A Combinatory Account of Substitution

Barry Jay
University of Technology Sydney
Barry.Jay@uts.edu.au

***Abstract*—There is a translation from a confluent λ-calculus to a combinatory calculus that preserves the equality determined by the reduction relation. The combinatory calculus is SF-calculus, which is able to define intensional functions, such as the substitution function used by the lambda-calculus being translated, which requires the target of substitution to be a weak head normal form. The normality restrictions are not of practical significance, since combinators are able to support fixpoint functions that are also normal forms. Proofs of all named results have been verified in Coq.**

I. Introduction

The main goal of combinatory logic is to give an equational account of λ-abstraction within a combinatory calculus [1], [2]. The traditional account is not completely satisfactory: although it supports β-reduction, it does not support the ξ-rule, in that terms which are equal (i.e. share a reduct) need not have equal abstractions. This paper solves the problem by showing how to translate a λ-calculus to a combinatory calculus in a way that preserves equality. The proof has been verified in Coq.

The combinatory calculus is SF-calculus [3], [4]. It is strictly more expressive than the traditional combinatory logic, SKI-calculus, because it supports *intensional* functions on normal forms, such as their equality or Gödelization, by using F to *factorise* compounds (partially applied operators). Recent work has shown how to factorise λ-abstractions, too, in λSF-calculus [5]. Hence λSF-calculus is strictly more expressive than pure λ-calculus.

These results show that neither traditional combinatory logic, nor pure λ-calculus is adequate as a foundation for computing. Of course, this contradicts the traditional account of the foundations, but close reading of the original papers and books shows how the misunderstanding has arisen [6].

Whether or not you are ready to accept these revisions is not actually crucial to your acceptance of this paper, whose definitions and theorems can be appreciated without adopting any particular position on foundations. Rather, an awareness of intensionality will help understand how the solutions were found.

Note that although λSF-calculus supports embeddings of both pure λ-calculus and of SF-calculus, it is no closer than pure λ-calculus to giving an equational account of abstraction. To achieve this in SF-calculus, it is sufficient to present substitution as an intensional function in the sense used by SF-calculus, e.g. as a pattern-matching function, with cases for operators, compounds, variables and abstractions. This will not include a case for applications that are not compounds, as these cannot be factorised, at least until further reduced. This is more constrained than the traditional substitution function, which can act on arbitrary applications. The gap is eliminated by developing a new λ-calculus, the *lifting λ-calculus*, in which substitution is restricted to weak head normal forms. In turn, this introduces the challenge of representing recursive functions normal forms.

A. Non-Termination

Consider the substitution of some combinator N for some variable x in some application Ω which reduces to itself, e.g. the λ-term $(\lambda x.xx)(\lambda x.xx)$ or the combinator $(SII)(SII)$. Since x does not appear free in Ω the result is Ω itself. However, Ω cannot be factorised by F since factorisation is restricted to partially applied operators, while Ω is fully applied.

The traditional view is that non-termination is unavoidable. First, the Halting Problem shows that there is no general procedure for deciding which terms have normal forms. Second, general recursive functions are defined by fixpoints which, as represented in λ-calculus, do not have normal forms. However, combinators give much finer control over reduction than λ-calculus.

In particular, the traditional account of abstraction with respect to a variable x in combinatory logic, that maps M to $\lambda^* x.M$, always produces normal forms. For example, given terms M and N then

$$\lambda^* x.MNx$$

is always a normal form, even if MN is a redex. Thus, we may define

$$\mathsf{a}[M,N] = S(S(KM)(KN))I$$

to be an optimised version of this, so that $\mathsf{a}[M,N]$ is *not* a redex, but $\mathsf{a}[M,N]P$ reduces to MNP for any term P.

By using $\mathsf{a}[-,-]$, fixpoint functions can be represented by normal forms that do not reduce until given an argument. It follows that intensional functions can be applied to fixpoints, including those for general recursive functions, and the fixpoint that will define substitution. Since terms without normal form are no longer required for any practical purpose, we are entitled to disregard them. Rather, it will be enough to focus on substitution into the λ-terms that are *weak head normal forms*, i.e. the variables, applications of variables, and abstractions, but not the applications of abstractions.

Now let us consider substitution into the different term forms. Operators are easily handled.

B. Applications

The first variation on λ-calculus is the *piecewise λ-calculus*, in which β-reduction

$$(\beta) \quad (\lambda x.M)N \longrightarrow \{N/x\}M$$

is restricted to require M to be a weak head normal form. This allows reduction to proceed without invoking any meta-function for substitution, by giving separate β-rules for the various cases, such as

$$(\lambda x.xM_1)N \longrightarrow N((\lambda x.M_1)N) \ .$$

Here, the substitution of N for x in xM_1 is performed piecewise: the substitution of N for x in x, to get N, and the application of a *new* abstraction $\lambda x.M_1$ to N. Further β-reduction is possible if M_1 is a weak head normal form.

Similar rules are required if M is of the form $xM_1 \dots M_k$. To avoid having countable families of rules requires a side-condition that tests for M being headed by a variable, i.e. being *active*. The only other side conditions concern variable equality, and variable occurrence.

C. Variables

Variables will be represented using de Bruijn indices. This will allow variables to be represented by combinators. In the standard account, the terms are given by the BNF

$$M, N := i \mid MN \mid \lambda M$$

where i is a natural number (or de Bruijn index) that denotes a variable, MN is the application of M to N as usual, and λM is the abstraction of M with respect to the variable 0. That is, the variable i is bound by the ith enclosing λ. It follows that when substituting N (for 0) into an abstraction λM then the free variables of N are enclosed within another λ and so must be *lifted*. Like substituting, lifting is usually regarded as a meta-function, but we are trying to eliminate them, not introduce them. Rather, it will be represented within the calculus as follows.

The *lifting* λ-calculus adds to the de Bruijn terms a new term form

$$\mathsf{lift}[M, k]$$

in which the free variables of M whose index is at least k are to be lifted by one. For example, $\mathsf{lift}[\lambda\, 0\, 1\, 2, 0]$ reduces to $\lambda(\mathsf{lift}[0\, 1\, 2, 1])$ and then to $\lambda\, 0\, 2\, 3$. Like the β-reduction rules, reduction of lifting is restricted to weak head normal forms.

Now reduction has two sorts of side-conditions. Those concerning variable occurrences in the piecewise λ-calculus are replaced by comparisons of natural numbers; these will be easily represented in SF-calculus. The remaining side-conditions concern active terms, of the form $iM_1 \dots M_k$ which will now be addressed.

The representation of the variable i in SF-calculus will be

$$\mathsf{var}[S^i S, \mathsf{nil}]$$

where $S^i S$ represents the numeral i and $\mathsf{nil} = S$ represents an empty list of arguments. This is a normal form but when applied to the representations $N_1, \dots N_k$ of $M_1, \dots, M_k$ it reduces to

$$\mathsf{var}[S^i S, L]$$

where $L = \mathsf{snoc}\ (\dots(\mathsf{snoc}\ \mathsf{nil}\ N_1)\dots)N_k$ represents the list of arguments, with snoc also represented by S. In this manner, the pattern $\mathsf{var}[-,-]$ can be used to detect applications of variables without any side-condition.

D. Abstraction

The abstraction of M with respect to the variable indexed by 0 in SF-calculus will be represented by

$$\mathsf{abs}\ M = \mathsf{a}[\mathsf{subst}, M]$$

where subst is the term for substitution. In turn, substitution will be defined using a combinator lift_rec for lifting, which is defined using abstraction. This apparent circularity will be broken by parametrising the definition of lift_rec by a variable s which is to represent substitution. Then the pattern for an abstraction could then be given by

$$\mathsf{lam}[s, y] = \mathsf{a}[s, y] \ .$$

However, this approach would require that we support *dynamic patterns* [7], [8] which may contain a mix of free variables s and bound variables y. Instead, it is simpler to stick with *static patterns* of the form

$$\mathsf{lam}[x, y]$$

and then check to see if x equals s afterwards by an application of the *generic equality function* equal which decides equality of closed normal forms.

That done, substitution is defined by a fixpoint whose body contains a call to lift_rec which must decide equality with subst. In brief, subst is required to recognise copies of itself! This peculiar feat is possible because subst will be a combinator in normal form, whose equality with other such is an intensional function.

E. Translation

The rest of the development of the representation $[\![-]\!]$ of lifting λ-calculus within SF-calculus consists of careful development of the terms for lifting and substituting, as pattern-matching functions, especially to check that the patterns for variables, abstractions and applications do not overlap, e.g. that the representation of an abstraction cannot match with the pattern for a variable. Fortunately, all of the proofs have been verified in Coq, which confirms that the translation preserves equality (having a common reduct).

Theorem 1 (lambda_to_SF_preserves_equality): For all terms M and N of lifting λ-calculus, if $M = N$ then $[\![M]\!] = [\![N]\!]$ in SF-calculus.

F. Abstraction Within SF-Calculus

This theorem shows the extent to which SF-calculus supports λ-calculus. That is, it supports β-reduction in the same way that lifting λ-calculus and piecewise calculus do. However, the original goal was to understand abstraction within the combinatory setting, to expand the system from the consideration of combinators to the corresponding term language, which also includes variables $x, y, z, \ldots$ and not just de Bruijn representations $\mathsf{var}[i, \mathsf{nil}]$ of variables. The solution is to define abstraction of t with respect to x by

$$\lambda^\dagger x.t = \mathsf{abs}\ \{\mathsf{var}[S, \mathsf{nil}]/x\}(\mathsf{lift\_rec}\ t\ S)\}$$

where S represents 0 as usual and the meta-function of substitution $\{-/x\}$ is defined in the usual way. That is, lift all representations of variables in t, replace x by the representation of the variable indexed by 0, and abstract as before. $\lambda^\dagger x.t$ satisfies the β-rule and the ξ-rule as required. Any limitations of this approach arise from the use of subst. As well as the usual constraints arising from the restriction to partially applied operators, substitution for x will become stuck when applied to a different term variable y or z. Of course, this is easily resolved in practice by converting all variables to de Bruijn representations before abstracting. It follows that Theorem 1 is the key result.

G. Contents of the Paper

The sections of the paper are as follows: Section I is the introduction; Section II describes the various λ-calculi employed; Section III describes the combinatory calculi to be used; Section IV describes the representation of variables; Section V describes the representation of lifting; Section VI describes the representation of substitution; Section VII translates the lift λ-calculus to SF-calculus; Section VIII describes the verification in Coq; Section IX considers related work; Section X discusses future work; and Section XI draws conclusions.

II. λ-Calculus

A. Pure λ-Calculus

This subsection recalls some basic facts about λ-calculus, as found in, say, Barendregt's book [9], and rewriting [10], and introduces some terminology that will be used later.

Pure λ-calculus has terms given by the BNF

$$M, N := x \mid MN \mid \lambda x.M$$

consisting of *variables* $x, y, z, \ldots$ and *applications* MN and *abstractions* $\lambda x.M$ whose *body* is M and whose bound variable is x. There is one reduction rule

$$(\beta) \quad (\lambda x.M)N \longrightarrow \{N/x\}M\ .$$

where $\{N/x\}M$ is the *substitution* of N for x in M, as defined by a meta-function, i.e. a function on the terms that is *not* definable within the calculus. The details of the definition of substitution are delicate, since there is the need to avoid variable capture, but well known. The congruence generated by $\longrightarrow$ is the *one-step* reduction relation denoted by $\longrightarrow^1$. This may also be called the *pure* reduction relation. The congruence can be specified by giving rules for each term form: that for abstractions is sometimes called the ξ-rule:

$$(\xi) \quad \frac{M \longrightarrow N}{\lambda x.M \longrightarrow \lambda x.N}\ .$$

The reflexive, transitive closure of $\longrightarrow^1$ is the *multi-step* reduction relation, denoted by $\longrightarrow^*$. Term *equality*, denoted by $=$ is the symmetric closure of $\longrightarrow^*$. Similar conventions will apply to the other reduction systems appearing in this paper.

Pure λ-calculus is *confluent*, meaning that if $M \longrightarrow^* N$ and $M \longrightarrow^* P$ then there is a term Q such that $N \longrightarrow^* Q$ and $P \longrightarrow^* Q$. When reduction is confluent, two terms M and N are equal if they have a common reduct R, so that $M \longrightarrow^* R$ and $N \longrightarrow^* R$.

Classify the terms as being either active or passive by defining their status as follows:

$$\begin{aligned} \mathsf{status}\ x &= \mathsf{active} \\ \mathsf{status}\ (MN) &= \mathsf{status}\ M \\ \mathsf{status}\ (\lambda x.M) &= \mathsf{passive}\ . \end{aligned}$$

Then we may define the *normal forms* as follows:

- all variables are normal;
- MN is normal if M and N are, and MN is active; and
- $\lambda x.M$ is normal if M is.

It is easy to prove that a term is irreducible with respect to pure reduction if and only if it is normal.

λ-calculus supports recursion through *fixpoint functions*, as follows. Define

$$\omega = \lambda x.\lambda f.f(xxf)$$

and

$$\mathsf{fix} = \omega\omega\ .$$

It follows that

$$\mathsf{fix}\ f = \omega\omega f \longrightarrow^* f(\omega\omega f) = f(\mathsf{fix}\ f)\ .$$

An unfortunate side-effect of this account is that fix, and hence recursive functions, are not strongly normalising since

$$\omega\omega \longrightarrow^* \lambda f.f(\omega\omega f) \longrightarrow^* \ldots$$

In practice, an evaluation strategy is required to avoid this.

For example, define the *weak head normal forms* as follows:

- all variables are weak head normal;
- MN is weak head normal if MN is active; and
- $\lambda x.M$ is weak head normal.

That is, the weak head normal forms are the active terms and the abstractions, but not applications of abstractions. A common evaluation strategy is to avoid reduction under the λ so that, for example, reduction of $\omega\omega$ stops after one step. This is still one step too many, but at least reduction terminates. The price for adopting such as strategy is that reduction is no longer confluent, so that one cannot freely replace a sub-term by another that is equal to it.

$$
\begin{array}{rcll}
(\lambda x.x\ M_1 \ldots M_k)N & \longrightarrow & N\ ((\lambda x.M_1)N) \ldots ((\lambda x.M_k)N) & \\
(\lambda x.y\ M_1 \ldots M_k)N & \longrightarrow & y\ ((\lambda x.M_1)N) \ldots ((\lambda x.M_k)N) & (y \neq x) \\
(\lambda x.\lambda y.M)N & \longrightarrow & \lambda y.(\lambda x.M)N & (y \notin \mathsf{fv}(N))\ .
\end{array}
$$

Fig. 1. Informal piecewise reduction

$$
\begin{array}{rcll}
(\lambda x.x)N & \longrightarrow & N & \\
(\lambda x.y)N & \longrightarrow & y & (y \neq x) \\
(\lambda x.M_1M_2)N & \longrightarrow & (\lambda x.M_1)N((\lambda x.M_2)N) & (\mathsf{status}\ (M_1M_2) = \mathsf{active}) \\
(\lambda x.\lambda y.M)N & \longrightarrow & \lambda y.(\lambda x.M)N & (y \notin \mathsf{fv}(N))\ .
\end{array}
$$

Fig. 2. Formal piecewise reduction

B. Piecewise λ-calculus

Here is a variant of the pure λ-calculus that constrains β-reduction to weak head normal forms while maintaining confluence.

Informally, the piecewise reduction rules are given in Figure 1. Each contractum (left-hand side) is β-reducible, with the bodies of the abstractions covering all possible weak head normal forms. The reducts (right-hand sides) are *not* expressed using a substitution function. Rather, they perform one step of the substitution, with any remaining work described by creating abstractions of sub-terms. For example $(\lambda x.x)N$ reduces to N as usual and

$$(\lambda x.xy)N \longrightarrow^1 N((\lambda x.y)N) \longrightarrow^1 Ny\ .$$

However, if M does not reduce to a weak head normal form then $(\lambda x.M)N$ will never reduce to a contractum of piecewise reduction, even though it is β-reducible. In practice, this is no loss, since $\{N/x\}M$ will not have a weak head normal form either.

As a calculus, the rules that are indexed by the number k of applications should be seen as countable families of reduction rules, which is distracting. To avoid this, we can replace the indexing with a side-condition for being an active term. The formal account of the piecewise reduction rules is given in Figure 2.

This is equivalent to the informal system, in that one reduction of the informal system can be represented by a sequence of reductions in the formal system.

A curious feature of the reduction relation is that it is *not* stable under substitution: substituting Ω for y in $(\lambda x.y)N$ converts a redex to a non-redex if Ω does not have a weak head normal form. Nevertheless, the reduction rules do not overlap, and so reduction is confluent. Also, since all normal forms are also weak head normal forms, the irreducible terms under piecewise reduction are the same as those under the usual β-reduction. These results are enough to establish the suitability of piecewise reduction as a basis for computation. It seems likely that the resulting observational equivalence is the same as that arising from the standard β-reduction. However, we shall not bother to formalise these results, but press on to consider our main target.

The side conditions to piecewise reduction are simpler to work with than in pure reduction, as substitution has been replaced by the test for being active. However, our ultimate goal is to eliminate all side conditions. The next step is to eliminate the side conditions involving variables and their scope.

C. de Bruijn Indices

The side-conditions concerning variable names can be eliminated by representing variables by de Bruijn indices. These are natural numbers, which are represented by zero 0 and successor S or the meta-variables i, j, k, m, n etc. Abstraction always binds variable 0 so there is no need to name the bound variable. Variable renaming is handled by *lifting*, as expressed by the term form $\mathsf{lift}\ M\ k$ which increases by one all indices that are at least k, (with due regard to scope). In traditional accounts, lifting may increase the index by more than one, but this is not essential. Also, it is usually given as a meta-function, but since we are trying to eliminate meta-functions, not add them, let us include lifting as a new term form.

The terms of the *lift* λ*-calculus* are thus given by

$$MN := i \mid MN \mid \lambda M \mid \mathsf{lift}[M, k]$$

where i and k denote natural numbers, called indices. As a term form, i is a variable. *Status* is defined as before, on the understanding that all lifted terms are passive. The reduction rules of lift λ-calculus are given in Figure 3.

The first three rules are much as in piecewise reduction. The fourth rule is a little harder to follow. First, note that N has been brought within the scope of the outermost λ. To avoid variable capture, all its variables are lifted. Second, to maintain the alignment between variables and their binders, the variables in M indexed by 0 and 1 must be swapped, using

$$\mathsf{swap}[M] = (\lambda\, \mathsf{lift}[M, 2])\ 1\ .$$

Now the fourth reduction rule can be written

$$(\lambda\lambda M)N \longrightarrow \lambda(\lambda\, \mathsf{swap}[M])\mathsf{lift}[N, 0]))\ .$$

The rules for lifting are as expected, except perhaps for the restriction to applications that are active. This has not made the side conditions any worse, and will make it easier to later represent lifting as a combinator.

$$
\begin{array}{rcll}
(\lambda 0)N & \longrightarrow & N & \\
(\lambda(Si))N & \longrightarrow & i & \\
(\lambda M_1 M_2)N & \longrightarrow & ((\lambda M_1)N)((\lambda M_2)N) & (\mathsf{status}\ (M_1 M_2) = \mathsf{active}) \\
(\lambda\lambda M)N & \longrightarrow & \lambda((\lambda(\lambda\, \mathsf{lift}[M,2])1))(\mathsf{lift}[N,0])) & \\
\mathsf{lift}[i,k] & \longrightarrow & i & (\text{if } i < k) \\
\mathsf{lift}[i,k] & \longrightarrow & Si & (\text{if } i \geq k) \\
\mathsf{lift}[M_1\ M_2,k] & \longrightarrow & (\mathsf{lift}[M_1,k]\ (\mathsf{lift}[M_2,k]) & (\mathsf{status}\ (M_1 M_2) = \mathsf{active}) \\
\mathsf{lift}[\lambda M,k] & \longrightarrow & \lambda \mathsf{lift}[M,Sk]\ . &
\end{array}
$$

Fig. 3. Reduction in lift λ-calculus

Confluence and progress results for lift λ-calculus have been formalised in Coq.

Theorem 2 (lift_confluence): Reduction in lift λ-calculus is confluent.

Theorem 3 (lift_progress): Every term of lift λ-calculus is either a normal form (in the sense of pure λ-calculus) or can be reduced.

III. Combinators

Combinators [1], [2] are built from *operators* (meta-variable O) and *applications*, as described by the BNF

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

Define a *program* to be a combinator that is irreducible.

Sometimes, it is convenient to add *variables* (meta-variables $x, y, z, \ldots$) to produce *combinatory terms* given by

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

Substitution for variables is defined in the obvious manner, since there is no binding operation to worry about.

A. Traditional Combinatory Logic

This section recalls some basic facts about traditional combinatory logic. Its operators are S, K and I. They support the following reduction rules

$$
\begin{array}{rcl}
SMNP & \longrightarrow & MP(NP) \\
KMN & \longrightarrow & M \\
IM & \longrightarrow & M\ .
\end{array}
$$

The resulting calculus is called SKI-calculus, or *combinatory logic*. All three reduction rules are *extensional*, in the sense that there is no need to explore the internal structure of the arguments: S makes a copy of the final argument; K deletes it; and I does nothing.

Since λ-calculus supports extensionality in a general manner, it is no surprise that the operators S, K and I are definable within λ-calculus, by the terms

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

Conversely, λ-abstraction of a combinatory term t with respect to a variable x can be defined as the $\lambda^* x.t$ given by

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

This account of abstraction supports β-reduction since

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

For example, define $\mathsf{re\_order}_0$ by

$$
\begin{array}{rcl}
\mathsf{re\_order}_0 & = & \lambda^* x.\lambda^* f.f\ x \\
& = & \lambda^* x.SI(Kx) \\
& = & S(S(KS)(KI))(S(KK)I)\ .
\end{array}
$$

However, the well known weakness of this account is that λ^* does not preserve reduction. For example $\lambda^* x.Ix = S(KI)I$ but if Ix is first reduced to x then we have $\lambda^* x.x = I$. Thus, λ^* does not preserve equality: it is defined on the syntax of combinators, without respecting their semantics.

Considerable effort has been spent on adding rules that might repair this break of confluence, or optimise implementations. The example above is handled by adding the reduction rule

$$S(Kt)I \longrightarrow t\ .$$

Another example is given by

$$S(K\ s)(K\ t) \longrightarrow K(s\ t)\ .$$

In this manner, we can replace $\mathsf{re\_order}_0$ with

$$\mathsf{re\_order} = S(K(SI))K\ .$$

However, the general situation is difficult.

Here is another example, replacing λ by λ^* in the definition of ω in Section II yields

$$\lambda^* x.\lambda^* f.f(xxf) = \lambda^* x.SI(S(S(Kx)(Kx))I)$$

which is much bigger than needed. A little optimisation (by η-contraction) can replace this with

$$\lambda^* x.SI(xx)$$

and then

$$\omega^* = S(K(SI))(SII)$$

$$\begin{array}{rcl} SMNP & \longrightarrow & MP(NP) \\ FOMN & \longrightarrow & M \\ F(PQ)MN & \longrightarrow & NPQ \quad (PQ \text{ is a compound}). \end{array}$$

Fig. 4. Reduction in SF-calculus

so that the analogue of fix becomes

$$(S(K(SI))(SII))(S(K(SI))(SII))$$

As in λ-calculus, this term does not have a normal form. However, this can be repaired by exploiting the combinatorial ability to exert fine control over reduction.

B. Controlling Reduction Order

This subsection shows how to control reduction order, so that we can define fixpoint functions that are normal forms.

Suppose that our goal is to delay the application of M to N until a second argument, say P, is given. The natural choice is

$$\lambda^*x.MNx = S(\lambda^*x.MN)I = S(S(\lambda^*x.M)(\lambda^*x.N))I \ .$$

Since x may be chosen fresh, we may replace $\lambda^*x.M$ with KM and $\lambda^*x.N$ with KN to get

$$\mathsf{a}[M,N] = S(S(KM)(KN))I \ .$$

Now $\mathsf{a}[M,N]$ is not a redex, but its application to some P reduces to MNP as desired.

In the same spirit, we can replace the usual fixpoint construction with one that does not introduce redexes, as follows. Define ω by

$$\omega = \lambda^*x.\lambda^*f.f(\mathsf{a}[\mathsf{a}[x,x],f])$$

and then

$$\mathsf{fix}[M] = \mathsf{a}[\mathsf{a}[\omega,\omega],M] \ .$$

This is a normal form if M is. Further, when $\mathsf{fix}[M]$ is applied to some argument N then

$$\begin{array}{rcl} \mathsf{fix}[M]\ N & \longrightarrow^* & \mathsf{a}[\omega,\omega]\ M\ N \\ & \longrightarrow^* & \omega\ \omega\ M\ N \\ & \longrightarrow^* & M(\mathsf{fix}[M])N \end{array}$$

as expected for a fixpoint function.

C. SF-calculus

This subsection recalls the combinatory SF-calculus [3]. Its reduction rules are given in Figure 4. The operator S is as above, but F is intensional, in that it branches according to the nature of its first argument. If this is an operator O then the second argument is used, by the rule

$$FOMN \longrightarrow M \ .$$

If the first argument is a *compound* PQ then the third argument is used, according to the rule

$$F(PQ)MN \longrightarrow NPQ \ .$$

Not every application is a compound. Rather, the *compounds* are the partially applied operators. In this setting, the operators S and F are both ternary, so that the compounds take one of the following four forms SM, SMN, FM and FMN. This *factorisation* by F exposes the components of a compound, but reduction remains confluent, since compounds are never redexes.

Theorem 4 (SF_confluence): SF-reduction is confluent.

The *normal forms* of SF-calculus are defined to be the operators, and partial applications of operators to normal forms.

Theorem 5 (SF_progress): Every SF-combinator is either a normal form or can be reduced.

The operators K and I are definable as SF-combinators by

$$\begin{array}{rcl} K & = & FF \\ I & = & SKK \ . \end{array}$$

Since K and I are definable, the constructions λ^*x and $\mathsf{a}[-,-]$ and $\mathsf{fix}[-]$ are also definable, without further ado.

On the other hand, F cannot be defined in terms of S, K and I. To see this, note that F can be used to distinguish the combinators SKS and SKF. As both represent the identity function, they cannot be distinguished using the extensional operators S, K and I. See [3] for details.

D. Natural Numbers

The traditional representation of natural numbers as combinators is to first represent them as λ-abstractions and then convert the abstractions to combinators using λ^*. Since factorisation is now supported there is an easier method.

Represent the natural numbers by identifying zero with S and successor with S, so that 2 is represented by $S(SS)$ and i is represented by S^iS where S^i applies S a total of i times. Then the primitive recursive functions are easily defined. For example, the *zero-test* is given by

$$\mathsf{is\_zero} = \lambda^*x.FxK(K(K(KI)))$$

where K and KI are the boolean values for true and false. Again, the predecessor function, which traditionally is a linear-time function, is here given by

$$\mathsf{pred} = \lambda^*x.Fxx(KI) \ .$$

$$
\begin{array}{rcl}
x \Rightarrow s & = & \lambda^* x.K\ s \\
O \Rightarrow s & = & \lambda^* x.Fx(\mathsf{eqop}\ O\ x\ (Ks)\ (\mathsf{swap}\ x))\ (K(K(\mathsf{swap}\ x))) \\
(p\ q) \Rightarrow s & = & \lambda^* x.FxI(\lambda^* y.\lambda^* z.(p \Rightarrow (q \Rightarrow K(Ks)))y(K(K(KI)))z(KI))(\mathsf{swap}\ x)\ .
\end{array}
$$

Fig. 5. Cases

In the sequel, we will have need of various arithmetic functions to manage de Bruijn indices. For example, define

$$\mathsf{relocate} = \lambda^* n.\lambda^* i.\mathsf{lte}\ n\ i\ (S\ i)\ i$$

where lte checks for being less-than-or-equal.

E. Equality

The equality of combinators in normal form is defined as follows. Since S and F have different extensional behaviours, there is a combinator eqop which decides equality of operators. Then we may define equal by

$$
\begin{array}{l}
\mathsf{fix}[\lambda^* e.\lambda^* x.\lambda^* y. \\
\quad F\ x\ (F\ y\ (\mathsf{eqop}\ x\ y))\ (\lambda^* x_1.\lambda^* x_2. \\
\qquad F\ y\ (KI)\ (\lambda^* y_1.\lambda^* y_2.e\ x_1\ x_2\ (e\ y_1\ y_2)\ (KI))) \\
\ ]
\end{array}
$$

Theorem 6 (equal_programs): If M is a program then $\mathsf{equal}\ M\ M \longrightarrow^* K$.

Theorem 7 (unequal_programs): If M and N are unequal programs then $\mathsf{equal}\ M\ N \longrightarrow^* KI$.

For example, $\mathsf{equal}\ (SKS)\ (SKK) \longrightarrow^* KI$ even though both SKS and SKK represent the identity function.

In a similar manner, we can define the Gödel number of a combinator, as in [5].

F. Pattern-Matching

More generally, we may define *pattern-matching* as follows. Given terms p and s (i.e. which may contain variables), the *case* $p \Rightarrow s$ will be defined so that, given an argument u, it tests whether u matches against p. More precisely, cases are defined to take two arguments, namely the u above and a *default* function r, which will be applied to u if matching fails. For example, we will have

$$
\begin{array}{rcl}
(x \Rightarrow s)ur & = & (\lambda^* x.Ks)ur \longrightarrow^* \{u/x\}s \\
(O \Rightarrow s)Or & \longrightarrow^* & s \\
(p \Rightarrow s)ur & \longrightarrow^* & \mathsf{re\_order}\ u\ r \\
 & \longrightarrow^* & r\ u \quad (\text{if } u \text{ does not match } p.)
\end{array}
$$

Cases are defined by induction on the structure of the *pattern* p, in Figure 5.

The most delicate case is that for applicative patterns, since matching with an argument may fail for three different sorts of arguments: operators; applications whose left component does not match; or by applications whose right components does not match. If the order of applications of $p \Rightarrow s$ to u and r is reversed then r will be needlessly duplicated. By placing u first, the default function r is not brought into play until the status of matching is decided.

Although the definition allows any term p to be a pattern, matching cannot succeed unless p is a normal form. All patterns appearing in this paper are normal forms.

It commonly happens that pattern-matching functions are constructed from a sequence of cases, with $p_1 \Rightarrow s_1$ followed by $p_2 \Rightarrow s_2$ through to $p_n \Rightarrow s_n$. This can be represented as a single case $p_1 \Rightarrow s$ with a default function built from the other cases, as follows.

Define the *extension* of the case $p \Rightarrow s$ by the default function r to be

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

It follows that

$$
\begin{array}{rcl}
(p \Rightarrow s \mid r)u & = & S(p \Rightarrow s)(Kr)u \\
 & \longrightarrow^1 & (p \Rightarrow s)u(Kru) \\
 & \longrightarrow^1 & (p \Rightarrow s)ur
\end{array}
$$

as desired. By making $\mid$ right-associative, we may then write the pattern-matching function with cases as above and ultimate default r as

$$
\begin{array}{l}
p_1 \Rightarrow s_1 \\
\mid p_2 \Rightarrow s_2 \\
\mid r\ .
\end{array}
$$

The process of matching, and of match failure, to produce substitutions σ and their application, can be formalised and lead to the following propositions.

Proposition 1 (extensions_by_matching): For all SF-terms P and N, if N matches against P to yield a substitution σ then for all terms M and R we have

$$(P \Rightarrow M \mid R)N \longrightarrow^* \sigma M\ .$$

Proposition 2 (extensions_by_matchfail): For all SF-terms P and N, if N fails to match against P then for all terms M and R we have

$$(P \Rightarrow M \mid R)N \longrightarrow^* RN\ .$$

Pattern-matching will be used heavily in the account of substitution. For example, we can match with arbitrary fixpoints using the function

$$(\mathsf{fix}[f] \Rightarrow K \mid K(KI)$$

since $\mathsf{fix}[f]$ is a normal form.

IV. Variables

The representation of variables uses two key ideas. First, variables are indexed by natural numbers, by de Bruijn indices. Second, applications of variables will reduce, to collect all of the arguments into a list L. Thus, the application of the variable with index i to arguments represented by L will reduce to $\mathsf{var}[S^i S, L]$, where var is to be defined. Substitution of N for the ith variable will then trigger the application of N to (the substitutes of) the entries of L.

A. Lists of Arguments

The *nil list* will be represented by S. The addition of an entry M to the right-hand end of a list L is given by SLM. This operation is sometimes called *snoc* (cons written backwards):

$$\begin{aligned} \mathsf{nil} &= S \\ \mathsf{snoc} &= S\,. \end{aligned}$$

The application of a combinator N to the entries of a list L is defined by

$$\mathsf{fold\_app} = \mathsf{fix}[\lambda^* f.\lambda^* n.\lambda^* l.Fln(\lambda^* x.\lambda^* y. FxS(K(\lambda^* z.fnzy)))]\,.$$

Creation of argument lists from a sequence of applications is achieved by the combinator V, defined by

$$V = \mathsf{fix}[\lambda^* v.\lambda^* i.\lambda^* x.\lambda^* y.\mathsf{a}[\mathsf{a}[v,i],Sxy]].$$

Note how $\mathsf{a}[-,-]$ is used to delay reduction until three arguments are supplied. Now define

$$\mathsf{var}[i,L] = \mathsf{a}[\mathsf{a}[V,i],L]\,.$$

In particular, the ith variable is given by

$$\mathsf{var}[S^i S, \mathsf{nil}]\,.$$

Now $\mathsf{var}[i,L]$ is normal if i and L are. Its basic property is given by the following lemma.

Lemma 1 (var_check): For all terms i, L and M we have

$$\mathsf{var}[i,L]M \longrightarrow^* \mathsf{var}[i,SLM]\,.$$

Now the pattern $\mathsf{var}[x,y]$ can be used to detect variables.

V. Lifting

Lifting of variables will be required when managing scope. In the de Bruijn approach, the ith variable is bound by the ith enclosing λ. For example, S, K and I are represented by $\lambda\lambda\lambda\,2\,0(1\,0)$ and $\lambda\lambda 1$ and $\lambda 0$ respectively. As the number of enclosing abstractions rises, the enclosed indices must also rise, by *lifting*, as specified by

$$\begin{aligned} \mathsf{lift\_rec}\ (\mathsf{var}[i,M])\ n &= \mathsf{var}[\mathsf{relocate}\ n\ i, \mathsf{lift\_rec}\ M\ n] \\ \mathsf{lift\_rec}\ (\lambda\ M)\ n &= \lambda\ (\mathsf{lift\_rec}\ M\ (S\ n)) \\ \mathsf{lift\_rec}\ (M\ N)\ n &= (\mathsf{lift\_rec}\ M\ n)\ (\mathsf{lift\_rec}\ N\ n)\,. \end{aligned}$$

However, we cannot define λ until lift_rec is defined, so let us parametrise lift_rec with another argument s, representing subst_rec and then define

$$\mathsf{lam}[s,M] = \mathsf{a}[s,M]$$

to get

$$\begin{aligned} \mathsf{lift\_rec}\ s\ (\mathsf{var}[i,M])\ n &= \mathsf{var}[\mathsf{relocate}\ n\ i, \mathsf{lift\_rec}\ s\ M\ n] \\ \mathsf{lift\_rec}\ s\ \mathsf{lam}[s,M]\ n &= \mathsf{lam}[s, \mathsf{lift\_rec}\ s\ M\ (S\ n)] \\ \mathsf{lift\_rec}\ s\ (M\ N)\ n &= (\mathsf{lift\_rec}\ s\ M\ n)(\mathsf{lift\_rec}\ s\ N\ n). \end{aligned}$$

These equations will underpin the pattern-matching definition of lift_rec. The test for being $\mathsf{lam}[s,M]$ in the third equation will reduce to a test for being equal to s. The general definition of lift_rec is thus $\lambda^* s.\mathsf{lift\_rec\_fn}$ where lift_rec_fn is defined by

$$\begin{aligned} &\mathsf{fix}[\lambda^* f. \\ &\quad \mathsf{var}[i,x] \Rightarrow \lambda^* n.\mathsf{var}[\mathsf{relocate}\ n\ i, fsxn] \\ &\quad \mid \mathsf{lam}[x,y] \Rightarrow \mathsf{equal}\ s\ x\ (\lambda^* n.\mathsf{lam}[x, fs(Sn)y]) \\ &\qquad\qquad\qquad\qquad \lambda^* n.\mathsf{lam}[fsxn, fsyn] \\ &\quad \mid z \Rightarrow Fz(Kz)\lambda^* x.\lambda^* y.\lambda^* n.(fsxn)(fsyn)]\,. \end{aligned}$$

Some of the properties of lift_rec are illustrated by the following propositions.

Proposition 3 (lift_rec_comb_op1): For all s, O and n,

$$\mathsf{lift\_rec}\ s\ O\ n \longrightarrow^* O\,.$$

Proposition 4 (lift_rec_comb_var): For all s, i, M and n

$$\mathsf{lift\_rec}\ s\ \mathsf{var}[i,M]\ n \longrightarrow^* \mathsf{var}[\mathsf{relocate}\ n\ i, \mathsf{lift\_rec\_fn}\ M\ n]\,.$$

The propositions for abstractions and applications require additional premises to ensure that abstractions and applications do not match against patterns for variables, etc.

Proposition 5 (lift_rec_comb_abs): For all s, n and M, if s is a program that does not match against $\mathsf{a}[V,x]$ then

$$\mathsf{lift\_rec\_comb}\ \mathsf{lam}[s,M]\ n \longrightarrow^* \mathsf{lam}[s, \mathsf{fix}[\mathsf{lift\_rec\_fn}\ M\ (Sn)]]\,.$$

Proposition 6 (lift_rec_app): For all terms s, M_1, M_2 and n, if s is a program and $M_1 M_2$ is a compound and $M_1 M_2$ does not match against $\mathsf{var}[x,y]$ or $\mathsf{lam}[x,y]$ then

$$\mathsf{lift\_rec\_fn}\ (M_1 M_2)\ n \longrightarrow^* (\mathsf{lift\_rec\_fn}\ M_1\ n)(\mathsf{lift\_rec\_fn}\ M_2\ n)\,.$$

VI. Substituting

Substitution is defined in a similar manner to lift_rec, except that the parameter s is now the fixpoint parameter.

Define

$$\mathsf{insert\_Ref\_comb}[i,N,M] = Fi(\mathsf{fold\_app}\ N\ M)(K(K(\mathsf{var}[\mathsf{pred}\ i, M])))\,.$$

and

$$\mathsf{swap\_vars}[s,M] = (\mathsf{lam}[s, \mathsf{lift\_rec}\ M\ (S^2 S)])\ \mathsf{var}[S^1 S, \mathsf{nil}]\,.$$

The definition of subst is in Figure 6. Its application to operators, variables, abstractions and compounds satisfies

$$
\begin{array}{rcl}
\mathsf{subst} & = & \mathsf{fix}[\lambda^* s. \\
& & \quad \mathsf{var}[i, x] \Rightarrow \lambda^* n.\mathsf{var}[\mathsf{insert\_Ref\_comb}[i, N, M], sxn] \\
& & \mid \mathsf{lam}[x, y] \Rightarrow \lambda^* n.\mathsf{equal}\ s\ x\ (\mathsf{lam}[x, \mathsf{lam}[x, \mathsf{swap\_vars}[x, y]]\ \mathsf{lift\_rec}\ x\ y\ 0]) \\
& & \mathsf{lam}[s\ x\ n, s\ y\ n] \\
& & \mid \lambda^* z.Fz(Kz)(\lambda^* x.\lambda^* y.S(sx)(sy))] \ .
\end{array}
$$

Fig. 6. The combinator for substitution

propositions similar to those for lift_rec, as described in Figure 7.

Then abstraction is defined by $\mathsf{abs}[M] = \mathsf{a}[\mathsf{subst}, M]$. The β-rule and ξ-rule follow directly.

Theorem 8 (beta): For all combinators M and N

$$\mathsf{abs}[M]N \longrightarrow^* \mathsf{subst}\ M\ N \ .$$

Theorem 9 (xi): For all combinators M and N, if $M \longrightarrow^* N$ then $\mathsf{abs}[M] \longrightarrow^* \mathsf{abs}[N]$.

VII. Translations

There is a translation from the lift λ-terms to SF-calculus that preserves reduction. It is given by

$$
\begin{array}{rcl}
[\![i]\!] & = & \mathsf{var}[S^i S, \mathsf{nil}] \\
[\![MN]\!] & = & [\![M]\!][\![N]\!] \\
[\![\lambda M]\!] & = & \mathsf{abs}\ [\![M]\!] \\
[\![\mathsf{lift}[M, k]]\!] & = & \mathsf{lift\_rec}\ \mathsf{subst}\ [\![M]\!]\ (S^k S) \ .
\end{array}
$$

The translation does not quite preserve reduction, since lift reduction introduces structure, e.g. abstractions in a manner that is not quite replicated by subst. However, the translation does preserve equality of terms, as stated by Theorem 1 in the introduction.

VIII. Verification in Coq

All the named propositions and theorems in this paper have been verified in Coq. The proof code can be found in the ancillary material associated with this paper [11]. A summary of the main results can be found in Figure 7.

IX. Related Work

The piecewise λ-calculus does not appear to have been studied previously. Its reduction rules only ever perform *needed reductions* [12] 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 [13] and surveyed by Kesner [14]. 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, piecewise substitution 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.

Predating the explicit substitution calculi is Curien's *categorical combinatory logic* [15], [16]. It corresponds to the pure λ-calculus augmented with pairs. Like SF-calculus, but unlike SKI-calculus, the categorical combinators are intensional. For example, the associativity rule $(x \circ y) \circ z = x \circ (y \circ)$ requires the composition operator $\circ$ to ask if its arguments are themselves compositions. However, the rules cannot be directed to produce a confluent rewriting system.

Another possible relation is to the work on *linear head reductions* [17], [18]. 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* [19] take quite a different approach to representing lambda-abstraction within algebra. In particular, it supports an abstract approach to substitution, rather than eliminating it.

The reduction rules of the piecewise λ-calculus only ever perform *needed reductions* [12] 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 [13] and surveyed by Kesner [14]. 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, piecewise substitution 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.

The reduction rules of the lift λ-calculus are of course reminiscent of those for λ-calculus using de Bruijn indices [20]. Some minor simplifications have been adopted here.

```
Theorem lift_confluence: confluence lambda lift_red.
Theorem lift_progress : forall (M : lambda), normal M \/ (exists N, seq_red1 M N) .
Theorem SF_confluence: confluence SF sf_red.
Theorem SF_progress : forall (M : SF), normal M \/ (exists N, sf_seqred1 M N) .
Theorem equal_programs:
  forall M, program M -> sf_red (App (App equal_comb M) M) k_op.
Theorem unequal_programs: forall M N, program M -> program N -> M<>N ->
  sf_red (App (App equal_comb M) N) (App k_op i_op).
Proposition extensions_by_matching: forall P N sigma, matching P N sigma ->
  forall M R, sf_red (App (extension P M R) N) (fold_left subst sigma M).
Proposition extensions_by_matchfail:
  forall P N,  matchfail P N ->
    forall M R, sf_red (App (extension P M R) N) (App R N).
Proposition var_check:
  forall i M N, sf_red (App (var i M) N) (var i (App (App s_op M) N)).
Proposition lift_rec_comb_op: forall s o n,
  sf_red (App (App (fix_comb (subst_rec lift_rec_comb_fn s 0)) (Op o)) n) (Op o).
Proposition lift_rec_comb_op1:
  forall s o n, sf_red (App (App (App lift_rec_comb s) (Op o)) n) (Op o).
Proposition lift_rec_comb_var: forall s i M n,
  sf_red (App (App (App lift_rec_comb s) (var i M)) n) (var (relocate_term n i)
         (App(App(fix_comb (subst_rec lift_rec_comb_fn s 0)) M) n)).
Proposition lift_rec_comb_abs: forall s n M, program s ->
  matchfail (app_comb vee (Ref 1)) s ->
    sf_red (App (App (App lift_rec_comb s) (lambda_term s M)) n)
           (lambda_term s (App (App (fix_comb (subst_rec lift_rec_comb_fn s 0))
                               M) (App (Op Sop) n))).
Proposition lift_rec_comb_app: forall s n M1 M2, program s ->
  compound (App M1 M2) ->
  matchfail (var (Ref 1) (Ref 0)) (App M1 M2) ->
  matchfail (lambda_term (Ref 1) (Ref 0)) (App M1 M2) ->
    sf_red (App (App (fix_comb (subst_rec lift_rec_comb_fn s 0)) (App M1 M2)) n)
           (App (App (App (fix_comb (subst_rec lift_rec_comb_fn s 0)) M1) n)
                (App (App (fix_comb (subst_rec lift_rec_comb_fn s 0)) M2) n)).
Proposition subst_rec_comb_op:
  forall N o, sf_red (App (App subst_rec_comb (Op o)) N) (Op o).
Proposition subst_rec_comb_var: forall N i M,
  sf_red (App (App subst_rec_comb (var i M)) N)
         (insert_Ref_comb i N (App (App subst_rec_comb M) N)).
Proposition subst_rec_comb_abs: forall N M,
  sf_red (App (App subst_rec_comb (abs M)) N)
         (abs (App (abs (swap_vars subst_rec_comb M))
                   (App (App (App lift_rec_comb subst_rec_comb) N) s_op))).
Proposition subst_rec_comb_app: forall N M1 M2, compound (App M1 M2) ->
  matchfail (var (Ref 1) (Ref 0))  (App M1 M2) ->
  matchfail (lambda_term  (Ref 1) (Ref 0)) (App M1 M2) ->
    sf_red (App (App subst_rec_comb (App M1 M2)) N)
           (App (App (App subst_rec_comb M1) N)
                (App (App subst_rec_comb M2) N)).
Theorem beta:
  forall M N, sf_red (App (abs M) N) (App (App subst_rec_comb M) N).
Theorem xi: forall M N, sf_red M N -> sf_red (abs M) (abs N).
Theorem lambda_to_SF_preserves_equality:
  forall M N, lift_equal M N -> SF_equal (lambda_to_SF M) (lambda_to_SF N).
```

Fig. 7. Propositions and theorems in Coq

For example, it is usual for lifting to take two numerical parameters; one determines the starting index for lifting, and the other determines how much lifting is to be done. Here, the amount of lifting is always limited to one.

The use of the pattern $x\ y$ which can match an arbitrary compound was a central motivation for developing the *pure pattern calculus* [7], [8], and then the SF-calculus. Some care is required to ensure that there is no needless copying of the default function; the technique used here was first elaborated by Jay and Palsberg when building typed self-interpreters [4].

X. Future Work

One obvious question not covered in this paper is the status of types. SF-calculus has been typed in [4], and used to support self-interpretation. This has been advanced to support a typed version of λSF-calculus, with key properties verified in Coq. However, it will not be a simple matter to type the translation of lift λ-calculus to SF-calculus as there is no natural relationship between a de Bruijn index and a type. Rather, one may consider representing the types within SF-calculus, perhaps as an additional parameter to $\mathsf{var}[-,-]$. Then type-level meta-functions, such as type inference, could be represented as programs in SF-calculus.

Another interesting question is the status of the Curry-Howard correspondence between programs and proofs [21]. Can it be adapted to handle intensionality? This would require a logic that can factorise proofs in the same way that F can factorise programs, so that proof theory can be incorporated into the logic itself.

The size of the terms produced by this account of substitution are far too large to be used in practice. However, dramatic contractions could be obtained by adding more operators to represent, say lifting and substitution directly, and optimising the definition of λ^* in the standard manner.

XI. Conclusions

The original motivation for combinatory logic was to give an account of abstraction in terms of simple equations on operators. Early attempts were only partially successful, in that the accounts were unable to preserve equality of terms. The solution given here has employed a more powerful combinatory calculus than the traditional SKI-calculus, namely the SF-calculus. It is able to support intensional computations, as given by pattern-matching functions, so that it is enough to recast the substitution function as an intensional function, suitable for definition as a pattern-matching function.

In turn, this requires both some cleverness and some compromise. The cleverness is to represent fixpoint functions as normal forms, suitable for pattern-matching. This can already be achieved using traditional combinators. Then the generic equality function can be used by the substitution function to recognise itself in the guise of an abstraction, a peculiar case of self-recognition. The compromise is to give up on substitution into terms without a head normal form. Such terms are of no practical significance, and are an obstacle to the theoretical development. Their only claim to status is that they are supported by pure λ-calculus.

However, λ-calculus is no longer the standard by which all foundations for computation may be judged, as it is clearly inferior to SF-calculus and λSF-calculus in its ability to represent intensional functions, such as equality, or substitution. Hence, without any regret, we may constrain substitutions to act on weak head normal forms only. In this way, combinatory calculi can now give a complete account of substitution and abstraction.

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

References

[1] H. B. Curry, R. Feys, W. Craig, and W. Craig, *Combinatory logic*. North-Holland Amsterdam, 1972.

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

[3] B. Jay and T. Given-Wilson, "A combinatory account of internal structure," *Journal of Symbolic Logic*, vol. 76, no. 3, pp. 807–826, 2011.

[4] B. Jay and J. Palsberg, "Typed self-interpretation by pattern matching," in *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, 2011, pp. 247–58.

[5] B. Jay, "Programs as data structures in λSF-calculus," *Electronic Notes in Theoretical Computer Science*, vol. 325, pp. 221 – 236, 2016, the Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

[6] B. Jay and J. Vergara, "Conflicting accounts of λ-definability," *Journal of Logical and Algebraic Methods in Programming*, vol. 87, pp. 1 – 3, 2017.

[7] B. Jay and D. Kesner, "First-class patterns," *Journal of Functional Programming*, vol. 19, no. 2, pp. 191–225, 2009.

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

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

[10] Terese, *Term Rewriting Systems*, ser. Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 53.

[11] B. Jay, "SF repository of proofs in Coq," January 2017, https://github.com/Barry-Jay/SF.

[12] H. P. Barendregt, J. R. Kennaway, J. W. Klop, and M. R. Sleep, "Needed reduction and spine strategies for the lambda calculus," *Information and Computation*, vol. 75, no. 3, pp. 191–231, 1987.

[13] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lèvy, "Explicit substitutions," in *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*. ACM, 1990, pp. 31–46. [Online]. Available: citeseer.nj.nec.com/abadi91explicit.html

[14] D. Kesner, "The theory of calculi with explicit substitutions revisited," in *International Workshop on Computer Science Logic*. Springer, 2007, pp. 238–252.

[15] P.-L. Curien, "Categorical combinators," *Information and Control*, vol. 69, no. 1, pp. 188 – 254, 1986.

[16] ——, *Categorical Combinators, Sequential Algorithms, and Functional Programming*. Boston, USA: Birkhäuser, 1993, second edition.

[17] V. Danos and L. Regnier, "Head linear reduction," *Unpublished*, 2004.

[18] P.-M. Pédrot and A. Saurin, "Classical by-need," in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, P. Thiemann, Ed. Springer, 2016, pp. 616–643.

[19] D. Pigozzi and A. Salibra, "Lambda abstraction algebras: Coordinatizing models of lambda calculus," *Fundamenta Informaticae*, vol. 33, no. 2, pp. 149–200, 1998.

[20] F. Kamareddine and A. Ríos, "A λ-calculus à la de bruijn with explicit substitutions," in *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 1995, pp. 45–62.

[21] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006, vol. 149.