators directly reflect the concerns of reduction in $\delta\lambda B$-calculus. Finally, the full power of intensional computation is realised by translating the L-calculus to T-calculus, which cannot then be translated back to λ-calculus without breaking the normalisation relation.

Summarising, this paper concludes some unfinished business in the foundations of computing, of finding a translation from λ-calculus to a combinatory calculus that preserves normalisation. However, the result does not confirm the status of λ-calculus and SKI-calculus as the king and queen of rewriting systems. Rather, it further emphasises their differences, and illustrates their weakness relative to any intensionally complete calculi such as SF-calculus, λSF-calculus and T-calculus.

7.0.0.1 Acknowledgments

Thanks go to the anonymous referees who suggested some of the related work.

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.

2 Hendrik Pieter Barendregt, J Richard Kennaway, Jan Willem Klop, and M Ronan Sleep. Needed reduction and spine strategies for the lambda calculus. *Information and Computation*, 75(3):191–231, 1987.

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

4 Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, César Muñoz, Chetan Murthy, Catherine Parent, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. The Coq Proof Assistant Reference Manual : Version 6.1. Research Report RT-0203, INRIA, May 1997. Projet COQ.

5 Pierre-Louis Curien. Categorical combinators. *Information and Control*, 69(1):188 – 254, 1986.

6 Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Birkhäuser, Boston, USA, 1993. Second edition.

7 Haskell Brooks Curry, Robert Feys, William Craig, and William Craig. *Combinatory logic*. North-Holland Amsterdam, 1972.

8 H.B. Curry and R. Feys. *Combinatory Logic*. Number v. 1 in Combinatory Logic. North-Holland Publishing Company, 1958.

9 Vincent Danos and Laurent Regnier. Head linear reduction. *Unpublished*, 2004.

10 N. de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat.*, 34:381–392, 1972.

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

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

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

14 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).

15 Barry Jay. A combinatory account of substitution, March 2017. `https://github.com/Barry-Jay/SF`.

16 Barry Jay. Repository of proofs for l-calculus in Coq, March 2017. `https://github.com/Barry-Jay/L-and-T-calculi`.

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

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

19 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.

20 Barry Jay and Jose Vergara. Conflicting accounts of λ-definability. *Journal of Logical and Algebraic Methods in Programming*, 87:1 – 3, 2017.

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

22 Pierre-Marie Pédrot and Alexis Saurin. Classical by-need. In Peter Thiemann, editor, *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, pages 616–643. Springer, 2016.

23 D. Pigozzi and A. Salibra. Lambda abstraction algebras: Coordinatizing models of lambda calculus. *Fundamenta Informaticae*, 33(2):149–200, 1998.

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