

A COMBINATORY ACCOUNT OF INTERNAL STRUCTURE

BARRY JAY AND THOMAS GIVEN-WILSON

Abstract. Traditional combinatory logic is able to represent all Turing computable functions on natural numbers, but there are effectively calculable functions on the combinators themselves that cannot be so represented, because they have direct access to the internal structure of their arguments. Some of this expressive power is captured by adding a factorisation combinator. It supports structural equality, and more generally, a large class of generic queries for updating of, and selecting from, arbitrary structures. The resulting combinatory logic is *structure complete* in the sense of being able to represent pattern-matching functions, as well as simple abstractions.

§1. Introduction. Traditional combinatory logic [21, 4, 10] is computationally equivalent to pure λ -calculus [3] and able to represent all of the Turing computable functions on natural numbers [23], but there are effectively calculable functions on the combinators themselves that cannot be so represented, as they examine the internal structure of their arguments.

This is consistent with the traditional theorems about computable functions of numbers, but imposes severe constraints upon the interpretation of Church's thesis, that all effectively calculable functions are general recursive [18]. The theoretical implications will be considered in Section 3, but the practical consequence is that the expressive power of traditional combinatory logic can be increased by adding new combinators.

Consider the combinators built from two *atoms* (meta-variable A), namely the traditional S and a new, factorisation operator F . The *SF-matchable forms* are the combinators of the form S, SM, SMN, F, FM and FMN . Then the defining equations for S and F are

$$\begin{aligned} SMNX &= MX(NX) \\ FAMN &= M \\ F(PQ)MN &= NPQ \text{ if } PQ \text{ is } SF\text{-matchable.} \end{aligned}$$

The traditional combinator K can be represented by FF since $KXY = FFXY = X$ but F cannot be represented by S and K . Just as the combinators S and K are able to support λ -abstraction, the combinators S and F are able to support a larger class of pattern-matching functions. That is, *SF*-combinators are *structure complete*.

Pattern-matching adds significant expressive power to combinatory logic. *SF*-logic is the first such to support a combinator for generic equality of normal

Our thanks to Roger Hindley and Samson Abramsky for their valuable comments on drafts of this work.

forms, obtained by comparing internal structures. Combinator equality has been considered indirectly by appealing to: meta-level operations [4, p. 245]; or partial combinatory algebras (not logics) such as the *uniformly reflexive structures* [24]; or *discriminators* [16, 17]. However, this is the first account we know of that is strictly within combinatory logic.

Further, one may define *generic queries* for selecting or updating structures that generalise the usual database queries from rows and tables to arbitrary structures. These queries are slightly more general than those of *pattern calculus* [15, 13, 14, 12] or Lisp [19] since they interact with arbitrary normal forms, while the latter are limited to *data structures* or *S-expressions*. Indeed, the present work has been motivated by the observation that the factorisation which is central to this new approach to pattern matching can be considered in isolation. It is worth emphasising that the novelty lies in the genericity of the queries, since traditional combinators are able to exploit internal structure when this is fixed in advance. For example, the internal structure of a number in unary arithmetic is accessed by the predecessor function, though the latter is surprisingly complex.

Unlike *S*, the factorisation operator 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, the missing information can be acknowledged by using existential quantification in System **F** of variable types [7, 6].

Although *SF*-combinatory logic is structure complete, there remain symbolic computations that it does not represent, which in turn suggest other novel combinators. Examples considered include a general equality *E*, and constructors for data structures, such as `Nil`, `Cons`, `Pair` etc. Another example arises by limiting factorisation to applications that are data structures.

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 introduces pattern-matching functions of combinators, the corresponding notion of structural completeness, and shows that *SF*-combinators are structurally complete. Section 6 shows how to type *F* using quantifiers. Section 7 introduces novel combinators for general equality, constructors, and factorising data structures. Section 8 draws conclusions and considers future work.

§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{A} of *atoms* (meta-variable *A*), or *operators* that are used to define the *combinators* (meta-variables *M, N, P, Q, R*) built from these by application

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

The \mathcal{A} -*combinatory calculus* or \mathcal{A} -*calculus* is given by the combinators plus their reduction rules. This section will focus on the traditional *SK*-calculus, with

atoms

$$A ::= S \mid K$$

and *reduction rules*

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

The combinator $SMNX$ uses X twice, as an 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 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{A} -*combinatory logic* is the system of equivalence classes of combinators from \mathcal{A} -combinatory calculus. When the distinction between the calculus and the logic is not important we may refer to the \mathcal{A} -*combinators*. Syntactic equality of combinators will be denoted by \equiv .

If the reduction relation cannot be applied then a combinator is a *normal form*. A combinator M is a *head normal form* if it satisfies two properties: it is not the instantiation of the left-hand side of any reduction rule; and if it is an application PQ then P is already a head normal form. For example, in SK -calculus, the head normal forms are those combinators of the form S, SM, SMN, K and KM .

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

$$\begin{aligned} \llbracket S \rrbracket &= \lambda g. \lambda f. \lambda x. g \ x \ (f \ x) \\ \llbracket K \rrbracket &= \lambda x. \lambda y. x \\ \llbracket MN \rrbracket &= \llbracket M \rrbracket \llbracket N \rrbracket. \end{aligned}$$

For example

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

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

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

⊣

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. Although it is tempting to define symbolic computations upon logics (where equality is the main notion) it is more convenient to work with calculi.

A *symbolic function* is here defined to be an n -ary partial function \mathcal{G} of some \mathcal{A} -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 $1 \leq i \leq n$ and $\mathcal{G}(X_1, X_2, \dots, X_n)$ is defined then $\mathcal{G}(X_1, X_2, \dots, X_n) = \mathcal{G}(Y_1, Y_2, \dots, Y_n)$. A combinator G in the calculus *represents* \mathcal{G} if there is a reduction

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

whenever the right-hand side is defined. Where a symbolic function is deemed to be effectively calculable it may be called a *symbolic computation*. Then \mathcal{G} is *representable* in \mathcal{A} -combinatory calculus. When \mathcal{G} is defined by the action of some operator A then we may say that G *represents* A . For example, if SK -calculus is augmented by an atom I with the rule

$$IY \longrightarrow Y$$

then I is represented (by I itself and) by any combinator of the form SKX since

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

In SK -calculus, I is defined to be SKK . Conversely, A is *independent* from a collection of atoms \mathcal{A} if A is not representable in \mathcal{A} -combinatory calculus.

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

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

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 is defined in the obvious manner, since the term calculus does not have any binding constructions built in. The \mathcal{A} -term calculus has reduction defined by the same rules as the \mathcal{A} -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 combinators is representable then the \mathcal{A} -combinatory logic is *combinatorially complete* in the sense of Curry [4, p. 5].

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

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

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

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

PROOF. Proof is straightforward 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 atom then $(\lambda^*x.M)N \equiv KMN \longrightarrow M \equiv \{N/x\}M$.
- Finally, if M is of the form M_1M_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. ⊢

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. ⊢

§3. Symbolic Computation. This section introduces some symbolic computations that are not defined by λ -abstraction, as they examine the internal structure of their arguments. After presenting an example, the implications for the general theory of computation will be discussed.

Consider some \mathcal{A} -calculus. Define the ternary function \mathcal{R} on combinators by

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

That is, \mathcal{R} branches according to whether it can *factorise* its first argument. Of course, \mathcal{R} does not respect equality since, for example, in SK -calculus the application $SKKK$ reduces to the atom K .

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 reduce the to head normal forms before factoring. These can be computed using

$$\begin{aligned} \mathcal{H}(X) &= X && \text{if } X \text{ is head normal} \\ \mathcal{H}(X) &= \mathcal{H}(Y) && \text{if } X \longrightarrow Y \text{ instantiates a rule} \\ \mathcal{H}(PQ) &= \mathcal{H}(\mathcal{H}(P)Q) && \text{otherwise.} \end{aligned}$$

Note that if X does not have a head normal form then $\mathcal{H}(X)$ is not defined. From this, one may define a ternary symbolic computation \mathcal{F} by

$$\mathcal{F}(X, M, N) = \mathcal{R}(\mathcal{H}(X), M, N) .$$

A routine induction shows that \mathcal{H} and \mathcal{F} preserve reduction and so are symbolic computations.

Each \mathcal{A} -calculus has its own collection of head normal forms and hence, its own version of \mathcal{F} . The symbol F shall be used to denote a combinator that represents \mathcal{F} .

THEOREM 3.1. *There are symbolic computations on SK-calculus that are not representable.*

PROOF. Clearly, \mathcal{F} is able to distinguish SKK from SKS . Now suppose that there is an SK -combinator F that represents \mathcal{F} . Then, for any combinator X we have

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

Translating this to λ -calculus yields both $\llbracket F(SKX)S(KI) \rrbracket \longrightarrow \llbracket X \rrbracket$ and

$$\llbracket F(SKX)S(KI) \rrbracket = \llbracket F \rrbracket \llbracket (SKX) \rrbracket \llbracket S \rrbracket \llbracket KI \rrbracket \longrightarrow \llbracket F \rrbracket (\lambda x.x) \llbracket S \rrbracket \llbracket KI \rrbracket .$$

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

It follows that, although SK -logic is combinatorially complete, it is not complete for combinators, in the sense of being able to represent all their symbolic computations. Since SK -logic is Turing complete, this sheds new light on our understanding of computation in general.

The 1930s witnessed an explosion of systems for supporting computation, including general recursive functions, Turing machines, combinatory logic and λ -calculus. These were all proven equivalent, in the sense of being able to support the same class of numerical functions, the general recursive ones, which led to Church's thesis, as expressed by Kleene [18]:

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

Of course, this statement cannot be proven, since there is no way to define the effectively calculable functions, but it captured the intuition that there is nothing beyond any of the systems mentioned above, and that, at least in principle,

it is enough to study any one. However, Theorem 3.1 shows that this intuition is wrong, and the following section will introduce combinators that add expressive power to combinatory calculus. So there is a gap between the theorems about numerical computation and the application of Church's thesis to symbolic computation. The gap is exposed by considering several interpretations of the thesis.

The most literal interpretation is that effective calculation is a property of numerical calculations, or that numbers are the only fit subject for calculation. This *Pythagorean* version of Church's thesis, since Pythagoras asserted that "All is number." If the thesis is intended to embrace symbolic and numerical computation then this interpretation must be amended.

A mechanistic interpretation is to confine computation to meaningless symbol pushing, e.g. by making marks upon a tape. This leads directly to Turing's thesis: anything that can be computed, can be computed on a Turing machine [23]. Of course, the ability to factorise is completely consistent with this thesis. However, though Turing's approach is good for establishing the existence of uncomputable functions, it does not say much about what can be computed.

A meaningful approach to Church's thesis requires a translation to general recursive functions that preserves the semantics. For numerical calculations the semantics is quite familiar, but for symbolic computation the semantics may be alien. [According to Richard Dedekind, however, such a reality check is not available for symbolic systems since "God made the natural numbers. Everything else is the work of man."] Rather than appeal to some external or denotational semantics, it is safer to preserve an operational semantics, e.g. equality or reduction of combinators. However, once the translation becomes explicit then two difficulties arise.

First, it is not clear how to choose the translation, even for familiar systems. For example, consider combinators for unary arithmetic, a zero and successor. One translation to general recursion will map the zero to 0, but a translation through pure λ -calculus will map the zero to the identity function.

Second, there may fail to be any translation that preserves the semantics. The existing translation from *SK*-logic to λ -calculus serves well enough for numerical computations, but does not provide a representation of \mathcal{F} . As \mathcal{F} is a meta-function, in the sense of Tarski [22], it is tempting to translate combinators to binary trees and then use tree operations to factorise. However, it is not clear what semantics is being preserved. Further, one may consider effective calculations that act upon meta-function \mathcal{F} , leading to meta-meta-functions, whose translations will be even more opaque. Of course, these hierarchical difficulties are well known in the context of logical paradoxes, such as that of Russell.

Finally, a narrow interpretation can escape the need for translation by restricting to a single system. However, it is still necessary to account for symbolic computations. *SK*-logic appeared to achieve this by being combinatorially complete but, as shown above, it has symbolic computations that cannot be represented.

To summarise, Church's thesis, as stated by Kleene, is plausible if one restricts attention to numerical computation (the Pythagorean approach), or abandons semantics (the Turing approach), but is implausible when semantics-preserving

translations are made explicit, and false when considering completeness of SK -combinators for symbolic computation. In particular, there are plenty of opportunities to add expressive power by introducing new combinators, as will now be shown.

§4. Factorisation. This section introduces SF -calculus, where F will be used to represent \mathcal{F} , as defined in Section 3. Basic properties are established, and examples culminating in a combinator for structural equality.

The definition of \mathcal{F} depends upon the notion of head normal form, which is parametrised by the reduction rules of the calculus. However, this approach cannot be applied to define F as its reduction rules must be given before determining the head normal forms. Fortunately, in each calculus considered, it is easy to give a syntactic characterisation of the head normal forms, as the matchable forms.

The SF -calculus has *matchable forms* given by

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

A *compound* is a matchable form that is also an application. Its *reduction rules* are

$$\begin{array}{lll} SMNX & \longrightarrow & MX(NX) \\ FAMN & \longrightarrow & M \quad \text{for any atom } A \\ F(PQ)MN & \longrightarrow & NPQ \quad \text{if } PQ \text{ is a compound .} \end{array}$$

The combinator S is as before. The factorisation operator branches according to whether its first argument is an atom, in which case the second argument results, or is a matchable application, in which case the third argument is applied to the components. The combinator K is defined to be $K \equiv FF$ and then $I \equiv SKK$ as before.

THEOREM 4.1. *Reduction is confluent.*

PROOF. It is enough to observe that the reduction rules are orthogonal [20, 11], since matchable forms are stable under reduction. \dashv

THEOREM 4.2. *Every normal form is a matchable form.*

PROOF. Trivial. \dashv

Despite this result, note that there are (equivalence classes of) combinators in the corresponding combinatory logic which do not have a matchable form, such as $FXMN$ where X does not normalise.

Here are some examples. The presentation will exploit the λ^* -abstraction defined in Section —refsec:combination. Just as in SK -calculus, it is convenient to introduce some familiar computing constructs. Define the conditionals, of the form **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 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 and $M(\mathbf{fix} M)$ have a common reduct.

The test for being a compound is

$$\text{isComp} \equiv \lambda^*x.Fx(KI)(K(KK))$$

since

$$\begin{aligned} \text{isComp } A &\longrightarrow FA(KI)(K(KK)) \\ &\longrightarrow KI \\ \text{isComp}(PQ) &\longrightarrow F(PQ)(KI)(K(KK)) \\ &\longrightarrow K(KK)PQ \\ &\longrightarrow KKQ \\ &\longrightarrow K \quad \text{if } PQ \text{ is a compound.} \end{aligned}$$

Similarly, the first and second components of a compound are obtained by

$$\begin{aligned} \text{fstComp} &\equiv \lambda^*x.FxIK \\ \text{sndComp} &\equiv \lambda^*x.FxI(KI). \end{aligned}$$

Note that they map atoms to I , so it is usual to check for being a compound first.

It is also useful to be able to distinguish F from S . Define

$$\text{isF} \equiv \lambda^*x.x(KI)(K(KI))K.$$

Then

$$\begin{aligned} \text{isF } F &\longrightarrow F(KI)(K(KI))K \longrightarrow KKI \longrightarrow K \\ \text{isF } S &\longrightarrow S(KI)(K(KI))K \longrightarrow KIK(K(KI)K) \longrightarrow KI \end{aligned}$$

as desired. This yields a test for equality of atoms, namely

$$\text{eqatom} \equiv \lambda^*x.\lambda^*y.\text{isF } x \text{ implies isF } y.$$

Putting the tests for compounds and atoms together yields a combinator for equality of normal forms, namely

$$\begin{aligned} \text{equal} &\equiv \text{fix } (\lambda e.\lambda x.\lambda y. \\ &\quad \text{if isComp } x \\ &\quad \text{then isComp } y \text{ and} \\ &\quad \quad e (\text{fstComp } x) (\text{fstComp } y) \text{ and } e (\text{sndComp } x) (\text{sndComp } y) \\ &\quad \text{else not (isComp } y) \text{ and eqatom } x y). \end{aligned}$$

It tests equality of combinators x and y as follows. If x is a compound then check that y is too, and that their corresponding components are equal. Otherwise, ensure that y is an atom and apply eqatom . The combinator equal is an example of a *path polymorphic function* or *query* [12] in that it may traverse all paths through the internal structure of its arguments.

§5. Structure Completeness. This section shows that SF -calculus is able to represent pattern-matching functions. The resulting syntax provides an attractive account of equality, and other generic queries.

Fix a collection of atoms \mathcal{A} and their reduction rules. The *patterns* (meta-variable P) are terms of the \mathcal{A} -term calculus that are in normal form. For example, in *SF*-term calculus they can be described by

$$\begin{aligned} P_v &::= x \mid P_v P \\ P &::= P_v \mid S \mid SP \mid SPP \mid F \mid FP \mid FPP . \end{aligned}$$

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 using whatever flavour of equality suits best, as discussed in Section 7.

Patterns can be used to define *cases* which are symbolic functions satisfying equations of the form

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

where P is a pattern and M is an arbitrary term. When all free variables of M are also free in P then \mathcal{G} is a function of combinators, as well as of terms. If, further, P is a variable x then \mathcal{G} is definable in a combinatorially complete calculus, but in general this is not so. For example $\mathcal{G}(x y) = N x y$ captures some of the functionality of F .

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 consider matching on syntax. Now there is at most one substitution σ such that $\sigma P \equiv U$, but other concerns arise when U reduces to some U' . For \mathcal{G} to yield a function of the logic, it is then necessary that there be a substitution σ' (equivalent to σ) such that $\sigma' P \equiv U'$. To define matching in this manner, it is sometimes necessary to restrict the argument U to be a matchable form. Hence, it is simplest to give a procedure for matching, rather than a characterisation like the one above.

Define a *match* to be either a *successful match*, **some** σ where σ is a substitution, or a *match failure*, **none**. The application of a match to a term is defined by

$$\begin{aligned} \text{some } \sigma \quad M &= \sigma M \\ \text{none} \quad M &= I . \end{aligned}$$

For definiteness, match failure must produce a combinator; the identity proves to be a useful choice when defining extensions, below. 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 *match* $\{U/P\}$ of a pattern P against a combinator U is defined by

$$\begin{aligned} \{U/x\} &= \text{some } \{U/x\} \\ \{A/A\} &= \text{some } \{\} \\ \{UV/PQ\} &= \{U/P\} \uplus \{V/Q\} \quad \text{if } UV \text{ is matchable} \\ \{U/P\} &= \text{none otherwise} \quad \text{if } U \text{ is matchable} \\ \{U/P\} &= \text{undefined} \quad \text{otherwise.} \end{aligned}$$

The restrictions to matchable forms are necessary to ensure confluence. 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 matchable form.

Now the evaluation of \mathcal{G} above is given by

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

whenever the right-hand side is defined.

A combinatory calculus is *structure complete* if every case $\mathcal{G}(P) = M$ is representable.

The combinators S and F almost suffice for structure completeness, but in general there remains the problem of identifying atoms. For each atom A one can define a symbolic computation $\mathcal{I}(A)$ by

$$\mathcal{I}(A)(X) = \begin{cases} K & \text{if } \mathcal{H}(X) \equiv A \\ KI & \text{otherwise, if } \mathcal{H}(X) \text{ is defined.} \end{cases}$$

The notation $\text{is}(A)$ is used to denote a representative for $\mathcal{I}(A)$. It satisfies the rules

$$\begin{aligned} \text{is}(A) A &\longrightarrow K \\ \text{is}(A) X &\longrightarrow KI \quad \text{if } X \text{ is matchable.} \end{aligned}$$

Consider a combinatory calculus that *supports* S, F and $\text{is}(A)$ for each atom A , in the sense that \mathcal{F} and all the $\mathcal{I}(A)$ are representable. For each pattern P and term M , there is a term $P \rightarrow M$ such that $(P \rightarrow M)U \longrightarrow \{U/P\}M$ whenever the right-hand side exists. It is defined by induction on the structure of P , employing a fresh variable x , as follows:

- If P is a variable y then $P \rightarrow M$ is $\lambda^*y.M$.
- If P is an atom A then $P \rightarrow M$ is $\lambda^*x.\text{is}(A) x MI$.
- If P is an application P_1P_2 then $P \rightarrow M$ is

$$\lambda^*x.F x I (S(P_1 \rightarrow K(P_2 \rightarrow M))(K(KI)))$$

THEOREM 5.1. *Any \mathcal{A} -calculus that supports S, F and $\text{is}(A)$ for each atom A is structure complete.*

PROOF. Every case \mathcal{G} defined by $\mathcal{G}(P) = M$ is represented by $P \rightarrow M$. The proof is by induction on the structure of P . Let U be a combinator such that $\mathcal{G}(U)$ is defined and consider $(P \rightarrow M)U$.

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

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

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

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

which reduces to $(P_1 \rightarrow K(P_2 \rightarrow M))U_1(KI)U_2$. Now if $\{U_1/P_1\}$ is **some** σ_1 for some substitution σ_1 then it becomes $\sigma_1(K(P_2 \rightarrow M))(KI)U_2$ which is $(P_2 \rightarrow \sigma_1M)U_2$ since P_2 and P_1 do not share any free variables. In turn, if $\{U_2/P_2\} = \text{some } \sigma_2$ for some substitution σ_2 then the combinator reduces to $\sigma_2(\sigma_1M) = (\sigma_1 \uplus \sigma_2)M = \{U/P\}M$. Alternatively, if $\{U_2/P_2\} = \text{none}$ then the result is I as required. Finally, if $\{U_1/P_1\} = \text{none}$ then the result is $K(KI)U_1U_2$ which reduces to I as required.

–

COROLLARY 5.2. *SF-calculus is structure complete.*

PROOF. *SF-calculus* supports $\text{is}(S)$ and $\text{is}(F)$ by using isComp to exclude compounds and then applying isF , as defined in Section 4. Now apply the theorem.

–

Pattern-matching functions are defined using a sequence of cases, as in

$$\begin{aligned} \mathcal{G} = & \\ & | P_1 \rightarrow M_1 \\ & | P_2 \rightarrow M_2 \\ & \dots \\ & | P_n \rightarrow M_n . \end{aligned}$$

When applied to some argument, the function part reduces to the first case $P_i \rightarrow M_i$ 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 of cases using *extensions* [15, 12]. In the combinatory setting, the *extension* of a combinator M (the *default*) by a *special case* consisting of a pattern P and a body M is given by

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

When applied to some term U such that $\{U/P\}$ results in some σ then

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

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

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

The following examples of generic queries exploit the new pattern-matching syntax (and implicit recursion). Structural equality can now be given by

$$\begin{aligned} \text{equal} = & \\ & x_1 x_2 \rightarrow (y_1 y_2 \rightarrow (\text{equal } x_1 y_1) \text{ and } (\text{equal } x_2 y_2) \\ & \quad \mid y \rightarrow KI) \\ & \mid x \rightarrow (y_1 y_2 \rightarrow KI \\ & \quad \mid y \rightarrow \text{eqatom } x y) . \end{aligned}$$

Perhaps the most primitive query is `apply2all` defined by

$$\begin{aligned} \text{apply2all } f x = & \\ & (y_1 y_2 \rightarrow (\text{apply2all } f y_1) (\text{apply2all } f y_2) \\ & \quad \mid y \rightarrow y) \\ & (f x) . \end{aligned}$$

The query `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

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

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.

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

$$\begin{aligned} S &: (T \rightarrow U \rightarrow V) \rightarrow (T \rightarrow U) \rightarrow T \rightarrow V \\ K &: T \rightarrow U \rightarrow T . \end{aligned}$$

Unfortunately, the 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

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

$$\frac{}{A : T_A} \quad \frac{M : U \rightarrow T \quad N : U}{MN : T} \quad \frac{M : \forall X.T}{M : \{U/X\}T} \quad \frac{M : T}{M : \forall X.T}$$

FIGURE 1. Typing Factor Calculus

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

$$\begin{aligned} S &: \forall X.\forall Y.\forall Z.(X \rightarrow Y \rightarrow Z) \rightarrow (X \rightarrow Y) \rightarrow X \rightarrow Z \\ K &: \forall X.\forall Y.X \rightarrow Y \rightarrow X \\ F &: \forall X.\forall Y.X \rightarrow Y \rightarrow (\forall Z.(Z \rightarrow X) \rightarrow Z \rightarrow Y) \rightarrow Y. \end{aligned}$$

We may write T_A for the type of the operator A . 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 eliminate and introduce 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,

$$\frac{T \quad U \quad \forall Z.(Z \rightarrow T) \rightarrow Z \rightarrow U}{U}$$

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

$$\frac{U \quad V}{U}$$

so this is not a new phenomenon.

§7. Variations. This section introduces three related calculi that support a general equality, constructors, and the restriction of factorisation to data structures. All three are confluent, with matchable forms proving to be the head normal forms. The first two are structure complete by Theorem 5.1.

General Equality. The structural equality of Section 4 is sufficient for normal forms, but if equality of arbitrary combinators is desired then a general equality combinator E can be introduced. Define the SFE -calculus with with matchable forms

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

where S and F are as usual, and E that satisfies the rules

$$\begin{aligned} EXX &\longrightarrow K \\ EA(PQ) &\longrightarrow KI \text{ if } PQ \text{ is matchable} \\ E(PQ)A &\longrightarrow KI \text{ if } PQ \text{ is matchable} \\ E(P_1Q_1)(P_2Q_2) &\longrightarrow EP_1P_2 \text{ and } EQ_1Q_2 \text{ if } P_1Q_1 \text{ and } P_2Q_2 \text{ are matchable.} \end{aligned}$$

Thus, E allows any term to match itself even if it does not have a normal form. Note, however, that if M and N are unequal and M does not head normalise then their equality EMN will be undecidable.

Constructors. A *constructor* is an atom that does not appear at the head of any reduction rule. Typical examples are `Pair` for building pairs, or `Nil` for the empty list. Let \mathcal{C} be a collection of atoms, each of which is either a constructor (meta-variable C) or an operator $\text{is}(C)$ that represents $\mathcal{I}(C)$ as described in Section 5. The constructors can be 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. Note that all data structures are head normal forms.

Define the *SFC*-calculus with matchable forms

$$d \mid S \mid SM \mid SMN \mid F \mid FM \mid FMN \mid \text{is}(C)$$

which now include all the data structures. The reduction rules for S , F and $\text{is}(C)$ are as usual.

Note that any countable collection of constructors can be encoded as data structures built from a single constructor C , as $CC, C(CC), C(CCC)$ etc. Further, in *SFC*-calculus, $\text{is}(C)$ can be defined by first checking for compounds and then applying

$$\text{is}C \equiv \lambda^*x.F (x KKK)(KI)(K(KK))$$

to any remaining atoms. Structure completeness follows once the accounts of $\text{is}(S)$ and $\text{is}(F)$ in Section 4 have been modified to handle C .

Factorising Data Structures. The main motive for developing factorisation combinators was to rework pattern calculi [15, 13, 14, 12] in a combinatory setting. However, there is a subtle difference between existing calculi for patterns and combinators: patterns which are cases never match, but their corresponding combinators may. For example, the identity function in (static) pattern calculus is given by the case $x \rightarrow x$ which, being a case, does not match itself. However, it translates to the combinator I which does match itself. There are two ways of eliminating the tension. One is to add case matching to pattern calculus. While feasible, keeping track of the nature of the different variables will be a burden. The other, adopted here, is to replace F by a weaker operator D that factorises data structures only. Now, the matchable forms of the *SDC*-calculus are

$$d \mid S \mid SM \mid SMN \mid D \mid DM \mid DMN \mid \text{is}(C)$$

where the data structures are defined as above. The complete reduction rules are

$$\begin{array}{llll}
 SMNX & \longrightarrow & MX(NX) & \\
 D(PQ)MN & \longrightarrow & NPQ & \text{if } PQ \text{ is a data structure} \\
 DXMN & \longrightarrow & M & \text{otherwise, if } X \text{ is matchable} \\
 \text{is}(C) C & \longrightarrow & K & \\
 \text{is}(C) X & \longrightarrow & KI & \text{otherwise, if } X \text{ is matchable.}
 \end{array}$$

Although, it is routine to show that *SDC*-calculus is to *static pattern calculus* [8, 12] as *SK*-calculus is to the λ -calculus, this would require a detailed account of static pattern calculus. We intend to cover all of this within a full treatment of the relationship between pattern calculi and combinatory logic.

§8. Conclusion. To factorise combinators within combinatory logic, to examine their internal structure, is simple and powerful, yet quite unexpected. Usually, such examination arises in more operational settings, in which compilers act upon source code, or evaluators adopt a fixed strategy, as in a Turing machine. Then ad hoc techniques are necessary to ensure that the semantics is respected. When this approach is adopted in theory the equality of combinators is unacceptably weakened, as with discriminators. However, by identifying the matchable forms, factorisation can be made to respect the usual reduction and equality relations, yielding a meaningful symbolic computation. Adding factorisation is enough to ensure that pattern-matching functions can be represented, as well as the usual λ -abstractions. This is enough to support structural equality and generic queries, as pioneered in pattern calculus.

The existence of symbolic computations that cannot be represented using traditional *SK*-combinators challenges our understanding of Church's thesis, as stated by Kleene. In particular, the thesis can no longer be used to discourage the search for new and interesting combinators or new ways of computing. Rather, it should be seen as a statement about numerical computation, or replaced by Turing's thesis about the mechanics of computation, as Church's thesis does not impose meaningful limits upon symbolic computation.

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 (and hence *SK*-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. In mathematical logic, structural induction [2] is the analogue of factorisation, but the relationship has not been formalised.

Pattern calculus also supports factorisation, albeit only for data structures. Future work will show how to translate the existing dynamic pattern calculus [14, 12] to compound calculus [8, 12] (a calculus related to Lisp), and thence to a combinatory calculus. Conversely, factorisation indicates that it is possible to match abstractions in a more general pattern calculus, so that the focus shifts from data structures to normal forms. These translations should imply that 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. Factorisation may help reveal some of the program structure after compilation is finished, either during program execution, or as a form of reverse engineering.

As well as the factorisation operator F , one can add a general equality E , constructors and factorisation for data structures. There is no reason to think that this exhausts the possibilities for novel combinators.

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, BAR h 85:1 1.Ex.
- [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*, vol. I, North-Holland, Amsterdam, 1958.
- [5] H. B. CURRY, J. R. HINDLEY, and J. P. SELDIN, *Combinatory logic*, 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’interprétation de Gödel à l’analyse, et son application à l’élimination des coupures dans l’analyse et la théorie des types*, *2nd scandinavian logic symp.* (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*, *J. 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, held as part of the joint european conferences on theory and practice of software, etaps 2006, vienna, austria, march 27–28, 2006, proceedings (ed: P. sestoft)*, 2006, Revised version at www-staff.it.uts.edu.au/~cbj/Publications/purepatterns.pdf, pp. 100–114.
- [14] BARRY JAY and DELIA KESNER, *First-class patterns*, *Journal of Functional Programming*, vol. 19 (2009), no. 2, p. 34 pages.
- [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] JOHN MCCARTHY, *Recursive functions of symbolic expressions and their computation by machine, part I*, **Commun. ACM**, vol. 3 (1960), no. 4, pp. 184–195.
- [20] BARRY K. ROSEN, *Tree-manipulating systems and Church-Rosser theorems*, **J. ACM**, vol. 20 (1973), no. 1, pp. 160–187.
- [21] M. SCHÖNFINKEL, *Über die bausteine der mathematischen logik*, **Mathematische Annalen**, vol. 92 (1924), no. 3 - 4, pp. 305–316.
- [22] A. TARSKI, *Logic, semantics, metamathematics*, **Intentions in communication**, Oxford University Press, 1956, pp. 325–363.
- [23] ALAN M. TURING, *Computability and lambda-definability*, **The Journal of Symbolic Logic**, vol. 2 (1937), no. 4, pp. 153–163.
- [24] 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: {cbj,tgwilson}@it.uts.edu.au