Beyond Lambda-Calculus Intensional Computation

Associate Professor Barry Jay

Centre for Artificial Intelligence
School of Software
University of Technology Sydney
Barry.Jay@uts.edu.au

Yow LambdaJam, 9th May 2017

abstract

There is a paradox in the foundations of computing. On the one hand, lambda calculus is supposed to compute anything that can be computed. On the other hand, many intensional computations, such as deciding equality of programs (terms in closed normal form), are not definable within lambda calculus. The paradox has now been resolved by close reading of the original papers and books, which clarifies that lambda calculus is inherently extensional.

More importantly, there are new, intensional calculi which are more expressive than lambda calculus. First, pattern calculus supports both higher-order functions and generic queries of data structures, by basing all computation on pattern matching. Second, SF-calculus can query programs as well as data structures. Third, lambda-SF calculus adds native support for lambda-abstraction to SF-calculus.

I love you, lambda . . .

I love you, lambda, because you support

- Turing computable functions,
- equational reasoning, and
- abstraction.

Many systems are Turing complete,
combinatory logic supports equational reasoning, but
you are the queen of abstraction.

λ-abstraction is good for

- programmers,
- modularity, and
- implementation.

. . . but I've met someone new (actually, a few others)

You don't support

evaluation strategies	like	*SKI*-calculus
easy proofs	"	$\delta\lambda$-calculus
generic queries or object-orientation	"	pattern calculus
program analysis	"	*SF*-calculus
everything	"	λ*SF*-calculus

[All are Turing-complete and support equational reasoning.
$\delta\lambda$-calculus, pattern calculus and λ*SF*-calculus support λ.
All can be typed (see further reading).]

menu

canapés		
evaluation strategies	in	*SKI*-calculus
easy proofs	”	$\delta\lambda$-calculus
starters		
generic queries	”	pattern calculus
object-orientation	”	pattern calculus
main course		
program analysis	”	*SF*-calculus
desserts		
abstraction	”	λ*SF*-calculus
programming	”	bondi

evaluation strategies

λ-calculus must add evaluation strategies
to avoid reduction under λ or choose between lazy and eager.
Also, the fixpoint function Y doesn't have a normal form since

$$Yf \longrightarrow f(Yf) \longrightarrow f(f(Yf)) \longrightarrow \ldots$$

So self-interpretation requires quotation to freeze the form.

In *SKI*-calculus (traditional combinatory logic) we can define a fixpoint combinator Y_2 such that $Y_2 f$ is not a redex but

$$Y_2 f\, x \longrightarrow f(Y_2 f)x\ .$$

Define programs to be normal forms without free variables.
So self-interpretation does not require quotation.

menu

easy proofs

In λ-calculus, substitution queries its target, producing critical pairs that obscure confluence and equational reasoning.

In delayed substitution λ-calculus ($\delta\lambda$-calculus), the β-rule is replaced by a family of rules, obtained by eliminating the $\ldots$ in

$$\begin{array}{rcl}
(\lambda x.x)u & \longrightarrow & u \\
(\lambda x.y)u & \longrightarrow & y \quad (y \neq x) \\
(\lambda x.z\, t_1 \ldots t_n\, t)u & \longrightarrow & (\lambda x.z\, t_1 \ldots t_n)u\, ((\lambda x.t)u) \\
(\lambda x.\lambda y.t)u & \longrightarrow & \lambda y.((\lambda x.t)u) \quad (y \text{ not free in } u)
\end{array}$$

No meta-function of substitution.
No critical pairs, so confluence is automatic.

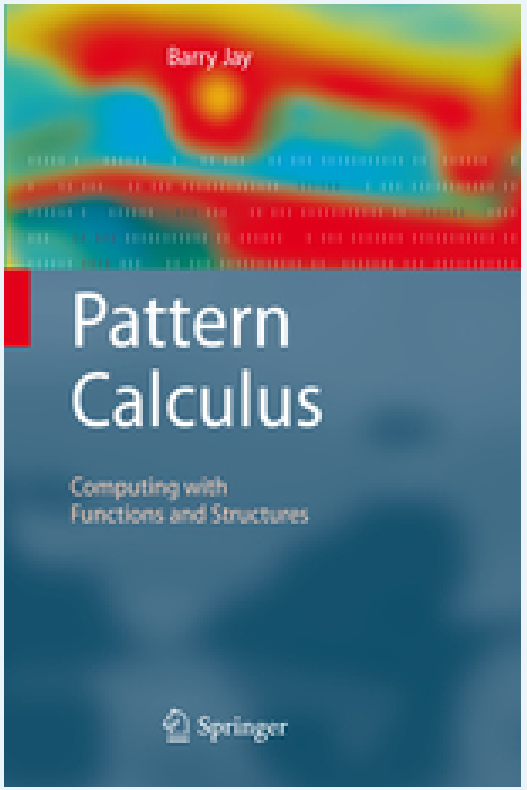
menu

generic queries

In λ-calculus, data structures are ad hoc, so queries, such as select, must be defined over and over.

In pattern calculus, the pattern x y matches any compound data structure, such as a pair or non-empty list, yielding generic queries.

```
select p z =
  if p z then [z] else
  match z with
   | x y → append (select p x) (select p y)
   | _ → []
(* if z satisfies p then return its singleton
else if z is a compound x y
then append the queries of its components
else return the empty list *)
```



object-orientation

In λ-calculus, method specialisation is ad hoc.

In pattern calculus, methods are pattern-matching functions which are specialised by adding another case. For example,

$$\text{move } dx = \ | \text{ Point } x\ r \rightarrow \text{Point } (x + dx)\ r$$

where Point is the constructor for a class of points with coordinate x, and r denotes additional data.

For coloured points, $r = \text{CPt } c\ s$.
Specialise move to change the color c to red, by

$$\begin{aligned} \text{move } dx = & \mid \text{Point } x\ (\text{CPt } c\ s) \rightarrow \text{Point } (x + dx)\ (\text{CPt } \mathit{red}\ s) \\ & \mid \text{Point } x\ r \rightarrow \text{Point } (x + dx)\ r \end{aligned}$$

In λ-calculus, program analysis requires encoding, quotation, reflection or staging. So,

λ-calculus is not complete for computation (JLAMP, 2017).

In *SF*-calculus (J Sym Log, 2011), the operator F can encode

$$\begin{array}{l}\text{match } x \text{ with}\\ \mid y\ z \to t\\ \mid \_ \to s\end{array}$$

as $F\,x\,s$ applied to the combinator for $\lambda y.\lambda z.t$. Recursive applications of F support arbitrary program analyses. So,

SF-calculus is intensionally complete.

λSF-calculus (MFPS, 2016) adds native support for λ-abstraction to SF-calculus, by showing how to factorise λ-abstractions. In brief, the components of $\lambda x.t$ are given by

- a tag that marks the source as a λ-abstraction, and
- $\lambda^* x.t$ that begins converting $\lambda x.t$ to an SF-combinator.

Recursive factorisation converts all programs to combinators, so the calculus is intensionally complete.

programming

bondi (2009) is our strongly typed programming language, based on pattern-calculus, with support for

- functional programming
- imperative programming
- queries
- object-oriented programming
- and even dynamic patterns.

However, it has no

- users or
- support for program analysis.

Shall we use typed λSF-calculus to grow a language for users?

menu

conclusions

Although there is much to love about it,

λ-calculus is not complete for computation.

Pattern calculus adds generic queries and object-orientation.

SF-calculus adds program analysis.

λ*SF*-calculus is Turing-complete, supports equational reasoning, supports abstraction and is intensionally complete.

Let's grow a multi-paradigm, self-analysing language based on

intensional computation.

appendix
λ-calculus

$$\begin{array}{rcll} s, t, u & ::= & x \mid tu \mid \lambda x.t & \text{(terms)} \\ (\lambda x.t)u & \longrightarrow & \{u/x\}t & (\beta\text{-reduction}) \end{array}$$

where $\{u/x\}t$ is the substitution of u for x in t defined by

$$\begin{array}{rcl} \{u/x\}x & = & u \\ \{u/x\}y & = & y \quad (y \neq x) \\ \{u/x\}(\lambda z.s) & = & \lambda z.\{u/x\}s \quad (z \text{ fresh}) \\ \{u/x\}(s\,t) & = & \{u/x\}s\{u/x\}t\,. \end{array}$$

A redex $s\,t$ is split then rebuilt, which makes proofs hard.

SKI-calculus

$$
\begin{array}{rcll}
O & ::= & S \mid K \mid I & \text{(operators)} \\
M, N, P, Q, R & ::= & O \mid MN & \text{(combinators)} \\
SMNP & \longrightarrow & MP(NP) & \text{(three reduction rules)} \\
KMN & \longrightarrow & M & \\
IM & \longrightarrow & M &
\end{array}
$$

λ-abstraction

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

new menu strategies

fixpoint combinators

In λ-calculus, $Y = \omega\omega$ where $\omega = \lambda x.\lambda f.f(xxf)$. Now $Y \longrightarrow \lambda^* f.f(Yf) \longrightarrow \ldots$.

In *SKI*-calculus, control evaluation order by defining

$$AMN := \lambda^* x.MNX = S(S(KM)(KN))I\ .$$

It delays application of M to N until a second argument is given. Now add A's everywhere to get

$$\begin{aligned} \omega_2 &= \lambda^* x.\lambda^* f.Af(A(Axx)f) \\ Y_2 &= A(A\omega_2\omega_2)\ . \end{aligned}$$

$Y_2 f$ is not a redex but $Y_2 fu \longrightarrow^* \omega_2\omega_2 fu \longrightarrow^* f(Y_2 f)u$.

strategies

$\delta\lambda$-calculus

Reduce $x\, t_1 \ldots t_n\, t$ to $x\langle t_1, \ldots, t_n, t\rangle$ by introducing tuples

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

The reduction rules become

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

new menu proofs

compounds

In pattern calculus, a compound is an application headed by a constructor, as indicated by capitalisation. Compounds include
Succ Zero
Pair Zero Nil
Cons h t and
Cons h
but not
Zero (which is an atom) or
$(\lambda x.x)$ Zero (which is a redex).

Given a function "even" for testing to be an even number, then

$$\text{select even (Pair 0 (Pair 1 Cons 2 (Cons 3 Nil)} \longrightarrow^* [0, 2]$$

ignores the pairs and lists while looking for numbers.
Like database queries, no need for the type scheme.

pattern calculus

$$\begin{array}{rcll} p, s, t, u & ::= & x \mid c \mid tu \mid [\theta]t \to t & \text{(terms)} \\ ([\theta]p \to s)u & \longrightarrow & \{u/[\theta]p\}s & \text{(match rule)} \end{array}$$

where $\theta = x_1, \ldots x_n$ is the bound variables and $\{u/[\theta]p\}$ is the match of p against u obtained by substituting for the variables in θ. Details are delicate.

Given a constructor Account then recover the account name by

$$\mid [\text{name}, \text{rest}]\ \text{Account name rest} \to \text{name}\ .$$

This uses a static pattern. Alternatively, use a dynamic pattern by generalising Account to a free variable account in

$$\mid [\text{name}, \text{rest}]\ \text{account name rest} \to \text{name}\ .$$

Then account can be later instantiated to Account or Customer, etc. by consulting an ontology.

λ-calculus is not complete

Equality of programs is computable.
Equality of programs is not definable in λ-calculus.
Therefore, there are
computable functions which are not definable in λ-calculus.

The workaround is to encode a λ-term by the Church numeral of its Gödel number, since equality of numbers is easily defined, but this encoding is not definable in λ-calculus either.

λ-definability (Kleene, 1936) is relative to an encoding, e.g. the identity function or the one above, but this was quite forgotten (J. of Logical and Algebraic Methods in Programming, 2017).

analysis conclusions

SF-calculus

$$
\begin{array}{rcll}
O & ::= & S \mid F & \text{(operators)} \\
M, N, P, Q, R & ::= & O \mid MN & \text{(combinators)} \\
SMNP & \longrightarrow & MP(NP) & \text{(three reduction rules)} \\
FOMN & \longrightarrow & M & O = S, F \\
F(PQ)MN & \longrightarrow & NPQ & P = S, SR, F, FR
\end{array}
$$

PQ above is a compound, and so is never a redex.
So no critical pairs, and confluence follows.

$$
\begin{aligned}
K &= FF \\
I &= SKK
\end{aligned}
$$

but F cannot be defined in terms of S, K and I
as it can separate the identity functions SKK and SKS.

intensional completeness

A Turing complete calculus is also

intensionally complete if it can define a Gödel function,

i.e. an invertible function from programs to natural numbers.

So program analyses can be reduced to numerical problems, which are handled by Turing-computability.

λSF-calculus

$$
\begin{array}{rcll}
O & ::= & S \mid F & \text{(operators)} \\
t, u & ::= & O \mid tu \mid \lambda x.t & \text{(terms)} \\
(\lambda x.t)u & \longrightarrow & \{u/x\}t & \text{(reduction rules)} \\
SMNP & \longrightarrow & MP(NP) & \\
FOMN & \longrightarrow & M & O = S, F \\
F(PQ)MN & \longrightarrow & NPQ & P = S, SR, F, FR \\
F(\lambda x.t)MN & \longrightarrow & NI(\lambda^* x.t) & (\lambda x.t \text{ is a compound})
\end{array}
$$

where $\lambda^* x.t$ is adapted from *SKI*-calculus, and abstractions are compounds if, in some sense, they are head normal (compare $\delta\lambda$-calculus). See (Math. Found. Progr. Lang. Sem., 2016).

No critical pairs, so confluence follows.

further reading


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


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


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


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


Barry Jay and Jens Palsberg.
Typed self-interpretation by pattern matching.
In *Proc. of the 2011 ACM Sigplan Int. Conference on Functional Programming*, pages 247–58, 2011 (ICFP).


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


Barry Jay.
Programs as data structures in λSF-calculus.
Electronic Notes in Theoretical Computer Science, 325:221 – 236, 2016 (MFPS XXXII).


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


Barry Jay.
Self-quotation in a typed, intensional lambda-calculus.
MFPS XXXIII, 2017, to appear.

further proving

My github repository of code, proofs, and some papers is
`https://github.com/Barry-Jay/`

Barry Jay.

Source code and examples for bondi, in Ocaml.
https://github.com/Barry-Jay/bondi.

Barry Jay.

SF repository of proofs in Coq.
https://github.com/Barry-Jay/SF.

Barry Jay.

LamSF repository of proofs in Coq.
https://github.com/Barry-Jay/lambdaSF.

Barry Jay.

Repository of proofs for L-calculus in Coq.
https://github.com/Barry-Jay/L-and-T-calculi.

Barry Jay.

Typed LambdaFactor Calculus repository of proofs in Coq.
https://github.com/Barry-Jay/Typed-lambdaFactor.

Barry Jay.

A combinatory account of substitution (pdf).
https://github.com/Barry-Jay/SF.