A COMBINATORY ACCOUNT OF INTERNAL STRUCTURE

BARRY JAY AND THOMAS GIVEN-WILSON

Abstract. Traditional combinatory logic uses combinators S and K to represent all Turing-computable functions on natural numbers, but there are Turing-computable functions on the combinators themselves that cannot be so represented, because they have direct access to the internal structure of their arguments. Much of this expressive power is captured by adding a factorisation combinator F. The resulting SF-calculus is structure complete, in that it supports all pattern-matching functions whose patterns are in normal form, including a function that decides structural equality of arbitrary normal forms. A general characterisation of the structure complete, confluent combinatory calculi is given along with some examples. These are able to represent all their Turing-computable functions whose domain is limited to normal forms. The combinator F can be typed using an existential type to represent internal type information.

§1. **Introduction.** The traditional combinatory logic [22, 4, 10] built from combinators S and K is able to represent all the extensional functions described by λ-calculus [3, 1], and all the Turing-computable functions [24] on natural numbers [18]. However, there is a Turing-computable function that distinguishes the combinators SKK and SKS, and yet cannot be represented by application of an SK-combinator, since SKK and SKS both represent the identity function. That is, there are Turing-computable, intensional functions that are not representable by SK-combinators. This may appear surprising, since Turing machines can be simulated within SK-logic, but this simulation does not yield a representation in the sense above. The difficulty is that the process of encoding the function argument onto the tape of a Turing machine, i.e. the factoring of a combinator into its constituent operators, is a metamathematical operation [23] that is not representable.

The common reaction is to dismiss intensionality as a bad idea, as expressed in the saying "Don't look under the lambda." However, some factorisation of internal structure is representable. An example is the operator F of SF-calculus, whose reduction rules are

$$
\begin{array}{rcll}
SMNX & \longrightarrow & MX(NX) & \\
FOMN & \longrightarrow & M & \text{if } O \text{ is } S \text{ or } F \\
F(PQ)MN & \longrightarrow & NPQ & \text{if } PQ \text{ is a factorable form}
\end{array}
$$

where the *factorable forms* are the combinators of the form S, SM, SMN, F, FM and FMN for any combinators M and N. These prove to be the partially applied operators; their stability ensures that reduction is confluent.

SF-calculus is able, like SK-calculus, to model all the extensional functions of λ-calculus since K can be represented by FF. Further, since all normal forms are partially applied operators, F can be used to examine their internal structure. For example, structural equality of normal forms is now representable. Previously, combinator equality was considered indirectly by appealing to: meta-level operations [4, p. 245]; or partial combinatory algebras (not logics) such as the *uniformly reflexive structures* [25]; or *discriminators* [16, 17]. Thus, SF-calculus is the first such to support both general recursive functions [18] and decidable structural equality of arbitrary normal forms. Indeed, any Turing-computable function whose domain is restricted to SF-combinators in normal form can be represented by an SF-combinator.

SF-calculus is one of many that support factorisation. For example, one may add more operators, such as K or I, or add *constructors*, i.e. operators without any special reduction rules, such as `Pair`, `Nil` and `Cons`, that are suitable for building data structures.

The combination of extensionality and intensionality can be exploited in novel ways. For example, the original challenge, to distinguish SKK from SKS, can be generalised to a function that maps SKX to X for any combinator X, whether or not X has a normal form. This is described by the *case*

$$SKx \rightarrow x \ .$$

When applied to SKU it matches the *pattern* SKx against the argument SKU to produce a substitution of U for the variable x which is then applied to the *body* of the case x to yield U.

The expressive power of pattern matching is driven by the class of patterns supported. In mainstream function programming, patterns for data structures are constrained by the type system so that cases can be translated into λ-abstractions. Recent work in *pattern calculus* [15, 13, 8, 14, 12] drops these limitations on patterns for data structures, but does not allow matching of cases. Here, the class of patterns is expanded to include all terms in normal form, even those representing cases. A combinatory calculus that is able to represent all such cases as combinators is *structure complete*.

A structure complete calculus is able to support λ-abstraction since $\lambda x.M$ can be defined to be $x \rightarrow M$ where the pattern is the binding variable x. In particular, it can support combinators S and K with the usual properties. Also, it can support F by converting its two reduction rules into cases in the *pattern-matching function*

$$\begin{aligned} x \ y &\rightarrow (m \rightarrow n \rightarrow n \ x \ y) \\ \mid x &\rightarrow (m \rightarrow n \rightarrow m) \end{aligned}$$

where the arrow in cases is right-associative and the vertical bar combines cases. Further, since there are only finitely many operators, it can support a test `eqatom` for equality of irreducible operators, or *atoms*. These combinators F and `eqatom` suffice for determining the structure of a normal form, so it should not be surprising that S, K, F and `eqatom` suffice to ensure structure completeness.

Unlike S, the factorisation operator F does not have a simple type since the type of the components of an application are not determined by the type of

application itself. However, this can be acknowledged by using existential quantification in System **F** of variable types [7, 6].

The paper is organised as follows. Section 1 introduces the paper. Section 2 reviews some elementary facts about combinators, including the combinatorial completeness of SK-combinators. Section 3 demonstrates that SK-combinators cannot represent arbitrary symbolic computations. Section 4 introduces the factorisation operator, with its basic properties, and the example of structural equality. Section 5 considers the relationship betwen factorable forms and head normal forms. Section 6 introduces some related calculi, especially for constructing data structures. Section 7 introduces pattern-matching. Section 8 defines structure complete calculi, characterises them in general, and shows how they support generic versions of the queries used in database programming. Section 9 shows how to type $(S, K$ and$)$ F using quantifiers, and proves that reduction preserves typing. Section 10 draws conclusions.

§**2. Combinators.** This section provides a skeletal introduction to traditional combinators. Since the focus of this paper is on computation rather than logical paradoxes, it emphasises calculi over logics, with rules given by reductions rather than equations. A *combinatory calculus* is given by a finite collection $\mathcal{O}$ of *operators* (meta-variable O) that are used to define the $\mathcal{O}$-*combinators* (meta-variables M, N, P, Q, X, Y, Z) built from these by application

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

Syntactic equality of combinators will be denoted by $\equiv$. The $\mathcal{O}$-*combinatory calculus* or $\mathcal{O}$-*calculus* is given by the combinators plus their reduction rules. A *homomorphism* of combinatory calculi is a mapping of combinators that preserves application and reduction.

The classic SK-calculus has *reduction rules*

$$\begin{aligned} SMNX &\longrightarrow MX(NX) \\ KXY &\longrightarrow X \, . \end{aligned}$$

The combinator $SMNX$ duplicates X as the argument to both M and N. The combinator KXY eliminates Y and returns X.

The rules are instantiated by replacing each meta-variable M, N, X or Y by a particular combinator. The *reduction relation* (also, denoted $\longrightarrow$) is the relation obtained by applying an instantiation of a reduction rule to some sub-expression. The reflexive, transitive closure of the reduction relation is denoted $\longrightarrow^*$ though the star may be elided if it is obvious from the context.

Further, the relation $\longrightarrow^*$ induces an equivalence relation $=$ on the combinators, their *equality*. The $\mathcal{O}$-*combinatory logic* or $\mathcal{O}$-*logic* is the system of equivalence classes of combinators from $\mathcal{O}$-combinatory calculus.

The SK-calculus can be translated to λ-calculus as follows [4, 5, 1, 10]:

$$\begin{aligned} [\![S]\!] &= \lambda g.\lambda f.\lambda x.g \ x \ (f \ x) \\ [\![K]\!] &= \lambda x.\lambda y.x \\ [\![MN]\!] &= [\![M]\!] \ [\![N]\!] \, . \end{aligned}$$

For example

$$\begin{aligned}
[\![SKX]\!] &= (\lambda g.\lambda f.\lambda x.g \ x \ (f \ x)) \ (\lambda x.\lambda y.x) \ [\![X]\!] \\
&\longrightarrow (\lambda f.\lambda x.(\lambda x.\lambda y.x) \ x \ (f \ x)) \ [\![X]\!] \\
&\longrightarrow \lambda x.(\lambda x.\lambda y.x) \ x \ ([\![X]\!] \ x) \\
&\longrightarrow \lambda x.(\lambda y.x) \ ([\![X]\!] \ x) \\
&\longrightarrow \lambda x.x
\end{aligned}$$

for any combinator X.

THEOREM 2.1. *Translation from SK-calculus to λ-calculus preserves reduction.*

PROOF. It is enough to consider the reduction rules:

$$\begin{aligned}
[\![SMNX]\!] &= (\lambda g.\lambda f.\lambda x.g \ x \ (f \ x)) \ [\![M]\!] \ [\![N]\!] \ [\![X]\!] \\
&\longrightarrow [\![M]\!] \ [\![X]\!] \ ([\![N]\!] \ [\![X]\!]) \\
&= [\![MX(NX)]\!] \\
[\![KXY]\!] &= (\lambda x.\lambda y.x) \ [\![X]\!] \ [\![Y]\!] \\
&\longrightarrow [\![X]\!] \ .
\end{aligned}$$

$\dashv$

It will be convenient to introduce some familiar logical constructs. Define the conditional if P then M else N by PMN. Then truth is given by K since $KMN \longrightarrow M$ while falsehood is given by KI since $KIMN \longrightarrow IN \longrightarrow N$. The usual boolean operations are defined in the obvious way; write not M for *negation*; M and N for the *conjunction* of M and N; M or N for their *disjunction*; and M implies N for *implication*. Similarly, there is a fixpoint combinator fix with the property that fix $M \longrightarrow^* M(\text{fix } M)$.

One of the goals of combinatory logic is to give an equational account of variable binding and substitution, especially as it appears in λ-calculus. More generally, one may consider the ability to represent arbitrary computable functions that act upon combinators.

A *symbolic function* is here defined to be an n-ary partial function $\mathcal{G}$ of some combinatory logic, i.e. a function of the combinators that preserves their equality, as determined by the reduction rules. That is, if $X_i = Y_i$ for $1 \leq i \leq n$ then $\mathcal{G}(X_1, X_2, \ldots, X_n) = \mathcal{G}(Y_1, Y_2, \ldots, Y_n)$ if both sides are defined. A symbolic function is *restricted* to a set of combinators, e.g. the normal forms, if its domain is within the given set.

A combinator G in the calculus *represents* $\mathcal{G}$ if

$$GX_1 \ldots X_n = \mathcal{G}(X_1, \ldots, X_n)$$

whenever the right-hand side is defined. For example, the symbolic functions

$$\begin{aligned}
\mathcal{S}(X_1, X_2, X_3) &= X_1 X_3 (X_2 X_3) \\
\mathcal{K}(X_1, X_2) &= X_1
\end{aligned}$$

are represented by S and K, respectively, in SK-calculus. Again, consider the symbolic function

$$\mathcal{I}(X) = X .$$

In SKI-calculus where I has the rule

$$IY \longrightarrow Y$$

then $\mathcal{I}$ is represented by I. In both SKI-calculus and SK-calculus $\mathcal{I}$ is represented by any combinator of the form SKX since

$$SKXY = KY(XY) = Y .$$

For later convenience, define I to be SKK in SK-calculus.

Of course, one can develop an algebra of symbolic functions by closing them under composition etc. but this is not necessary for the purposes of this paper.

In order to represent λ-abstraction, it is necessary to have some variables to work with. Given $\mathcal{O}$ as before, define the $\mathcal{O}$-*terms* by

$$M, N ::= x \mid O \mid MN$$

where x is as in λ-calculus. Free variables and the substitution $\{N/x\}M$ of the term N for the variable x in the term M are defined in the obvious manner, since the term calculus does not have any binding constructions built in. The $\mathcal{O}$-*term calculus* has reduction defined by the same rules as the $\mathcal{O}$-calculus, noting that instantiation may introduce variables. Symbolic computation and representation can be defined for terms just as for combinators.

Given a variable x and term M define a symbolic function $\mathcal{G}$ on terms by

$$\mathcal{G}(X) = \{X/x\}M .$$

Note that if M has no free variables other than x then $\mathcal{G}$ is also a symbolic computation of the combinatory logic. If every such function $\mathcal{G}$ on $\mathcal{O}$-combinators is representable then the $\mathcal{O}$-combinatory logic is *combinatorially complete* in the sense of Curry [4, p. 5].

Given S and K then $\mathcal{G}$ above can be represented by a term $\lambda^*x.M$ given by

$$\begin{aligned}
\lambda^*x.x &= I \\
\lambda^*x.y &= Ky \quad \text{if } y \neq x \\
\lambda^*x.O &= KO \\
\lambda^*x.MN &= S(\lambda^*x.M)(\lambda^*x.N) .
\end{aligned}$$

LEMMA 2.2. *For all terms M and N and variables x there is a reduction*

$$(\lambda^*x.M)\, N \longrightarrow^* \{N/x\}M .$$

PROOF. The proof is by induction on the structure of the combinator M.

- If M is x then $(\lambda^*x.M)N \equiv IN \longrightarrow N \equiv \{N/x\}M$.
- If M is any other variable or an operator then $(\lambda^*x.M)N \equiv KMN \longrightarrow M \equiv \{N/x\}M$.

- Finally, if M is of the form $M_1 M_2$ then

$$
\begin{aligned}
(\lambda^* x.M)N & \equiv S(\lambda^* x.M_1)(\lambda^* x.M_2)N \\
& \longrightarrow (\lambda^* x.M_1)N((\lambda^* x.M_2)N) \\
& \longrightarrow \{N/x\}M_1(\{N/x\}M_2) \\
& \equiv \{N/x\}M
\end{aligned}
$$

by two applications of induction.

$\dashv$

The following theorem is a central result of combinatory logic [4].

THEOREM 2.3. *Any combinatory calculus that is able to represent S and K is combinatorially complete.*

PROOF. Given $\mathcal{G}(X) = \{X/x\}M$ as above define G to be $\lambda^* x.M$ and apply Lemma 2.2. $\dashv$

Where no confusion is likely, we may write $\lambda x.M$ for $\lambda^* x.M$.

§3. Symbolic Computation. For the purposes of this paper, all effective calculation may be assumed to be Turing computable. Hence, define a *symbolic computation* to be a Turing-computable symbolic function. This section introduces factorisation for any confluent combinatory calculus by the symbolic computation $\mathcal{F}$ and shows that it is not representable in SK-calculus.

Define the symbolic function $\mathcal{R}$ on combinators by

$$
\begin{aligned}
\mathcal{R}(O, M, N) &= M \\
\mathcal{R}(PQ, M, N) &= NPQ \ .
\end{aligned}
$$

That is, $\mathcal{R}$ branches according to whether it can *factorise* its first argument. Of course, $\mathcal{R}$ does not respect equality since applications may reduce to operators.

One approach to handling $\mathcal{R}$ would be to modify the reduction relation, so that rules cannot be applied to the right-hand side of an application. This approach is adopted for Kearns' system of *discriminators* [16, 17] which includes a discriminator R that is similar to $\mathcal{R}$ above. Discriminators are well suited to their purpose of directly modelling the symbolic computations of Turing machines, with their asymmetric treatment of state and tape. Here $\mathcal{R}$ preserves this weakened notion of reduction, but the equivalence relation is not an equality relation in the sense of Leibnitz, which permits the substitution of equals for equals.

The approach adopted here is to restrict the equations for $\mathcal{R}$ to partially applied operators. Each operator O has an *arity* given by the minimum number of arguments it requires to instantiate a rule. Thus, K has arity 2 while S has arity 3. A *partially applied operator* is a combinator of the form $OX_1 \ldots X_k$ where k is less than the arity of O. An operator with a positive arity is an *atom* (meta-variable A). A partially applied operator that is an application is a *compound*. Hence, the partially applied operators of SK-calculus are the atoms S and K, and the compounds SM, SMN and KM for any M and N.

Now define a *factorisation function* $\mathcal{F}$ on combinators by

$$\begin{aligned}
\mathcal{F}(A, M, N) &\longrightarrow M & &\text{if } A \text{ is an atom} \\
\mathcal{F}(PQ, M, N) &\longrightarrow NPQ & &\text{if } PQ \text{ is a compound.}
\end{aligned}$$

LEMMA 3.1. *If reduction is confluent then factorisation is a symbolic computation.*

PROOF. To prove that $\mathcal{F}$ is a symbolic function, it suffices to prove that $\mathcal{F}(X, M, N) = \mathcal{F}(X', M', N')$ whenever both sides are defined and $X = X'$ and $M = M'$ and $N = N'$ are three pairs of combinators. If X is a compound PQ then, by confluence, X' must also be a compound $P'Q'$ such that $P = P'$ and $Q = Q'$. Thus, $\mathcal{F}(X, M, N) = NPQ = N'P'Q' = \mathcal{F}(X', M', N')$. Similarly, if X is an atom A then by confluence X' must also be A so that $\mathcal{F}(X, M, N) = M = M' = \mathcal{F}(X', M', N')$. Finally, $\mathcal{F}$ is computable by leftmost reduction of its first argument to a partially applied operator. $\dashv$

THEOREM 3.2. *Factorisation of SK-combinators is a symbolic computation that is not representable.*

PROOF. Suppose that there is an SK-combinator F that represents $\mathcal{F}$. Then, for any combinator X we have

$$F(SKX)S(KI) \longrightarrow KI(SK)X \longrightarrow X \ .$$

Translating this to λ-calculus as in Lemma 2.2 yields $[\![F(SKX)S(KI)]\!] \longrightarrow [\![X]\!]$ and also

$$[\![F(SKX)S(KI)]\!] = [\![F]\!]\,[\![(SKX)]\!]\,[\![S]\!]\,[\![KI]\!] \longrightarrow [\![F]\!]\,(\lambda x.x)\,[\![S]\!]\,[\![KI]\!] \ .$$

Hence, by confluence of reduction in λ-calculus, all $[\![X]\!]$ share a reduct with $[\![F]\!]\,(\lambda x.x)\,[\![S]\!]\,[\![KI]\!]$ but this is impossible since $[\![S]\!]$ and $[\![K]\!]$ are distinct normal forms. Hence $\mathcal{F}$ cannot be represented by an SK-combinator. $\dashv$

This result stresses the traditional understanding of computation. On the one hand, factorisation can be encoded using Turing machines in the obvious manner. On the other hand, combinatory logic, λ-calculus and Turing machines all compute the same things.

The tension eases upon observing that the classical theorems address numerical computations rather than symbolic ones. For example, Kleene [18] states Church's thesis as

> THESIS 1: Every effectively calculable function (effectively decidable predicate) is general recursive.

Since general recursive functions are numerical by definition, it is clear that there is an implicit restriction of effective calculation to numbers, and things that can be encoded as numbers by Gödelisation.

It may be objected that the proofs of the numerical results employ encodings that are symbolic, not numerical, and so can be generalised. In particular, one can encode an SK-combinator on the tape of a Turing machine that performs combinator factorisation, and then express this machine in SK-logic, so that factorisation of SK-combinators can be expressed in SK-calculus. However, the crucial point is that this expression of factorisation does not imply its representation as a combinator.

For example, consider the factorisation of SKK. Using Polish notation, it can be encoded on a tape by $aaSKK$ where a represents application. Then expressing the tape as an SK combinator yields a list $[a, a, S, K, K]$ whose factorisation is a routine list operation. However, to produce this list within the calculus would require support for factorisation that SK-calculus does not possess (Theorem 3.2). In other words, the metamathematical process of encoding a combinator onto the tape of a Turing machine is not representable by an SK-combinator. Note that Gödelisation is not relevant here, being solely concerned with the representation of $[a, a, S, K, K]$ by a number.

Summarising, the relationship between combinatory calculi and Turing machines presupposes the ability to encode combinators as lists of operators (and applications). SK-calculus is able to handle numbers, since their factorisation is handled by the zero test and the predecessor function. At the other extreme, non-normalising combinators are a challenge for any calculus. In between lie the normal forms, whose factorisation is considered next.

§4. Factorisation. This section shows that factorisation can be represented in a combinatory calculus, namely SF-calculus. Basic properties are established and examples are given for later use.

It is tempting to specify F as a representative for $\mathcal{F}$ but the latter is defined using the partially applied operators, which cannot be known until all the reduction rules are given, including those for F. This circularity is easily broken by beginning with a syntactic characterisation of the combinators that are to be factorised.

The SF-calculus has *factorable forms* given by

$$S \mid SM \mid SMN \mid F \mid FM \mid FMN$$

and *reduction rules*

$$
\begin{array}{rcll}
SMNX & \longrightarrow & MX(NX) & \\
FOMN & \longrightarrow & M & \text{if } O \text{ is } S \text{ or } F \\
F(PQ)MN & \longrightarrow & NPQ & \text{if } PQ \text{ is a factorable form.}
\end{array}
$$

LEMMA 4.1. *The partially applied operators of SF-calculus are its factorable forms. Hence F represents $\mathcal{F}$.*

PROOF. Trivial.   ⊣

THEOREM 4.2. *Reduction of SF-calculus is confluent.*

PROOF. It is enough to observe that the reduction rules are orthogonal [21, 11], since partially applied operators are stable under reduction.   ⊣

The expressive power of SF-calculus subsumes that of SK-calculus since K is here defined to be FF (and I is defined to be SKK as before). Its further power is illustrated by defining a combinator for structural equality of normal forms. In brief, the algorithm is as follows: if both normal forms are compounds then compare their corresponding components: if both are atoms then compare them directly using a combinator `eqatom`; otherwise the normal forms are not equal. Here are the details.

Clearly F is able to distinguish the atoms from the compounds. The combinator $\mathtt{isComp} = \lambda x.Fx(KI)(K(KK))$ tests for being a compound since

$$
\begin{aligned}
\mathtt{isComp}\ O &\longrightarrow FO(KI)(K(KK)) \\
&\longrightarrow KI \\
\mathtt{isComp}(PQ) &\longrightarrow F(PQ)(KI)(K(KK)) \\
&\longrightarrow K(KK)PQ \\
&\longrightarrow KKQ \\
&\longrightarrow K \qquad \text{if } PQ \text{ is a compound.}
\end{aligned}
$$

Further, the first and second components of a factorable application can be recovered by

$$
\begin{aligned}
\mathtt{car} &= \lambda x.FxIK \\
\mathtt{cdr} &= \lambda x.FxI(KI)
\end{aligned}
$$

whose names are taken from the corresponding Lisp [20] operators. Note that they map operators to I so it is normal to check for being a compound first.

Now all that remains is to separate the operators S and F. Define $\mathsf{is}(F)$ by

$$
\mathsf{is}(F) = \lambda x.x(KI)(K(KI))K\ .
$$

It maps F to K and S to KI as desired since

$$
\begin{aligned}
\mathsf{is}(F)\ F &\longrightarrow F(KI)(K(KI))K \longrightarrow KKI \longrightarrow K \\
\mathsf{is}(F)\ S &\longrightarrow S(KI)(K(KI))K \longrightarrow KIK(K(KI)K) \longrightarrow KI\ .
\end{aligned}
$$

Further, define equality of operators by

$$
\mathtt{eqatom} = \lambda x.\lambda y.\mathtt{if}\ \mathsf{is}(F)x\ \mathtt{then}\ \mathsf{is}(F)y\ \mathtt{else}\ (\mathtt{not}\ \mathsf{is}(F)y)\ .
$$

Structural equality of normal forms can now be given by

```
equal = fix(λe.λx.λy.
   if isComp x
   then if isComp y
        then (e (car x) (car y)) and (e (cdr x) (cdr y))
        else KI
   else if isComp y
        then KI
        else eqatom x y .
```

THEOREM 4.3. *Let M and N be SF-combinators in normal form. If $M = N$ then* $\mathtt{equal}\ M\ N \longrightarrow K$ *else* $\mathtt{equal}\ M\ N \longrightarrow KI$.

PROOF. The proof is by straightforward induction on the structure of M. $\dashv$

Thus, the combinator $\mathtt{equal}$ represents the symbolic computation whose domain is given by pairs of combinators that have a normal form. Of course, $\mathtt{equal}$ also detects inequality of, say, an operator O and a factorable form PQ even if Q does not have a normal form.

The development above illustrates a more general result.

THEOREM 4.4. *Any symbolic computation restricted to normal forms of SF-calculus is representable.*

PROOF. The encoding of the normal form of the function argument can be revealed by first factorising into operators and then using `eqatom` to identify the operator values. That done, the computation can be represented by a combination of S and K in the traditional manner. ⊣

§5. Head Normal Forms. In extensional calculi such as SK-calculus, the head normal forms are the partially applied operators, but this is not true of intensional calculi such as SF-calculus. This section defines head normal forms in this new setting, and shows that the definition of SF-calculus cannot be modified to force the factorable forms to be the head normal forms without making the logic unsound.

The *head normal forms* are defined by induction on their structure. An operator is head normal if it is irreducible. An application PQ is head normal if P is head normal and no reduct of PQ instantiates a reduction rule.

For SK-calculus the head normal forms are exactly the partially applied operators, but this is not true of SF-calculus as the combinator $\Omega = (SII)(SII)$ does not reduce to a factorable form, and so $F\Omega FF$ is a head normal form but not a partially applied operator.

THEOREM 5.1. *Given a collection of* headable forms *among the combinators built from two operators S and H, consider the combinatory calculus defined by the reduction rules*

$$
\begin{array}{rcll}
SMNX & \longrightarrow & MX(NX) & \\
HOMN & \longrightarrow & M & O \text{ is } S \text{ or } H \\
H(PQ)MN & \longrightarrow & NPQ & PQ \text{ is headable.}
\end{array}
$$

Let $K = HH$ and $I = SKK$ as usual. If the head normal forms of this calculus are the headable forms then the resulting logic is unsound, i.e. $K = KI$.

PROOF. The proof proceeds by using the decidability of head normality to obtain decidability of normality, which yields a contradiction. Consider the combinator

$$\texttt{isn} = \texttt{fix}(\lambda i.\lambda x.H(HxH(\lambda y.\lambda z.H(i\ y)(i\ z)I))\ S\ (K(K(SS))))\ .$$

For all combinators X, the combinator $\texttt{isn}\,X$ reduces to S if X is normalisable and to SS otherwise. The proof is by induction on the structure of X.

First consider the situation when X does not reduce to a head normal form or headable form. Then $(HXH(\lambda y.\lambda z.H(\texttt{isn}\ y)(\texttt{isn}\ z)I))$ is a head normal form, i.e. a headable form, and so

$$
\begin{aligned}
\texttt{isn}\,X &\longrightarrow H(HXH(\lambda y.\lambda z.H(\texttt{isn}\,y)(\texttt{isn}\,z)I))\ S\ (K(K(SS)))) \\
&\longrightarrow K(K(SS))(HXH)(\lambda y.\lambda z.H(\texttt{isn}\,y)(\texttt{isn}\,z)I) \\
&\longrightarrow SS
\end{aligned}
$$

as required. If X is an atom A then

$$
\begin{aligned}
\mathtt{isn}\, A &\longrightarrow H(HAH(\lambda y.\lambda z.H(\mathtt{isn}\, y)(\mathtt{isn}\, z)I))\ S\ (K(K(SS)))) \\
&\longrightarrow HHS(K(K(SS))) \\
&\longrightarrow S\ .
\end{aligned}
$$

If X is a headable form PQ then

$$
\begin{aligned}
\mathtt{isn}(PQ) &\longrightarrow H(H(PQ)H(\lambda y.\lambda z.H(\mathtt{isn}\, y)(\mathtt{isn}\, z)I))\ S\ (K(K(SS)))) \\
&\longrightarrow H((\lambda y.\lambda z.H(\mathtt{isn}\, y)(\mathtt{isn}\, z)I)PQ)\ S\ (K(K(SS)))) \\
&\longrightarrow H(H(\mathtt{isn}\, P)(\mathtt{isn}\, Q)I)\ S\ (K(K(SS))))\ .
\end{aligned}
$$

If P is not normalisable then

$$
\begin{aligned}
\mathtt{isn}(PQ) &\longrightarrow H(H(SS)(\mathtt{isn}\, Q)I)\ S\ (K(K(SS)))) \\
&\longrightarrow H(ISS)\ S\ (K(K(SS)))) \\
&\longrightarrow SS
\end{aligned}
$$

as required. If P is normalisable then

$$
\begin{aligned}
\mathtt{isn}(PQ) &\longrightarrow H(HS(\mathtt{isn}\, Q)I)\ S\ (K(K(SS)))) \\
&\longrightarrow H(\mathtt{isn}\, Q)\ S\ (K(K(SS))))
\end{aligned}
$$

If Q is normalisable then this reduces to $HSS(K(K(SS))) \longrightarrow S$ while if Q is not normalisable then it reduces to $K(K(SS))SS \longrightarrow SS$, all as required.

This completes the proof of the properties of $\mathtt{isn}$. Now it is routine to show that

$$
\mathtt{isnormal} = \lambda x.H(\mathtt{isn}\, x)K(K(K(KI)))
$$

decides whether its argument has a normal form. Finally, consider the paradoxical combinator

$$
\begin{aligned}
\mathtt{paradox} &= \mathtt{fix}(\lambda f.\mathtt{if\ isnormal}\, f\ \mathtt{then}\ \Omega\ \mathtt{else}\ K) \\
&= \mathtt{if\ isnormal\ paradox\ then}\ \Omega\ \mathtt{else}\ K\ .
\end{aligned}
$$

If $\mathtt{isnormal\ paradox} = KI$ then $\mathtt{paradox} = K$ and so

$$
KI = \mathtt{isnormal\ paradox} = \mathtt{isnormal}\, K = K
$$

in which case the logic is unsound. Alternatively, if $\mathtt{isnormal\ paradox} = K$ then $\mathtt{paradox} = \Omega$ so that $K = \mathtt{isnormal\ paradox} = \mathtt{isnormal}\, \Omega = KI$.   ⊣

§6. **Related Combinatory Calculi.** Now consider how the factorisation of Section 3 can be exploited in the presence of different operators. Each calculus to be developed includes a formal description of its factorable forms. It is then trivial to show that these are the partially applied operators and that reduction is confluent in the style of the corresponding proofs for SF-calculus. It is easy to confirm in each case that structural equality of normal forms is definable, once equality of atoms is supported, and that the analogue of Theorem 4.4 holds. Rather than do the proofs here, these results will follow from Corollaries 8.4 and 8.5.

Perhaps the closest calculus to SF-calculus is SKF-calculus where S, K and F take their usual meanings and the factorable forms are S, SM, SMN, K, KM, F, FM and FMN.

Define:

$$\mathsf{is}(K) = \lambda x.F(xFF)K(K(K(KI)))$$
$$\mathsf{is}(F) = \lambda x.x(KI)(K(KI))K$$
$$\mathtt{eqatom} = \lambda x.\lambda y.\mathtt{if}\ \mathsf{is}(K)x\ \mathtt{then}\ \mathsf{is}(K)y$$
$$\mathtt{else\ if}\ \mathsf{is}(F)x\ \mathtt{then}\ \mathsf{is}(F)y\ \mathtt{else}\ (\mathtt{not}\ (\mathsf{is}(K)y\ \mathtt{or}\ \mathsf{is}(F)y))\ .$$

There is a trivial homomorphism of SF-calculus into SKF-calculus that maps S to S and F to F but it is not clear if there is a homomorphism in the opposite direction. The natural approach would be to map S to S and K to FF but then F cannot be mapped to F since $FKMN$ reduces to M in SKF-calculus but to NFF in SF-calculus, so this will not do. Such problems will arise whenever an atom is translated to a compound, as when Schönfinkel's original combinators are represented as SK-combinators. Hence, it is not yet clear if there is a "best" combinatory logic, much less that SF-calculus is best.

Another way of extending SF-calculus is to add constructors. A *constructor* is an atom that does not appear at the head of any reduction rule, so that its arity is infinite. Typical examples are `Pair` for building pairs, or `Nil` for the empty list. Let $\mathcal{C}$ be a finite collection of constructors (meta-variable C). Also required is an operator `eqatom` for deciding equality of atoms, since constructors are extensionally equal.

The constructors are used to build *data structures* (meta-variable d) given by

$$d ::= C \mid d\ M\ .$$

That is, data structures are combinators headed by a constructor. The factorable forms are given by

$$d \mid S \mid SM \mid SMN \mid F \mid FM \mid FMN \mid \mathtt{eqatom} \mid \mathtt{eqatom}\ M$$

which now include all the data structures. The reduction rules for S and F are as usual. The reduction rules for `eqatom` are

$$\mathtt{eqatom}\ O\ O \longrightarrow K$$
$$\mathtt{eqatom}\ P\ Q \longrightarrow KI\ \text{otherwise, if } P \text{ and } Q \text{ are factorable.}$$

Note, too, that any countable collection of constructors can be encoded as data structures built from a single, universal constructor C, as $CC, C(CC), C(CCC)$ etc; exploiting this yields the SFC-calculus. There is no need for `eqatom` to be an operator, since it can be defined as follows. The combinator

$$\mathsf{is}(C) = \lambda x.F(x(FK)IK)(KI)(K(KK))$$

maps C to K and maps S and F to KI. Hence, `eqatom` can be defined by using $\mathsf{is}(C)$ first and then separating S and F as before.

Finally, data structures can be represented in a variant of SF-calculus in which $\Omega = (SII)(SII)$ plays the role of constructor. Define $\{S, F, \Omega\}$-calculus to have *data structures*

$$d ::= \Omega \mid d\ M$$

and *factorable forms*

$$d \mid S \mid SM \mid SMN \mid F \mid FM \mid FMN$$

and reduction rules

$$
\begin{aligned}
SMNX &\longrightarrow MX(NX) \\
FXMN &\longrightarrow M \qquad \text{if } X \in \{S, F\} \\
F(PQ)MN &\longrightarrow NPQ \qquad \text{if } PQ \text{ is factorable .}
\end{aligned}
$$

Confluence is established as for SF-calculus since the only reduct of Ω that is a factorable form is Ω itself. Again, eqatom can be defined to distinguish S and F and Ω from each other using

$$\mathsf{is}(\Omega) = \lambda x.F(x(FK)IK)(KI)(K(KK))$$

which is the obvious adaptation of $\mathsf{is}(C)$ above. In a sense, Ω is playing the role of a constructor, even though it is not a operator, or even a partially applied operator. Obviously, this approach can be generalised to include other non-normalising combinators as "constructors."

§7. Pattern Matching. Pattern matching explores internal structure in a way that can be represented by factorisation. This section considers pattern matching as symbolic computation, as a prelude to the representation of pattern-matching functions in the section following.

Consider a confluent combinatory calculus whose *patterns* (meta-variable P) are its terms that are in normal form. From now on we shall restrict attention to *linear patterns* in which no variable occurs twice. While this may seem a little artificial, non-linear patterns describe structures that come with side-conditions about the equality of substructures; it is simpler, and more natural, to replace these side conditions with an explicit equality test.

A *case* $\mathcal{G}$ is given by an equation of the form

$$\mathcal{G}(P) = M$$

where P is a pattern and M is an arbitrary term. Such a case yields a symbolic function on terms given by pattern matching, which is defined as follows.

When $\mathcal{G}$ is applied to some term U the pattern P will be matched against U to try and determine the values of the free variables in P so that these can be substituted into M. That is, matching seeks a substitution σ such that $\sigma P = U$. However, the presence or absence of such a substitution is not an infallible guide to evaluation.

If the equation $\sigma P = U$ is that of the logic then there can be more than one such substitution. For example, consider that P is $x\ y$ and U is S. A naive interpretation would consider that matching must fail, but recall that $SKSS = S = SKKS$ and so it would be acceptable to match x against either SKS or SKK. Rather than take this course, it is more natural to develop a syntactic procedure for matching. In turn, this must respect reduction, which requires a notion of partially applied operator in a term calculus.

The *matchable forms* upon which matching acts can be identified with the partially applied operators of the term calculus on the understanding that variables have arity 0. It follows that variables are neither atoms nor appear at the

head of a compound, which is appropriate since substitution may trigger new reductions.

Define a *match* to be either a *successful match*, some σ where σ is a substitution, or a *match failure*, none. *Match equality* is defined using term equality. Two successful matches some σ_1 and some σ_2 are equal if σ_1 and σ_2 have the same domain and $\sigma_1 x = \sigma_2 x$ for each variable x in their domain. Also none equals none. Otherwise, matches are not equal. Disjoint unions $\uplus$ of matches are defined as follows. If both matches are successful then form the disjoint union of their substitutions (regarded as relations). If either match is undefined then so is their disjoint union. Otherwise the result is none.

The application of a match to a term is defined by

$$\text{some } \sigma \ M = \sigma M$$
$$\text{none } M = I \ .$$

For definiteness, match failure must produce a combinator; the identity proves to be a useful choice when defining extensions in the next section.

The *match* $\{U/P\}$ of a pattern P against a term U is defined by

$$
\begin{array}{llll}
\{U/x\} & = & \text{some } \{U/x\} & \\
\{A/A\} & = & \text{some } \{\} & \text{if } A \text{ is an atom} \\
\{UV/PQ\} & = & \{U/P\} \uplus \{V/Q\} & \text{if } UV \text{ is a compound} \\
\{U/P\} & = & \text{none otherwise} & \text{if } U \text{ is matchable} \\
\{U/P\} & = & \text{undefined} & \text{otherwise.}
\end{array}
$$

The restrictions to matchable forms in the above definition are necessary to ensure that matching is stable under reduction of U. For example, if the pattern is $x \ y$ and the argument is $SMNX$ then x should *not* be bound to SMN. Conversely, if the pattern is $S \ y$ and U is $SMNX$ then matching should not (yet) fail. In each case the match is undefined until $SMNX$ is reduced to some matchable form.

Now the case $\mathcal{G}$ introduced earlier becomes a partial function of combinators defined by

$$\mathcal{G}(U) = \{U/P\}M \ .$$

LEMMA 7.1. *Cases are symbolic computations on confluent term calculi.*

PROOF. For $\mathcal{G}$ above to be well-defined, it suffices to prove that if $U = U'$ and the matches $\{U/P\}$ and $\{U'/P\}$ are both defined then these matches are equal. The proof is by induction upon the structure of P. If P is a variable x then $\{U/x\} = \{U'/x\}$ since $U = U'$. Otherwise, U and U' must be matchable forms. By confluence, if U is an atom then U' must be the same atom, while if U is some compound $U_1 U_2$ then U' must be some compound $U_1' U_2'$ such that $U_1 = U_1'$ and $U_2 = U_2'$. Thus the only possibility of interest is when P is an application $P_1 P_2$ and U and U' are compounds as described above. Hence

$$\{U/P\} = \{U_1/P_1\} \uplus \{U_2/P_2\} = \{U_1'/P_1\} \uplus \{U_2'/P_2\} = \{U'/P\}$$

by two applications of induction.   ⊣

§8. Structure Completeness. A confluent combinatory calculus is *structure complete* if, for every normal term P and term M, the case $\mathcal{G}(P) = M$ is represented by some term $P \to M$.

Such a calculus is combinatorially complete since $\lambda x.M$ is given by $x \to M$. Hence, the calculus has combinators S and K with the usual behaviours. Given a sequence of cases $P_i \to M_i$ the *pattern-matching function*

$$
\begin{aligned}
&P_1 \to M_1\\
&\mid P_2 \to M_2\\
&\quad \ldots\\
&\mid P_n \to M_n
\end{aligned}
$$

is defined as follows. When applied to some argument, it reduces to the first case $P_i \to M_i$ where matching succeeds. Fortunately, it is not necessary to generalise the definition of structure completeness to handle such functions, since they can be represented as cases using *extensions* [15, 12]. In the combinatory setting, the *extension* of a combinator N (the *default*) by a *special case* consisting of a pattern P and a body M is given by

$$P \to M \mid N = S(P \to KM)N .$$

When applied to some term U such that $\{U/P\} = \mathsf{some}\ \sigma$ for some substitution σ then

$$
\begin{aligned}
(P \to M \mid N)U\ &=\ S(P \to KM)NU\\
&\longrightarrow (P \to KM)U(NU)\\
&\longrightarrow \sigma(KM)(NU)\\
&=\ K(\sigma M)(NU) = \sigma M .
\end{aligned}
$$

Alternatively, if $\{U/P\}$ is none then

$$
\begin{aligned}
(P \to M \mid N)U\ &\longrightarrow (P \to KM)U(NU)\\
&\longrightarrow I(NU)\\
&\longrightarrow NU .
\end{aligned}
$$

For example, F is defined by

$$
\begin{aligned}
&x\ y \to (m \to n \to n\ x\ y)\\
&\mid x \to (m \to n \to m)
\end{aligned}
$$

where the arrow in the case is right-associative. Similarly, $\mathtt{eqatom}$ is defined by

$$
\begin{aligned}
\mathtt{eqatom} =&\\
&A_1 \to (A_1 \to K \mid y \to KI)\\
&\mid A_2 \to (A_2 \to K \mid y \to KI)\\
&\quad \ldots\\
&\mid A_n \to (A_n \to K \mid y \to KI)\\
&\mid x \to y \to KI .
\end{aligned}
$$

where $A_1, \ldots, A_n$ is a listing of the finite collection of atoms.

A confluent combinatory calculus *supports duplication* (respectively, *elimination, factorisation, separation of atoms*) if it has a combinator S (respectively,

$K, F, \mathtt{eqatom}$) that represents $\mathcal{S}$ (respectively, $\mathcal{K}, \mathcal{F}$, equality of atoms). Less formally, it *supports S* (respectively, $K, F, \mathtt{eqatom}$) if it supports the corresponding symbolic function.

LEMMA 8.1. *For any three of S, K, F and $\mathtt{eqatom}$ there is a confluent combinatory calculus that supports them but does not support the fourth.*

PROOF. For not supporting S, consider the F-calculus with factorable forms F, FM and FMN. Then define $K = FF$ and $\mathtt{eqatom} = K(KK)$. As nothing can be duplicated, S is not representable.

For K, consider the S-calculus in which the usual rule for S is supplemented by $S \longrightarrow S$. Since there are no normal forms there are no partially applied operators or atoms, so that we may define $F = \mathtt{eqatom} = S$. However, nothing can be eliminated by S so K is not definable.

For F, use SK-calculus with $\mathtt{eqatom}$ defined by extensionality.

For $\mathtt{eqatom}$, consider SFT-calculus in which T satisfies the same rules as S. ⊣

THEOREM 8.2. *A confluent combinatory calculus is structure complete if and only if it supports S, K, F and $\mathtt{eqatom}$.*

PROOF. The forward direction follows from the previous constructions. For the converse, suppose that suitable combinators S, K, F and $\mathtt{eqatom}$ exist.

Every case $\mathcal{G}$ defined by $\mathcal{G}(P) = M$ is represented by $P \to M$ which is defined by induction on the structure of P, employing a fresh variable x, as follows:

- If P is a variable y then $P \to M$ is $\lambda y.M$.
- If P is an atom A then $P \to M$ is $\lambda x.Fx(\mathtt{eqatom}\, AxMI)(K(KI))$.
- If P is an application P_1P_2 then $P \to M$ is

$$\lambda x.FxI(S(P_1 \to K(P_2 \to M))(K(KI)))$$

The proof that $P \to M$ represents $\mathcal{G}(P) = M$ is by induction on the structure of P. Let U be a combinator such that $\mathcal{G}(U)$ is defined and consider $(P \to M)U$. Without loss of generality, no free variable of P is free in U.

- If P is a variable x then $(P \to M)U$ is $(\lambda x.M)U$ which reduces to $\{U/x\}M$ by Lemma 2.2.
- If P is an atom A then $(P \to M)U$ reduces to $FU(\mathtt{eqatom}\, AUMI)(K(KI))$. When U is A this reduces to M. If U is any other matchable form then $(P \to M)U$ reduces to I.
- If P is an application P_1P_2 then $(P \to M)U$ reduces to

$$FUI(S(P_1 \to K(P_2 \to M))(K(KI)))\,.$$

If U is an atom then this reduces to I. Alternatively, if U is a matchable form U_1U_2 then this reduces to

$$S(P_1 \to K(P_2 \to M))(K(KI))U_1U_2$$

which reduces to $(P_1 \to K(P_2 \to M))U_1(KI)U_2$. Now if $\{U_1/P_1\}$ is $\mathtt{some}\, \sigma_1$ for some substitution σ_1 then the reduct becomes $\sigma_1(K(P_2 \to M))(KI)U_2$ which is $(P_2 \to \sigma_1 M)U_2$ since P_2 and P_1 do not share any free variables. In turn, if $\{U_2/P_2\} = \mathtt{some}\, \sigma_2$ for some substitution σ_2 then the combinator reduces to $\sigma_2(\sigma_1 M)$. Now, free variables in the range of σ_1 are also free in U,

and so not in the domain of σ_2. Thus, the result is $(\sigma_1 \uplus \sigma_2)M = \{U/P\}M$. Alternatively, if $\{U_2/P_2\} = \mathsf{none}$ then the result is I as required. Finally, if $\{U_1/P_1\} = \mathsf{none}$ then the result is $I(KI)U_2 = I$ as required.

$\dashv$

COROLLARY 8.3. *A confluent combinatory calculus that supports S, F and* eqatom *is structure complete if it has any normal forms.*

PROOF. If the calculus has a normal form then it has an atom A so that K can be defined to be FA. $\dashv$

It follows that all the calculi defined in Sections 4 and 6 are structure complete.

COROLLARY 8.4. *The calculi with operators SF or SKF or $SF\mathcal{C}$ or SFC, and the $\{S, F, \Omega\}$-calculus are all structure complete.*

COROLLARY 8.5. *Any symbolic computation restricted to normal forms of a structure complete, confluent combinatory calculus is representable.*

PROOF. The combinators S, K, F and eqatom are sufficient to redeploy the proof of Theorem 4.4. $\dashv$

Pattern-matching functions of the sort described here have been used to define *path polymorphic functions* [12] which traverse the internal structure of their arguments. This is achieved by recursively using the pattern $x \, y$ to represent an arbitrary compound. In the examples below, recursion is made implicit, and function arguments may be placed on the left-hand side of defining equations.

A familiar example is structural equality, which can be described by the pattern-matching function

$$
\begin{aligned}
&\mathsf{equal} = \\
&\quad x_1 \, x_2 \to (\ \ y_1 \, y_2 \to (\mathsf{equal} \ x_1 \ y_1) \ \mathsf{and} \ (\mathsf{equal} \ x_2 \ y_2) \\
&\qquad\qquad\qquad | \ y \to KI) \\
&\quad | \ x \to (\ \ y_1 \, y_2 \to KI \\
&\qquad\qquad | \ \mathsf{eqatom} \ x)
\end{aligned}
$$

THEOREM 8.6. *Let M and N be combinators in normal form. If $M = N$ then* $\mathsf{equal} \ M \ N \longrightarrow K$ *else* $\mathsf{equal} \ M \ N \longrightarrow KI$.

PROOF. The proof is by straightforward induction on the structure of M. $\dashv$

More generally, path polymorphism can be used to define generic queries that can select from, or update within, arbitrary structures. Since selecting requires a significant amount of list processing, the principles are better illustrated through updating. First, define $\mathsf{apply2all}$ by

$$
\begin{aligned}
&\mathsf{apply2all} \ f \ x = \\
&(\ \ y_1 \, y_2 \to (\mathsf{apply2all} \ f \ y_1) \ (\mathsf{apply2all} \ f \ y_2) \\
&\ | \ y \to y) \\
&\quad (f \ x) \, .
\end{aligned}
$$

The query $\mathsf{apply2all} \ f \ x$ recursively applies itself to the components of the result of applying f to x as a whole. Building on this, we can define the update combinator by

$$
\mathsf{update} \ t \ f = \mathsf{apply2all} \ (\lambda x. \ \mathsf{if} \ t \ x \ \mathsf{then} \ f \ x \ \mathsf{else} \ x) \, .
$$

$$\frac{}{O : T_O} \qquad \frac{M : U \to T \quad N : U}{MN : T} \qquad \frac{M : T}{M : \forall X.T} \qquad \frac{M : \forall X.T}{M : \{U/X\}T}$$

FIGURE 1. Typing SF-calculus

The basic path polymorphism of `apply2all` is used, but the function f is only applied when a test t is passed. Once lists have been defined, then it is equally easy to define a query `select` that produces a list of components of a structure satisfying some property.

§9. Typing. The operators S and K can be given simple types, built from some type constants and function types $T \to U$ that represent functions from T to U. Given types T, U and V then

$$S : (T \to U \to V) \to (T \to U) \to T \to V$$
$$K : T \to U \to T \ .$$

Unfortunately, the factorisation operator F does not have a simple type since the type of a compound does not determine the types of its components. Rather, some sort of existential type is required to describe the type of the second component, since this is not determined by the type of the compound as a whole. Existential type quantification can be represented by universally quantified types in System **F** [6]. Here X, Y and Z will denote type variables, so that $\forall X.T$ universally quantifies the variable X in the type T. Also, $\{U/X\}T$ substitutes U for free occurrences of X in T, in the usual manner. Now the factorisation operator has type

$$F : T \to U \to (\forall Z.(Z \to T) \to Z \to U) \to U$$

in which any function acting on the components must be polymorphic with respect to the unknown type Z of the second component. Given that quantifiers are in play, the operators S, K and F can be given the following closed types

$$S : \forall X.\forall Y.\forall Z.(X \to Y \to Z) \to (X \to Y) \to X \to Z$$
$$K : \forall X.\forall Y.X \to Y \to X$$
$$F : \forall X.\forall Y.X \to Y \to (\forall Z.(Z \to X) \to Z \to Y) \to Y \ .$$

We may write T_O for the type of the operator O. The complete set of type derivation rules is given in Figure 1. The first rule is a schema for the typing of the operators. The second rule types applications in the usual manner. The third and fourth rules implicitly introduce and eliminate type quantification. Note that there is no need for a context to record the types of term variables, since there are none to consider. Hence there is no need to impose side conditions on the introduction of type quantification, as in System **F**.

Computationally, this works very well, but may look a little odd from the viewpoint of logic. The function types can be interpreted as logical implications, but the use of quantified types for a premise is rather unusual, especially as Schönfinkel's original goal was to eliminate (bound) variables; perhaps the types need the combinatorial treatment, too. Additionally, the type of F does not look

very appealing as a logical axiom, say,

$$\frac{T \qquad U \qquad \forall Z.(Z \to T) \to Z \to U}{U}$$

since the conclusion U is already a premise. However, this defect already appears in the rule corresponding to K, namely

$$\frac{U \qquad V}{U}$$

so this is not a new phenomenon.

THEOREM 9.1. *Reduction of SF-calculus preserves typing.*

PROOF. It is enough to consider the reduction rules. Consider the typing of a combinator of the form $F(PQ)MN$ where PQ is a compound. A typing of the left-hand side must take the form

$$\frac{\begin{array}{l} F : V \to W \to \forall Z.(Z \to V) \to Z \to W \\ M : W \\ N : \forall Z.(Z \to V) \to Z \to W \end{array} \qquad \frac{P : U \to V \quad Q : U}{PQ : V}}{F(PQ)MN : W}.$$

Hence, the right-hand side of the rule can be typed by instantiating Z to U in the type of N to get $NPQ : W$. The other reduction rules are even easier to check. ⊣

§10. Conclusion. The ability to factorise combinators, to examine their internal structure, is simple and powerful, yet quite unexpected. Usually, such examination arises in a more operational setting, as when a Turing machine acts upon combinator syntax, but then ad-hoc techniques are necessary to ensure that the semantics given by combinator equality is respected. However, by giving a syntactic characterisation of the forms to be factored, factorisation can be made to respect the usual reduction and equality relations, and so be made into a symbolic computation. When this intensional expressive power is combined with that of extensionality then the resulting calculus is structurally complete, in that one is able to represent pattern-matching functions in which arbitrary normal forms are allowed as patterns. Examples include a generic test for structural equality of normal forms, and general forms of the queries popular in database programming. In turn, structure complete calculi are characterised by their support for four combinators, namely S (for duplication), K (for elimination), F (for factorisation) and `eqatom` for separating irreducible operators.

Since SF-calculus is a rewriting system, it is natural to ask about its denotational semantics. Dana Scott showed how to model pure λ-calculus using continuous lattices and then ω-complete partial orders [9]. However, it is not clear how to handle the examination of internal structure, i.e. what it means to factor elements of a partial order, or arrows in a *category* [19]. In mathematical logic, structural induction [2] is the analogue of factorisation, but the relationship has not been formalised.

Pattern calculi also support factorisation, albeit only for data structures. The relationship between pattern calculi and combinators may be strengthened by

considering matching of cases in the former and patterns that are not normal in the latter. In particular, this may confirm that pure pattern calculus cannot be represented in λ-calculus.

In addition to querying data structures, factorisation may have some relevance to program compilation and optimisation. A source program is, after all, an encoding of a function which is manipulated by a compiler before producing a "black box" executable. Just as SK-combinators can be used to compile the λ-abstractions in functional programming languages, SF-combinators may support a smoother treatment of data structures and pattern matching. Further, factorisation may help reveal program structure after compilation is finished, either during program execution, or as a form of reverse engineering.

The ability to factorise shows that symbolic computation is much richer than might be supposed from the study of SK-logic; that pattern matching adds significant new expressive power to the extensional expressive power of pure λ-calculus. Perhaps most important, it suggests that we continue the search for new and interesting combinators, and new logics for computing.

REFERENCES

[1] HENK P. BARENDREGT, *The lambda calculus. Its syntax and semantics*, Studies in Logic and the Foundations of Mathematics, Elsevier Science Publishers B.V., 1985.

[2] R. M. BURSTALL, *Proving properties of programs by structural induction*, **The Computer Journal**, vol. 12 (1969), no. 1, pp. 41–48.

[3] ALONZO CHURCH, *An unsolvable problem of elementary number theory*, **American Journal of Mathematics**, vol. 58 (1936), no. 2, pp. 345–363.

[4] H. B. CURRY and R. FEYS, **Combinatory logic**, Studies in Logic and the Foundations of Mathematics, vol. I, North-Holland, Amsterdam, 1958.

[5] H. B. CURRY, J. R. HINDLEY, and J. P. SELDIN, **Combinatory logic**, Studies in Logic and the Foundations of Mathematics, vol. II, North-Holland, Amsterdam, 1972.

[6] J-Y. GIRARD, Y. LAFONT, and P. TAYLOR, **Proofs and types**, Tracts in Theoretical Computer Science, Cambridge University Press, 1989.

[7] J.Y. GIRARD, *Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types*, **2nd Scandinavian logic symposium** (J.E. Fenstad, editor), Springer Verlag, 1971.

[8] THOMAS GIVEN-WILSON, **Interpreting the untyped pattern calculus in bondi**, Honours Thesis, University of Technology, Sydney, Australia, August 2007.

[9] C. A. GUNTER and D. S. SCOTT, *Semantic domains*, **Handbook of theoretical computer science** (J. van Leeuwen, editor), vol. B: Formal Models and Semantics, MIT Press, 1990.

[10] J. ROGER HINDLEY and JONATHAN P. SELDIN, **Introduction to combinators and λ-calculus**, Cambridge University Press, New York, NY, USA, 1986.

[11] GÉRARD HUET, *Confluent reductions: Abstract properties and applications to term rewriting systems*, **Journal of the ACM**, vol. 27 (1980), no. 4, pp. 797–821.

[12] BARRY JAY, **Pattern calculus: Computing with functions and data structures**, Springer, 2009.

[13] BARRY JAY and DELIA KESNER, *Pure pattern calculus*, **Programming languages and systems, 15th European symposium on programming, ESOP 2006** (P. Sestoft, editor), LNCS, vol. 3924, Springer, 2006, pp. 100–114.

[14] BARRY JAY and DELIA KESNER, *First-class patterns*, **Journal of Functional Programming**, vol. 19 (2009), no. 2, pp. 191–225.

[15] C.B. JAY, *The pattern calculus*, **ACM Transactions on Programming Languages and Systems (TOPLAS)**, vol. 26 (2004), no. 6, pp. 911–937.

[16] JOHN T. KEARNS, *Combinatory logic with discriminators*, **The Journal of Symbolic Logic**, vol. 34 (1969), no. 4, pp. 561–575.

[17] ———, *The completeness of combinatory logic with discriminators*, **Notre Dame Journal of Formal Logic**, vol. 14 (1973), no. 3, pp. 323–330.

[18] S.C. KLEENE, **Introduction to metamathematics**, North-Holland (originally published by D. Van Nostrand), 1952.

[19] J. LAMBEK and P.J. SCOTT, **Introduction to higher-order categorical logic**, Cambridge Studies in Advanced Mathematics, vol. 7, Cambridge University Press, 1986.

[20] JOHN McCARTHY, *Recursive functions of symbolic expressions and their computation by machine, Part I*, **Communications of the ACM**, vol. 3 (1960), no. 4, pp. 184–195.

[21] BARRY K. ROSEN, *Tree-manipulating systems and Church-Rosser theorems*, **Journal of the ACM**, vol. 20 (1973), no. 1, pp. 160–187.

[22] M. SCHÖNFINKEL, *Über die bausteine der mathematischen logik*, **Mathematische Annalen**, vol. 92 (1924), no. 3 - 4, pp. 305–316.

[23] A. TARSKI, **Logic, semantics, metamathematics**, Oxford University Press, 1956.

[24] ALAN M. TURING, *Computability and λ-definability*, **The Journal of Symbolic Logic**, vol. 2 (1937), no. 4, pp. 153–163.

[25] ERIC G. WAGNER, *Uniformly reflexive structures: On the nature of Gödelizations and relative computability*, **Transactions of the American Mathematical Society**, vol. 144 (1969), pp. 1–41.

UNIVERSITY OF TECHNOLOGY, SYDNEY
SYDNEY, AUSTRALIA
E-mail: {Barry.Jay,Thomas.Given-Wilson}@uts.edu.au