Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 325 (2016) 221–236

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Programs as Data Structures in λSF-Calculus

Barry Jay

Centre for Quantum Computing & Intelligent Systems
School of Software
University of Technology Sydney
Sydney
Australia

Abstract

Lambda-SF-calculus can represent programs as closed normal forms. In turn, all closed normal forms are data structures, in the sense that their internal structure is accessible through queries defined in the calculus, even to the point of constructing the Goedel number of a program. Thus, program analysis and optimisation can be performed entirely within the calculus, without requiring any meta-level process of quotation to produce a data structure.
Lambda-SF-calculus is a confluent, applicative rewriting system derived from lambda-calculus, and the combinatory SF-calculus. Its superior expressive power relative to lambda-calculus is demonstrated by the ability to decide if two programs are syntactically equal, or to determine if a program uses its input. Indeed, there is no homomorphism of applicative rewriting systems from lambda-SF-calculus to lambda-calculus.
Program analysis and optimisation can be illustrated by considering the conversion of a programs to combinators. Traditionally, a program p is interpreted using fixpoint constructions that do not have normal forms, but combinatory techniques can be used to block reduction until the program arguments are given. That is, p is interpreted by a closed normal form M. Then factorisation (by F) adapts the traditional account of lambda-abstraction in combinatory logic to convert M to a combinator N that is equivalent to M in the following two senses. First, N is extensionally equivalent to M where extensional equivalence is defined in terms of eta-reduction. Second, the conversion is an intensional equivalence in that it does not lose any information, and so can be reversed by another definable conversion. Further, the standard optimisations of the conversion process are all definable within lambda-SF-calculus, even those involving free variable analysis.
Proofs of all theorems in the paper have been verified using the Coq theorem prover.

Keywords: lambda-calculus, SF-calculus, self-interpretation, xi-rule

1 Introduction

λ-calculus [1] provides a completely general account of the extensional behaviour of functions, of all that can be discovered by evaluating them. This may be enough for applications, but the implementation of programming languages requires access to the internal structure of programs. As this is not possible from within the pure λ-calculus, meta-level analysis is commonly required. For example, self-interpretation

[1] Thanks to Thomas Given-Wilson, Neil Jones, Jens Palsberg and Jose Vergara for helpful discussions as this work gestated, and the anonymous referees.

[2] Email:Barry.Jay@uts.edu.au

http://dx.doi.org/10.1016/j.entcs.2016.09.040

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

$$\begin{aligned}
(\lambda x.M)N &\longrightarrow \{N/x\}M \\
SMNP &\longrightarrow MP(NP) \\
FOMN &\longrightarrow M \qquad (O \text{ is } S \text{ or } F) \\
FPMN &\longrightarrow NP\rceil\,\lceil P \quad (P \text{ is a compound of } P\rceil \text{ and } \lceil P).
\end{aligned}$$

Fig. 1. λSF-calculus

of λ-calculus [12,17,2,14,15,4,3,18,9], usually begins by applying a meta-function quote which converts an arbitrary λ-term into a data structure, whose internal structure can be queried at will.

Recent work suggests an alternative approach, using calculi that support a more general class of queries. *Pure pattern calculus* [8,5] uses pattern matching to define *generic queries* of data structures built from arbitrary constructors. However, it is unable to analyse pattern-matching functions themselves. *SF-calculus* [7] can query any closed normal form by using its operator F to reveal its internal structure, e.g. the components P_1 and P_2 of a closed normal application P_1P_2. However, it does not provide first-class support for λ-abstraction or any other mechanism for binding variables.

This paper shows how to factorise abstractions in a new calculus, the λSF-calculus, by converting them to combinators when it is safe to do so, i.e. when this will not break any redexes in the body of the abstraction. The syntax and reduction rules of λSF-calculus are just those of λ-calculus and SF-calculus, as given in Figure 1, on the understanding that the compounds now include some abstractions as well as some applications. The result is a proper extension of λ-calculus in the sense that there is *no* function from λSF-calculus to λ-calculus that preserves its structure as an applicative rewriting system.

This expressive power can be used to support arbitrary queries of closed normal forms. In this sense, we can identify the *data structures* with the closed normal forms. What about programs? The standard interpretation of programs does *not* yield normal forms since recursion is modeled by a fixpoint function that does not have a normal form. However, traditional combinators can be used to identify programs, even recursive ones, with closed normal forms. Hence, we can identify the programs with closed normal forms, to get

$$\text{programs} = \text{closed normal forms} = \text{data structures}.$$

That is, programs can be represented by terms that are simultaneously functions, ready to act on arguments, and data structures, ready for analysis and optimisation, and this without any need for quotation. Except when justifying this equation, we will identify the programs with the closed normal forms.

This provides a more flexible foundation for computation than any of the traditional models, as these emphasise only one aspect of a program's nature. In particular, λ-calculus emphasises its functional aspect, while Turing machines emphasise

its structure, as a string of symbols on a tape. This new flexibility suggests fresh approaches to many issues in theory and practice, especially the implementation of programming languages.

The structure of the paper is as follows. Section 1 is the introduction. Section 2 introduces λSF-calculus and its basic properties. Section 3 shows that equality of programs is definable. Section 4 defines extensional equivalence. Section 5 shows that there is no homomorphism of applicative rewriting systems from λSF-calculus to λ-calculus. Section 6 show how to represent recursive programs as closed normal forms. Section 7 converts programs to extensionally equivalent combinators. Section 8 optimises the conversion by program analysis. Section 9 converts programs to combinators in a way that preserves intensions as well as extensions. Section 9 discusses the proof verifications in Coq. Section 10 suggests some fresh approaches to existing issues. Section 11 draws conclusions.

2 λSF-calculus

The terms and reduction rules of λSF-calculus are given in Figure 1. The terms (meta-variables $M, N, P, \ldots$ consist of *variables* $x, y, z, \ldots, f, g, \ldots$, the *operators* S and F, *abstractions* $\lambda x.M$ with bound variable x and *body* M, and *applications* MN of M to N. The reduction rules for λ and S are standard. The rules for F have the same high-level semantics as in SF-calculus in that F branches according to whether its first argument P is an *atom*, i.e. an operator, or a *compound*. If P is an atom then return the first branch: if P is a compound then apply the second branch to its two components. The intention is that the compounds are terms whose decomposition into components does not break any redexes. They are, in a sense, head normal forms. The technical point is that there is a syntactic test for this property, even in the presence of abstractions. The reflexive, transitive closure of $\longrightarrow$ is denoted $\longrightarrow^*$.

2.1 Compounds

In combinatory calculi, the compounds are all the partially applied operators. For example, in SF-calculus, the compounds are all terms of the form SM or SMN or FM or FMN. These forms are compounds in λSF-calculus, too. All other compounds of λSF-calculus are abstractions $\lambda x.M$ whose decomposition is safe because either M is already an atom or compound, or outermost reduction in M awaits the instantiation of x, i.e. x is *active* in M in the following sense.

Define the set active(M) of *active variables* of a term M to be a set that has at most one element, that is defined by the pattern-matching function in Figure 2 (active$(M) - \{x\}$ removes x from active(M)).

Here are some examples of compounds. The body of $\lambda x.x\ y$ has x active. The body of $\lambda x.\lambda y.x$ has x active. The body of $\lambda x.\lambda y.y$ is a compound. The body of $\lambda x.F$ is an atom. The body of $\lambda x.Fx$ is a compound. The body of $\lambda x.FxM$ is a compound. The body of $\lambda x.FxMN$ has x active, since F is an intensional operator that needs to know the value of x to reduce. The body of $\lambda x.\lambda y.F(FxMN)PQ$ has

$$
\begin{array}{l}
\text{active} = \\
\quad \mid x \Rightarrow \{x\} \\
\quad \mid O \Rightarrow \{\,\} \\
\quad \mid \lambda x.M \Rightarrow \text{active}(M) - \{x\} \\
\quad \mid OM \Rightarrow \{\,\} \\
\quad \mid OMN \Rightarrow \{\,\} \\
\quad \mid SMNP \Rightarrow \{\,\} \\
\quad \mid FMNP \Rightarrow \text{active } M \\
\quad \mid MN \Rightarrow \text{active } M \quad \text{otherwise.}
\end{array}
$$

Fig. 2. Active Variables

x active.

2.2 Star Abstraction

The decomposition of an abstraction $\lambda x.M$ will use the *star abstraction* $\lambda^* x.M$ of M with respect to x. This is an adaptation of the standard technique for defining the abstraction of a combinator M with respect to a variable. Since this is defined using the combinators S, K and I, the latter two must be defined in terms of S and F, as follows. Define

$$K = FF$$

so that $KMN = FFMN \longrightarrow M$ for any choice of M and N. Then define

$$I = SKK$$

so that $IM = SKKM \longrightarrow KM(KM) \longrightarrow M$ for any M.

The *star abstraction* $\lambda^* x.M$ of M with respect to x is defined by

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

This definition modifies the traditional definition of $\lambda^* x.M$ for combinators M in two ways. First, when the body is an application MN the result uses $\lambda x.N$ instead of $\lambda^* x.N$. To see why this is necessary, consider $\lambda^* x.F(KN_1N_2)$. Now $F(KN_1N_2)$ is a compound, so it is safe to separate F from KN_1N_2 but $\lambda^* x.KN_1N_2$ breaks the redex KN_1N_2 so a recursive call to $\lambda^* x$ would here be unsafe. Second, there needs to be a rule for $\lambda^* x$ when the body is an abstraction $\lambda y.M$. The result is $\lambda x.\lambda^* y.M$ and not $\lambda^* x.\lambda^* y.M$ since it is important that only one abstraction is eliminated at a time, namely, the innermost one.

Here are some simple examples of star abstraction. In SKI-calculus, the λ-abstraction $\lambda x.\lambda y.y$ can be represented by

$$\lambda^* x.\lambda^* y.y = \lambda^* x.I = KI$$

where λ^* is used to convert abstractions into combinators in the traditional manner. In λSF-calculus, the $\lambda x.\lambda y.y$ is already a closed normal form. However, its

factorisation will introduce $\lambda^* x.\lambda y.y$ which is calculated as follows:

$$\lambda^* x.\lambda y.y = \lambda x.\lambda^* y.y = \lambda x.I \ .$$

This has eliminated the innermost abstraction, just like the first step in the calculation of $\lambda^* x.\lambda^* y.y$ in SKI-calculus. A second factorisation exposes

$$\lambda^* x.SKK = S(\lambda x.SK)(\lambda x.K) \ .$$

Further factorisation eliminates the remaining abstractions to produce the combinator

$$S(S(KS)(S(KF)(KF)))(S(KF)(KF))$$

which when applied to terms M and N reduces to N, just like the original abstraction. Of course, it is much bigger than the original term, as it does not take advantage of the standard optimisation, in which $\lambda^* x.I$ takes advantage of the fact that x is not free in I to produce KI. This will be addressed in Section 8.

2.3 Components

The *left component* $M\rceil$ of a term M is defined as follows

$$\begin{aligned} (MN)\rceil &= M \\ M\rceil &= \mathsf{abs\_left} \quad \text{(otherwise)} \end{aligned}$$

where $\mathsf{abs\_left} = SKF$ will be used as the left component of any term that is not an application, especially of any abstraction. The key point about $\mathsf{abs\_left}$ is that it cannot be the left component of an application to some N since $\mathsf{abs\_left}\ N = SKFN$ is a fully applied instance of S. In general, words in sans-serif, such a $\mathsf{abs\_left}$ may be used to name particular terms of λSF-calculus, as well as the meta-variables M and N, etc.

Now the *right component* $\lceil M$ of M is defined by

$$\begin{aligned} \lceil(MN) &= N \\ \lceil(\lambda x.M) &= \lambda^* x.M \\ \lceil M &= M \quad \text{(otherwise.)} \end{aligned}$$

It follows that if M is a compound and $M \longrightarrow N$ then $M\rceil \longrightarrow N\rceil$ and $\lceil M \longrightarrow \lceil N$. That is, no redexes are broken by taking components of compounds. To put is another way, there is a derived reduction rule

$$(\xi)\ \frac{M \longrightarrow N}{\lambda^* x.M \longrightarrow \lambda^* x.N}\ (\lambda x.M \text{ is a compound.})$$

2.4 Confluence

Theorem 2.1 (confluence_lamSF_red) *Reduction in λSF-calculus is confluent.*

Proof. The proof can be seen as an instantiation of Klop's result [13] for extensions of λ-calculus, in that the additional reduction rules are left-linear and orthogonal.

The only catch is that the reduction rule for F has a side-condition, so some care is required. □

2.5 Normal Forms

The *normal forms* are defined to be the variables, operators, abstractions of normal forms, and applications MN in which M and N are both normal and MN is either a compound or has an active variable.

Theorem 2.2 (irreducible_iff_normal) *A term is irreducible if and only if it is a normal form.*

A *program* is a closed normal form. A *factorable form* is either an operator or a compound.

Theorem 2.3 (programs_are_factorable) *All programs are factorable forms.*

Hence, any closed term of the form $FPMN$ must reduce. This is a form of progress result.

3 Definable Equality

It follows from Theorem 2.3 that the equality term defined in SF-calculus [7] serves to define equality in λSF-calculus too. The algorithm is as follows. Operators are equal if they have the same extensional behaviour, which can be decided by some term eqop. Atoms and compounds are never equal. Compounds are equal if their components are. The actual term is given

$$\mathsf{fix}\,(\lambda e.\lambda x.\lambda y.F\,x\,(\mathsf{eqop}\,x\,y)\,(\lambda x_l.\lambda x_r.Fy(KI)(\lambda y_l.\lambda y_r.e\,x_l\,y_l(e\,x_r\,y_r)(KI)))) \;.$$

where fix is a fixpoint term. This, and other approaches to recursion, will be addressed in Section 6.

Theorem 3.1 (equal_programs) equal $M\ M \longrightarrow^* K$ *for all programs* M.

Theorem 3.2 (unequal_programs) equal $M\ N \longrightarrow^* KI$ *for all distinct programs* M *and* N.

Proof. The proof is by induction on the rank of M, as defined in the Coq implementation. The only case of interest arises when M is an abstraction and N is an application. Now the left component of N cannot be abs_left since any application of abs_left reduces, and so the left components of M and N cannot be equal. □

4 Extensionality

Mathematically, two functions f and g are extensionally equivalent if they have the same graph. For unary functions, this means that $f\ x = g\ x$ for all x. In λ-calculi, extensionality is captured by adding the η-reduction rule

$$\lambda x.f\ x \longrightarrow f \quad \text{if } x \text{ is not free in } f.$$

When added to the basic λ-calculus, with just the β-rule, we get the $\lambda\beta\eta$-calculus, which is confluent. Define $=_{\beta\eta}$ to be the equivalence relation on λ-calculus induced by β-reduction and η-reduction. However, adding the η-rule to λSF-calculus is unsound, as can be seen from the following calculations. Define $\equiv_{\beta\eta SF}$ to be the equivalence relation on λSF induced by its reduction rules and the η-rule. First, the operators S, K and I become equal to their usual interpretations, by

$$\begin{aligned} S &\equiv_{\beta\eta SF} \lambda x.\lambda y.\lambda z.Sxyz \equiv_{\beta\eta} \lambda x.\lambda y.\lambda z.xz(yz) \\ K &\equiv_{\beta\eta SF} \lambda x.\lambda y.Kxy \equiv_{\beta\eta SF} \lambda x.\lambda y.x \\ I &\equiv_{\beta\eta SF} \lambda x.Ix \equiv_{\beta\eta SF} \lambda x.x \ . \end{aligned}$$

Then we have $SKM \equiv_{\beta\eta SF} \lambda x.x \equiv_{\beta\eta SF} (SKN)$ for any terms M and N. Further,

$$F(SKM)I(KI) \equiv_{\beta\eta SF} KI(SK)M =_{\beta\eta SF} M$$

shows that $M \equiv_{\beta\eta SF} N$ and this for any M and N. The calculus has collapsed.

A more useful relation is obtained by excluding the rule for factoring compounds from the equivalence relation, to get the equivalence relation $\equiv_{\beta\eta SK}$. Define terms M and N of λSF to be *extensionally equivalent* if $M \equiv_{\beta\eta SK} N$. For example, we have the following lemma.

Lemma 4.1 (star_equiv_abs) *$\lambda^* x.M \equiv_{\beta\eta SK} \lambda x.M$ for all terms M.*

Here are three more examples of definable program manipulations that preserve extensional behaviour.

Define a combinator wait so that

$$\mathsf{wait}\ M\ N \longrightarrow^* S(S(KM)(KN))I$$

using standard combinatorial techniques. The right-hand side is normal if M and N are, but application to some P reduces this to $M\ N\ P$ so that $\mathsf{wait}\ M\ N$ waits for P before applying M to N. It follows that

Lemma 4.2 (wait_ext) *For all terms M and N, $\mathsf{wait}\ M\ N \equiv_{\beta\eta SK} M\ N$.*

Define a combinator tag with the property that

$$\mathsf{tag}\ T\ M \longrightarrow^* S(KM)(SKT) \ .$$

Now SKT is an identity function for any T since

$$SKTP \longrightarrow KP(TP) \longrightarrow P \ .$$

It follows that when $\mathsf{tag}\ M\ N$ is applied to some P then it reduces by

$$S(KM)(SKT)P \longrightarrow KMP(SKTP) \longrightarrow^* MP \ .$$

Lemma 4.3 (tag_ext) *For all terms T and M, $\mathsf{tag}\ T\ M \equiv_{\beta\eta SK} M$.*

The resulting system of tags is as rich as the calculus as a whole, and so can be used to carry information about, say, constructors or types. In this paper, we will use just three tags in program analysis as follows: $\mathsf{abs} = \mathsf{tag}\ F$ will tag abstractions; $\mathsf{com} = \mathsf{tag}\ S$ will tag combinators; and $\mathsf{app} = \lambda x.\lambda y.\mathsf{tag}\ K\ (\mathsf{wait}\ x\ y))$ will tag applications.

Define a combinator eager such that eager $M\ N$ reduces to $M\ N$ if and only if N is factorable. That is, replacing $M\ N$ by eager $M\ N$ forces the application to evaluate N before evaluating the application of M to it. The target is a term similar to

$$FN(\lambda x.xN)(\lambda y.\lambda z.\lambda x.x(yz))M$$

where x, y and z are fresh. Now M is applied to N only if N has been reduced to factorable form. The new abstractions can be eliminated by ensuring

$$\mathsf{eager}\ M\ N \longrightarrow^* FN(SI(KN))(S(K(S(K(SI))))(S(KK)))$$

This will be used later to block non-terminating reductions. However, if your goal is to avoid re-computation of N then this approach has the weakness that if N reduces to an operator then it will be evaluated twice! This problem can be eliminated by introducing a variant of F in which the atomic branch uses its argument.

Lemma 4.4 (eager_is_eager) eager $M\ N \longrightarrow^* M\ N$ *for all programs* M *and* N*.*

5 Homomorphisms

The common features shared by all these calculi are that they are *applicative rewriting systems* [20] that have variables as a term form. Accordingly, it makes sense to define a *homomorphism* of applicative rewriting systems with variables to be a function from one such to another which has the following characteristics:

- it preserves the equivalence relation derived from reduction;
- it preserves applications;
- it preserves variables;
- it does not introduce free variables.

It is enough to require preservation up to equivalence, but for convenience, we will demand strict equality. Similarly, the requirement that a homomorphism does not introduce free variables can be weakened to require that closed terms be mapped to closed terms, or even that operators be mapped to closed terms. Note that the definition does *not* require that λ-abstractions be preserved, or that the image of S takes any particular form. All conditions are expressed in terms of concepts common to all the calculi under consideration, namely rewriting, variables and applications.

Theorem 5.1 (no_homomorphism) *There is no homomorphism from* λSF*-calculus to* λ*-calculus.*

Proof. Assume that there is such a homomorphism. Then it can be composed with the embedding of λ-calculus into $\lambda\beta\eta$-calculus to get a homomorphism $[\![-]\!]$. It follows that

$$[\![S]\!] \equiv_{\beta\eta} \lambda x.\lambda y.\lambda z.[\![S]\!]xyz \equiv_{\beta\eta} \lambda x.\lambda y.\lambda z.[\![Sxyz]\!] \equiv_{\beta\eta} \lambda x.\lambda y.\lambda z.xz(yz)\ .$$

Similarly, we can show that $[\![K]\!] \equiv_{\beta\eta} \lambda x.\lambda y.x$ and $[\![I]\!] \equiv_{\beta\eta}$-equivalent to $\lambda x.x$. Finally, in SF-calculus we have $F(SKM)F(KI) \longrightarrow KI(SK)M \longrightarrow^* M$, for each term

M, and so

$$\begin{aligned}[F(SKM)I(KI)] &\equiv_{\beta\eta} [F]([S][K][M])[I]([K][I]) \\ &\equiv_{\beta\eta} [F](\lambda x.x)(\lambda x.x)(\lambda x.\lambda y.y) \ .\end{aligned}$$

Hence, by the homomorphism property, we have

$$[M] \equiv_{\beta\eta} [F](\lambda x.x)(\lambda x.x)(\lambda x.\lambda y.y) \ .$$

Now the right-hand side is independent of M and so we have, for any N, that $[M] \equiv_{\beta\eta} [N]$. In particular we have $x = [x] \equiv_{\beta\eta} [y] = y$ for any variables x and y, which yields a contradiction. □

Corollary 5.2 *There is no homomorphism from SF-calculus to λ-calculus.*

Proof. The proof of Theorem 5.1 applies equally to SF-calculus. □

6 Programs as Normal Forms

The identification of programs with (closed) normal forms in an untyped setting is rather unusual. Of course, we cannot isolate the terminating computations, as this would solve the Halting Problem. If, further, we allow any computation to be a program, i.e. albeit one that takes no inputs, then the game is over. However, by separating the program from its inputs, we can use combinatory techniques to block any troublesome reductions in the program until the input is given. In this manner, programs can be made strongly normalising, and so can be identified with (closed) normal forms.

A crude solution would be to replace all abstractions with the corresponding star abstractions, as these are closed normal forms by construction. However, this is surely more violent than necessary.

Ideally, the identification should be demonstrated using a small programming language, with a conversion function from programs to closed normal forms of λSF-calculus, but this is beyond our current scope. There may be several ways to do this, and the options will change dramatically if the language is typed. Rather than explore these options, which would take some time, let us rather show how to overcome the key difficulty, namely the representation of recursive programs.

Consider a recursive program of the form

$$\texttt{let rec f x} = \texttt{M}$$

where `M` may contain `f` and `x` as free variables. Its standard representation is by a term of the form $\mathsf{fix}\ (\lambda f.\lambda x.M)$ where fix is a fixpoint function defined to be $\omega\omega$ where $\omega = \lambda x.\lambda f.f(xxf)$. It follows that

$$\mathsf{fix} = \lambda x.\lambda f.f(xxf)\omega \longrightarrow \lambda f.f(\omega\omega f) = \lambda f.f(\mathsf{fix}\ f) \ .$$

so that $\mathsf{fix}\ f \longrightarrow f\ (\mathsf{fix}\ f)$. This expresses the recursion very cleanly, but now program representations do not have a normal form.

However, we can delay the application of ω to ω by replacing fix by the extensionally equivalent term

$$\mathsf{fix2} = \lambda f.\mathsf{wait}\ (\mathsf{wait}\ \omega\ \omega)\ f \ .$$

Its application to a normal form f also has a normal form, but further application to some x reduces to $\omega\ \omega\ f\ x$ which is $\mathsf{fix}\ f\ x$. Now the original program can be interpreted by $\mathsf{fix2}\ (\lambda f.\lambda x.M)$. In the same manner, we may define $\mathsf{fix3}$ and $\mathsf{fix4}$ etc, so that recursive programs can be made to wait for any number of arguments before risking non-termination.

This accounts for the outermost recursion in a program, but when recursive functions are composed then this technique produces terms of the form

$$\lambda x.\mathsf{fix2}\ f\ (\mathsf{fix2}\ g\ x)$$

which re-introduces arbitrary computations into programs through $\mathsf{fix2}\ g\ x$. To block this, introduce eager evaluation, as described in Section 4 and define yet another fixpoint term by

$$\mathsf{fix\_eager} = \lambda f.\lambda x.\mathsf{eager}\ (\mathsf{wait}\ (\mathsf{wait}\ \omega\ \omega)\ f)\ x\ .$$

Now the composition of recursive programs normalises since evaluation of the recursion is blocked until the bound variable x takes a value. In this manner, the core constructions used to create recursive programs can be controlled by combinators to ensure that they do not introduce non-termination.

7 Extensional Conversion to Combinators

The extensional conversion of program to combinators is given by the recursive, pattern-matching function

$$\begin{array}{l}\mathsf{to\_combinator} := \\ \mid O \Rightarrow O \\ \mid \lambda x.M \Rightarrow \mathsf{to\_combinator}\ (\lambda^* x.M) \\ \mid MN \Rightarrow (\mathsf{to\_combinator}\ M)\ (\mathsf{to\_combinator}\ N))\end{array}$$

which eventually converts each abstraction $\lambda x.M$ in its argument to $\lambda^* x.M$.

Theorem 7.1 (to_combinator_makes_combinators) *If M is a closed term then* $\mathsf{to\_combinator}\ M$ *is a combinator.*

Since it is easy to test for abstractions and compounds, there is no difficulty in representing $\mathsf{to\_combinator}$ as a program, namely,

$$\mathsf{to\_comb} = \mathsf{fix}(\lambda f.\lambda x.F\ x\ x\ (\lambda x_l.\lambda x_r.\mathsf{equal\ abs\_left}\ x_l\ (f\ x_r)\ ((f\ x_l)\ (f\ x_r))).$$

Theorem 7.2 (to_combinator_is_extensional) $\mathsf{to\_combinator}\ M \equiv_{\beta\eta SK} M$ *for all terms M.*

Theorem 7.3 (to_combinator_to_comb) *For all programs M we have*

$$\mathsf{to\_comb}\ M \longrightarrow^* \mathsf{to\_combinator}\ M\ .$$

Summarising, if M is a program then $\mathsf{to\_comb}\ M$ reduces to the combinator $\mathsf{to\_combinator}\ M$ which is extensionally equivalent to M.

8 Program Analysis and Optimisation

The extensional conversion above can be optimised in various ways. In particular, there is no need to convert programs that are already combinators. Also, it is more efficient to convert $\lambda x.M$ to KM if x is not free in M. Define $\mathsf{is\_comb}$ by

$$\mathsf{is\_comb} = \mathsf{fix}(\lambda f.\lambda x.F\, x\, K(\lambda x_l.\lambda x_r.\mathsf{equal}\ \mathsf{abs\_left}\, x_l\, (KI)\, ((f\, x_l)\, (f\, x_r)\, (KI))\ .$$

Theorem 8.1 (is_comb_true) *For all programs M, if M is a combinator then* $\mathsf{is\_comb}$ M *reduces to K.*

Theorem 8.2 (is_comb_false) *For all programs M, if M is not a combinator then* $\mathsf{is\_comb}$ M *reduces to KI.*

The test for deciding if a program $\lambda x.M$ uses its argument x can be defined by a term binds that detects copies of I in $\lambda^* x.M$. It is given by

$$\mathsf{binds} = \mathsf{fix}\ (\lambda f.\lambda x.\mathsf{equal}\ I\ x\ K\ (F\ x\ (KI)\ (\lambda x_l.\lambda x_r.(f\ x_l)\ K\ (f\ x_r))))\ .$$

Theorem 8.3 (binds_abs_false) *For all programs $\lambda x.M$, if M is closed then* binds $(\lambda x.M)$ *reduces to KI.*

Theorem 8.4 (binds_abs_true) *For all programs $\lambda x.M$, if x is free in M then* binds $(\lambda x.M)$ *reduces to K.*

These ideas lead to the definition of the *optimised extensional conversion function* given by

$$\begin{array}{l}
\mathsf{to_combinator_opt} := \\
\quad \mid O \Rightarrow O \\
\quad \mid \lambda x.M \Rightarrow (\mathsf{to_combinator_opt}\ (\mathsf{if}\ \mathsf{binds}\ (\lambda x.M)\ \mathsf{then}\ (\lambda x.M)\ \mathsf{else}\ (KM)) \\
\quad \mid MN \Rightarrow \mathsf{if}\ \mathsf{is_combinator}\ (MN) \\
\qquad\qquad \mathsf{then}\ (MN) \\
\qquad\qquad \mathsf{else}\ (\mathsf{to_combinator_opt}\ M)\ (\mathsf{to_combinator_opt}\ N))
\end{array}$$

It is easy to reprise the treatment of $\mathsf{to\_combinator}$ for $\mathsf{to\_combinator\_opt}$, but since these ideas will recur in the next section, there is no particular reason to go through the details here.

9 Intensional Conversion to Combinators

Although the conversion functions above preserve extensionality, they lose intensional information, in that an abstraction $\lambda x.M$ becomes indistinguishable from a star abstraction $\lambda^* x.M$ or combinator. A conversion function f *preserves intensions* if it does not lose information, i.e. there is another transformation g such that $g(f\ M)$ reduces to M for all programs M.

For example, star abstraction is intensional, since there is an inverse, given by

$$\begin{array}{l}
\mathsf{unstar} = \\
\quad \mid O \Rightarrow O \\
\quad \mid \lambda x.M \Rightarrow \lambda x.(\mathsf{unstar} M) \\
\quad \mid KM \Rightarrow \mathsf{abs_K}\ M \\
\quad \mid SMN \Rightarrow \mathsf{abs_S}\ M\ N
\end{array}$$

where $\mathsf{abs\_K} = \lambda x.\lambda y.x$ and $\mathsf{abs\_S} = \lambda x.\lambda y.\lambda z.x\ z\ (y\ z)$. The corresponding program, also called unstar, is given by program

$$\begin{aligned}\mathsf{unstar} = \mathsf{fix}\ (\lambda f.\lambda x.f\ x\ x\ (\lambda x_l.\lambda x_r.\mathsf{equal\ abs\_left}\ x_l\ (\lambda z.f\ (x\ z))\\ (\mathsf{equal}\ K\ x_l\ (\mathsf{abs\_K}\ x_r)\ (F\ x_l\ x\ (\lambda x_{ll}.\lambda x_{lr}.\mathsf{abs\_S}\ x_{lr}\ x_r)\ .\end{aligned}$$

Theorem 9.1 (unstar_star) *Star abstraction is intensional, with inverse* unstar.

The extensional conversion from programs to combinators can be made intensional, too, by adding tags to record the presence of abstractions and combinators. The *optimised, intensional* conversion of programs to combinators is given by

```
to_combinator_int :=
| O ⇒ O
| λx.M ⇒ abs (to_combinator_int (if binds (λx.M) then (λ*x.M) else (KM)))
| MN ⇒ if is_combinator (MN)
          then com (MN)
          else app (to_combinator_opt M) (to_combinator_opt N)
```

Theorem 9.2 (to_combinator_int_makes_combinators) *If M is a closed term then* to_combinator_int M *is a combinator.*

Theorem 9.3 (to_combinator_int_is_extensional) *For all closed terms M we have* to_combinator_int $M \equiv_{\beta\eta SK} M$.

The corresponding program to_comb_int is given by

$$\begin{aligned}\mathsf{to\_comb\_int} = \mathsf{fix}\ (\lambda f.\lambda x.F\ x\ x\ (\lambda x_l.\lambda x_r.\mathsf{equal\ abs\_left}\ x_l\\ (\mathsf{abs}\ (f(\mathsf{binds}\ x_r\ x_r\ (K(x_rK))))\\ (\mathsf{is\_comb}\ x\ (\mathsf{com}\ x)\ (\mathsf{app}\ (f\ x_l)\ (f\ x_r))).\end{aligned}$$

Theorem 9.4 (to_comb_int_to_combinator_int) *For all programs M, there is a reduction* to_comb_int $M \longrightarrow^*$ to_combinator_int M.

For the conversion in the opposite direction, define to_program by

```
to_program :=
| O ⇒ O
| abs M ⇒ unstar(to_program M)
| com M ⇒ M
| app MN ⇒ (to_program M)(to_program N)
```

This can be defined by a term to_prog.

Theorem 9.5 (to_comb_int_is_intensional) to_comb_int *is intensional, with inverse given by* to_prog.

Summarising, to_comb_int maps programs to combinators in a manner that is both extensional and intensional.

Verification in Coq

The proofs of all the named lemmas and theorems in the paper have been verified using the Coq proof assistant. Details can be found in the source files [6]. This section will reprise some of the key definitions and theorems, to gain some feeling about how well aligned are the manual and automated approaches.

The operators and terms of `lamSF` are given by

```
Inductive operator := | Sop | Fop .
Inductive lamSF : Set :=
  | Ref : nat -> lamSF
  | Op  : operator -> lamSF
  | Abs : lamSF -> lamSF
  | App : lamSF -> lamSF -> lamSF .
```

The declaration of `operator` declares a type `operator` with two constructors `Sop` and `Fop`. Then the declaration of the type `lamSF` introduces four constructors. `Ref` is used to construct variables, represented by de Bruijn indices of type `nat`, the type of natural numbers. `Op` is used to build the operators S and F as `Op Sop` and `Op Fop`. In most situations, all operators are treated uniformly, which is exploited by giving them a separate type. `Abs` constructs abstractions and `App` constructs applications. In this manner, the function $\lambda x.\lambda y.xySF$ is represented by

$$\texttt{Abs(Abs(App(App(App(Ref 1)(Ref 0))(Op Sop))(Op Fop)))}\ .$$

The biggest gap between this representation and the paper representation is the use of de Bruijn indices for variables. For example, the requirement `maxvar M = 1` means that `M` has exactly one free variable (indexed by `0`).

The Coq versions of the named results in the paper are given in Figure 3. Most of the unexplained notation, such as `confluence` should be self-explanatory. Note, however, that `homomorphism` is here defined to be a homomorphism from `lamSF` to `lambda` rather than a homomorphism in general. Also, `beta_eta_eq` is here the equivalence relation generated from $\beta\eta SK$-reduction, and not just from β- and η-reduction.

10 Fresh Approaches

Having established the basic machinery of λSF-calculus and seen something of its expressive power, it is interesting to consider, at least in outline, how it suggests fresh approaches to some issues.

Gödelisation Although λ-calculus is Turing-complete, in the sense of being able to compute any number that a Turing machine can, there are strong limits to its ability to compute functions of λ-terms. For example, equality of closed normal λ-abstractions is not definable as λ-abstraction [1]. Nor is it possible to so define the Gödel number of a closed normal form. With a little effort, the conversion of programs to combinators can be extended to support Gödelisation.

Self-interpretation Self-interpretation is used to support programming language implementation within the language itself. In particular, it can be used to impose an evaluation strategy upon a confluent calculus such as λ-calculus [14] or SF-calculus [9]. Traditionally, the first step in self-interpretation is to use meta-level

```
Theorem confluence_lamSF_red: confluence lamSF lamSF_red.
Theorem irreducible_iff_normal:
  forall M, irreducible M lamSF_red1 <-> normal M.
Theorem programs_are_factorable : forall M, program M  -> factorable M.
Theorem equal_programs : forall M, program M -> lamSF_red (App (App equal M) M) k_op.
Theorem unequal_programs :
  forall M N, program M -> program N -> M<>N ->
    lamSF_red (App (App equal M) N) (App k_op i_op).
Lemma star_equiv_abs : forall M, beta_eta_eq (star M) (Abs M) .
Theorem no_homomorphism: forall h, homomorphism h -> False.
Theorem to_combinator_makes_combinators :
  forall M, closed M -> combinator (to_combinator M).
Theorem to_combinator_is_extensional : forall M, beta_eta_eq M (to_combinator M).
Theorem to_combinator_to_comb:
  forall M, program M -> lamSF_red (App to_comb M) (to_combinator M).
Theorem is_comb_true: forall M, program M -> combinator M -> lamSF_red (App is_comb M) k_op.
Theorem is_comb_false:
  forall M, program M -> (combinator M -> False) ->
    lamSF_red (App is_comb M) (App k_op i_op).
Theorem binds_abs_false :
  forall M, program (Abs M) -> closed M ->
    lamSF_red (App binds (Abs M)) (App k_op i_op).
Theorem binds_abs_true :
  forall M, program (Abs M) -> maxvar M = 1 ->
    lamSF_red (App binds (Abs M)) k_op.
Theorem unstar_star : forall M, normal M -> lamSF_red (App unstar (star M)) (Abs M).
Lemma wait_ext : forall M N, beta_eta_eq (wait M N) (App M N).
Lemma tag_ext : forall T M, beta_eta_eq (tag T M) M.
Lemma eager_is_eager : forall M N, factorable N -> lamSF_red (eager M N) (App M N).
Theorem to_combinator_int_makes_combinators :
  forall M, closed M -> combinator (to_combinator_int M).
Theorem to_combinator_int_is_extensional :
  forall M, closed M -> beta_eta_eq M (to_combinator_int M).
Theorem to_comb_int_to_combinator_int:
  forall M, program M ->
    lamSF_red (App to_comb_int M) (to_combinator_int M).
Theorem to_comb_int_is_intensional :
  forall M, program M -> lamSF_red (App to_prog (App to_comb_int M)) M.
```

Fig. 3. Theorems Verified in Coq

calculations to *quote* a program, to produce a data structure that is suitable for analysis. Since programs in λSF-calculus are already data structures, there is no need for quotation. Indeed, evaluation strategies can be defined within the calculus, without the need for any meta-level analysis.

Term constructors In the traditional λ-calculus account, the same λ-abstraction may have several different meanings. For example, the natural number zero may be represented as $\lambda f.\lambda x.x$, in which f is applied zero times to x. Also, the boolean for falsehood may be represented by $\lambda x.\lambda y.y$ in which the second branch, represented by its second argument, is taken. However, $\lambda f.\lambda x.x$ and $\lambda x.\lambda y.y$ are equivalent under renaming of bound variables, so that the same term has two different meanings. Traditionally, these have been distinguished by either introducing constructors, such as Zero and False, or adding types, such as Nat and Bool, or both. Now, we can tag these abstractions with information about their status as constructors, or their types. Similarly, constructor *arities* can be recorded by using wait.

Pattern calculus In λSF-calculus, it should be possible to give a complete account of constructor equality and pattern-matching by manipulating intensional information.

Type checking Similarly, once terms are tagged with type information, the calculus should support type checking and type inference.

Evaluation strategy Confluent rewriting systems support a natural model of program optimisation by changing the order in which sub-expression are evaluated.

However, sequential execution requires that an evaluation strategy be imposed. As with intensional information, different strategies give rise to a variety of different calculi [16]. These can be captured by using terms such as wait and eager to control evaluation order.

Partial evaluation Once programs are represented by normal forms, it is much easier to understand the nature of partial evaluation, of static arguments versus dynamic arguments, etc [11]. As before, these analyses should now be representable as programs.

Domain specific languages Users are driven to create their own, domain-specific programming languages because general purpose languages prove to be sub-optimal for their needs. One approach is to grow a language from a small core [19,10]. This will be easier once program analysis and evaluation strategies are definable.

11 Conclusions

λSF-calculus combines the best features of λ-calculus and combinatory calculi within a single calculus in that λ-abstraction provides a natural account of functionality through its β-reduction, while combinators provide a natural account of data structures, once the factorisation operator F is supported. Together, they show how programs and data structures can both be identified with the closed normal forms of λSF-calculus, so that they may be applied or analysed at any time. Further, the combinators can be used to tag programs with additional, intensional information, e.g. about constructors or types, or to control evaluation strategy by making applications wait before reducing.

The identification of programs and data structures also removes a layer of indirection from program analysis. There is no need to quote or Gödelise abstractions. Nor is there need for a separate state machine, to evaluate programs expressed on a tape. The ramifications may extend to all aspects of programming language design and implementation, including analysis and optimisation.

Like pattern calculus and SF-calculus, λSF-calculus supports powerful collection of generic queries for searching and updating data structures. However, the earlier calculi were far removed from current experience, making adoption difficult. By contrast, λSF-calculus merely adds a couple of operators to the popular λ-calculus approach, which makes migration much easier.

In conclusion, λSF-calculus adds intensionality to the extensional nature of λ-calculus, so that one can query the internal structure of arbitrary closed normal forms, and treat programs as data structures.

References

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

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

[3] Michel Bel. A recursion theoretic self interpreter for the lambda-calculus. http://www.belxs.com/michel/#selfint.

[4] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In *CSL*, pages 85–99, 1992.

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

[6] Barry Jay. LamSF repository of proofs in Coq. https://github.com/Barry-Jay/lambdaSF, February 2016.

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

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

[9] Barry Jay and Jens Palsberg. Typed self-interpretation by pattern matching. In *Proceedings of the 2011 ACM Sigplan International Conference on Functional Programming*, pages 247–58, 2011.

[10] Barry Jay and Jose Vergara. Growing a language in pattern calculus. In *Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on*, pages 233–240. IEEE, 2013.

[11] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice Hall International, 1993.

[12] Stephen C. Kleene. λ-definability and recursiveness. *Duke Math. J.*, pages 340–353, 1936.

[13] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematical Center Amsterdam, 1980. Tracts 129.

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

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

[16] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.

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

[18] Fangmin Song, Yongsen Xu, and Yuechen Qian. The self-reduction in lambda calculus. *Theoretical Computer Science*, 235(1):171–181, March 2000.

[19] Guy L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3), 1999.

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