A combinatorial account of intensional lambda calculus

Barry Jay
Centre for Artificial Intelligence, University of Technology
Sydney, Australia
Barry.Jay@uts.edu.au

March 22, 2017

Abstract

Intensional lambda calculus is, in many ways, an improvement upon pure lambda calculus, in that its rewriting is closer to standard programming language practice while still being confluent, which allows aggressive optimisation. Further, this calculus can be translated to a combinatory calculus in a way that preserves reduction, and this without analysing abstractions, or increasing the term size, thus solving a problem that has been open for decades. The operators of the combinatory calculus for abstractions are purpose built, but exploit ideas about intensionality first developed in SF-calculus. Key theorems have been verified in Coq. Although this work is theoretical, it does suggest new avenues for programming language design and implementation.

Why λ-calculus?

- It is Turing complete.
- It is equational.
- It supports abstraction.

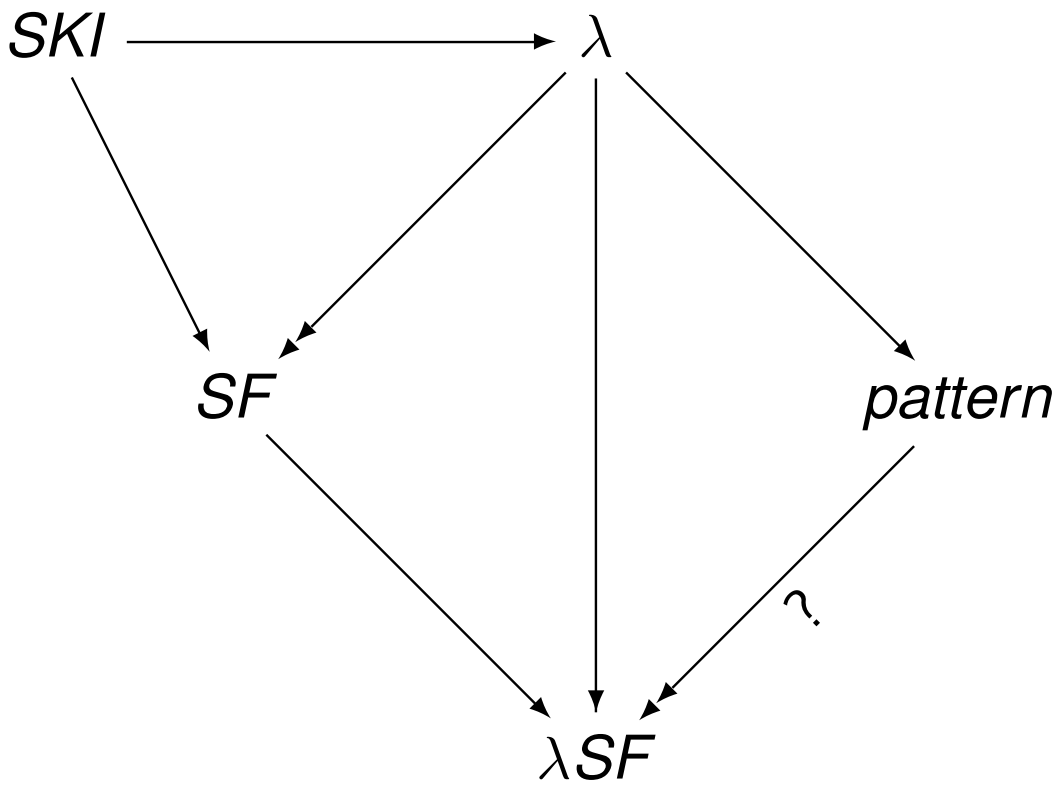
By contrast, the combinators of *SKI*-calculus are Turing complete, and equational, but abstraction is an afterthought. In particular, there is a reduction-preserving translation from *SKI* to λ

$$SKI \longrightarrow \lambda$$

but the standard translation in the other direction analyses the bodies of abstractions and so **breaks redexes**. This defect in the equivalence of the two calculi is an old puzzle, of preserving the ξ-rule, which we can now resolve.

Why **not** λ-calculus?

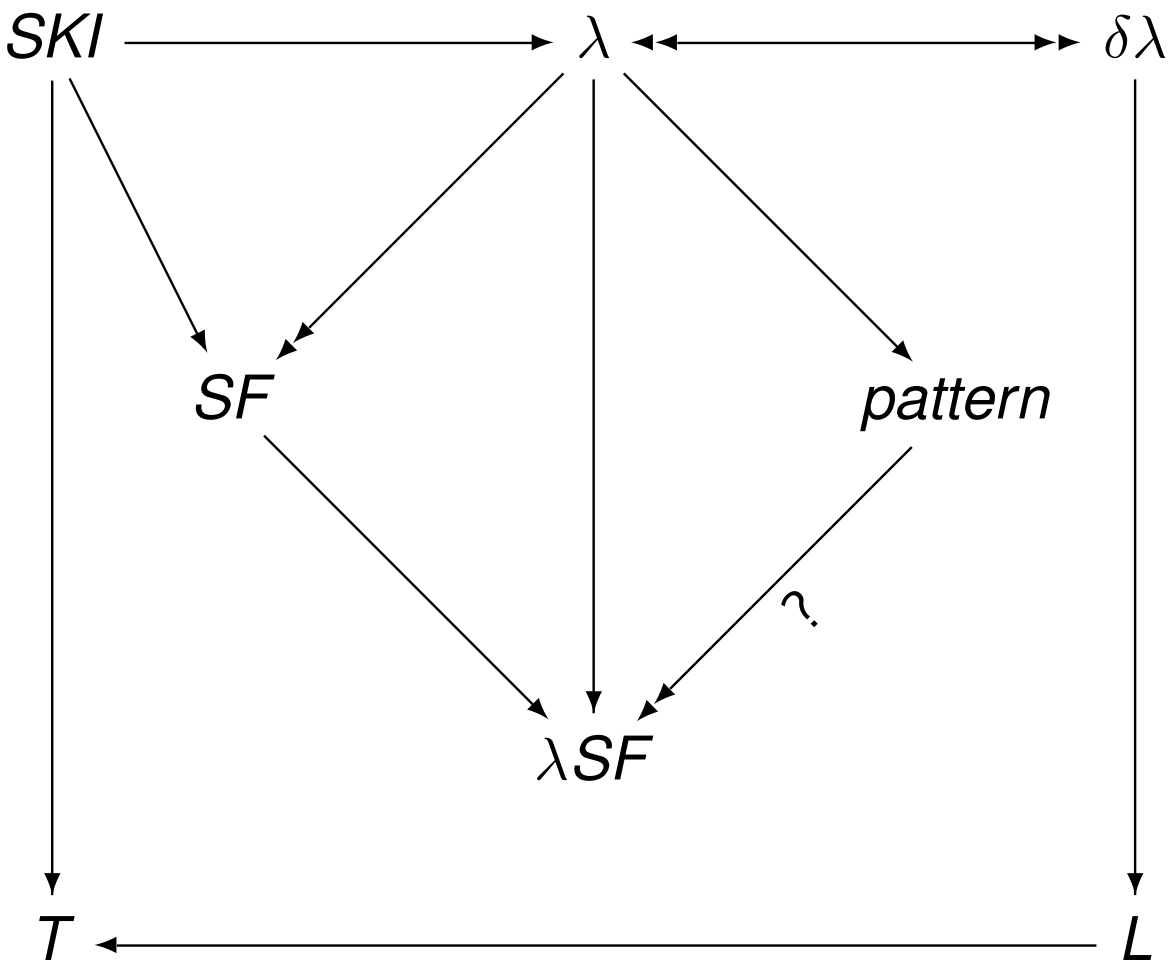
Three calculi (previously seen at FPSyd) are **more** expressive than λ-calculus, because **intensional**. From λ to *SF* is new.



$\longrightarrow$ preserves reduction

$\longrightarrow\!\!\!\!\!\rightarrow$ preserves reduction to normal form

The ξ-rule, intensionally



$\delta\lambda$-calculus

$\delta\lambda$-calculus weakens β-reduction to require that the abstraction body is in weak head normal form

$$(\lambda x.t)u \longrightarrow \ldots \quad (t \text{ is a whnf}) .$$

This makes the theory easier to work with, and closer to programming language implementations, without impacting upon normalisation (evaluation).

L-calculus

L-calculus is a combinatory calculus with operators

$$O ::= U \mid P \mid Z \mid N \mid Q \mid H \mid D \mid V \mid R \mid L\,.$$

For example, the λ-abstraction $\lambda x.\lambda y.\lambda z.x$ can be re-written in a style that is like de Bruin's, as $\lambda 0.\lambda 0.\lambda 0.2$ which translates to the combinator

$$LZ(LZ(LZ \quad (V(N(NZ))U \quad)))$$

where L is the abstraction operator, that thrice binds the variable indexed (in de Bruijn's style) by Z (for zero), while $V(N(NZ))U$ is the variable indexed by 2 and having no arguments, as indicated by the unit value U.

T-calculus

T-calculus has a single, ternary operator, the *tally* $|$ which represents everything. Looks like machine code, but is **confluent**. The tally queries all three of its arguments.

	Z	N	P
Z	$I = \|\|\|$	$Qi = \|\|(\|i)\|$ $Hij = \|\|(\|i)(\|j)$ $Rtjk = \|\|(\|t)(\|jk)$	$Ljt = \|\|(\|jt)$
N	$ViU = \|(\|i)\|$	$Sst = \|(\|s)(\|t)$	$Vi(Pst) = \|(\|i)(\|st)$
P	$Gst = \|(\|st)\|$	$Astu = \|(\|st)(\|u)$	$Dijst = \|(\|ij)(\|st)$

The table describes the meanings of terms of the form $|uv$. The rows are for u being $|$ or $|i$ or $|ij$. The columns are for v. The abstraction $\lambda x.\lambda y.\lambda z.x$ becomes $||(||(||(||(||(|(|(||)))|)))))$ by

$$\begin{array}{cccccccccccccc} ||(| & |(& ||(| & |(& ||(| & |(& |(|(& |(||))) & | &)))))) \\ L & Z(& L & Z(& L & Z(& V(& N(NZ)) & U &))) \end{array}$$

Verification in Coq

The properties of all the translations are being verified in Coq (almost done :) and will be uploaded to GitHub. Modulo these last steps, the paper has been written.

Status of *SKI* and λ

There is no translation that preserves normalisation
from *SF* or *T* to *SKI* or λ .

Neither *SKI* nor λ-calculus is complete for computation.

Use *SF*-calculus or λ*SF*-calculus or *T*-calculus instead.

Say what?

The traditional results on expressive power **cannot** be used to argue for the computational completeness of λ-calculus, as Jose Vergara and I explain in *Journal of Logical and Algebraic Methods in Programming* (2017).

- λ-definability of numerical functions is relative to an **encoding** of numbers as Church numerals.
- λ-definability in general is relative to an encoding of **symbols** as λ-terms, e.g. by Gödelising to get a number, and then Church encoding.
- λ-calculus supports **two** encodings, the Church-of-Gödel encoding and the identity function.
- The Church-of-Gödel encoding is not definable wrt the identity encoding, is not definable **within** λ-calculus.
- The Church-of-Gödel function is definable within SF-calculus, λSF-calculus and T-calculus.

Conclusions

The problem of representing abstraction within a combinatory calculus is solved by:

- weakening β-reduction to abstractions whose body is in weak head normal form; and
- representing abstractions etc. as **intensional** combinators, that query the structure of their arguments.

We end up with a menagerie of new calculi, all more powerful than λ-calculus or *SKI*-calculus, which expose the **weakness** of such extensional calculi, and the **ambiguities** in traditional theory.

The menagerie