Domain Theory for Intensional Computation

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

dedicated to Dana Scott

Intensional Computation

Extensional computation	applies	functions.
Intensional computation	queries	internal structure.
Pattern calculus	queries	data structures.
SF-calculus	queries	program structures.
λ*SF*-calculus	queries	lambda abstractions.

All three calculi are confluent, higher-order rewriting systems.

What is their denotational semantics?

Focus on *SF*-calculus versus *SK*-calculus (combinatory logic).

Domain Theory

In λ-calculus all closed normal forms are abstractions, so the domain equation is

$$D \cong D \to D \ .$$

In *SK*-calculus, S and K have arities 3 and 2 so normal forms are given by

$$n ::= S \mid S\ n \mid S\ n\ n \mid K \mid K\ n$$

and the domain equation is

$$C \cong 1 + C + C \times C + 1 + C \ .$$

A basis for C is given by adding $\perp$ to the normal forms

$$c ::= \perp \mid S \mid S\ c \mid S\ c\ c \mid K \mid K\ c \ .$$

Incompleteness of *SK*-calculus

There is a function from C to $C \to C$ that maps each combinator to the corresponding function of combinators. For example, *SKK* and *SKS* are both mapped to the identity on C.

There is **no** inverse from $C \to C$ to C.

For example, equality of normal forms is not *SK*-definable, since combinators, being extensional, cannot separate the identity functions *SKK* and *SKS*.

What about programs?

Recursive programs are fixpoint functions,
so who cares about normal forms?
Non-termination of fixpoints is unavoidable in λ-calculus, but *SK*-calculus supports

recursive programs in normal form

where programs are normal until given arguments. In brief, there is a combinator Y_2 such that $Y_2 f$ is a fixpoint **function**

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

but $Y_2 f$ is strongly normalizing (SN) if f is. For example, all μ-recursive functions are given by SN combinators.

SF-calculus

$$\begin{array}{rcll} SMNP & \longrightarrow & MP(NP) & \\ FOMN & \longrightarrow & M & O \text{ is } S \text{ or } F \\ F(PQ)MN & \longrightarrow & NPQ & PQ \text{ is a compound.} \end{array}$$

NOT all applications are compounds.
ONLY head normal applications are compounds, i.e. combinations of the form SM, SMN, FM or FMN.
Three rules with side conditions become seven rules without side conditions.

Combinations $\neq$ Combinators

Define

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

since $KMN = FFMN \longrightarrow M$ and $SKKM \longrightarrow KM(KM) \longrightarrow M$. So SF-calculus is combinatorially complete.

F is **not** definable in SK-calculus since it can separate the compounds SKS and SKK.

SF-calculus supports combinations that are not combinators.
SK-calculus is **not** combinationally complete.

Intensional Completeness

SF-calculus supports

- equality of normal forms
- pattern matching, including generic queries
- a Gödel function from normal forms to natural numbers
- arbitrary (computable) program analyses.

SF-calculus is intensionally complete.

Denotational Semantics of *SF*-calculus

Normal forms are given by

$$n ::= S \mid S\ n \mid S\ n\ n \mid F \mid F\ n \mid F\ n\ n$$

so the domain equation is

$$H \cong 1 + H + H \times H + 1 + H + H \times H\ .$$

Give H a basis by adding $\perp$ to the normal forms.

Theorem

$H \to H$ is a retract of H.

Proof.

$H \to H$ has a basis of step functions $d \downarrow e$ where d and e are in the basis for H. These are representable in $SF_\perp$-calculus as pattern-matching functions where $\perp = \_$ matches anything

$$\begin{array}{l} \mid d \Rightarrow e \\ \mid \_ \Rightarrow \perp\ . \end{array}$$

Conclusions

SK-calculus is incomplete for computation. The syntactic proof (2011) is now complemented by a semantic proof, that $C \to C$ is not a retract of C.

SF-calculus is intensionally complete. The syntactic proof for normal forms (2011), and the identification of programs with normal forms (2018), is now complemented by a semantic proof, that $H \to H$ is a retract of H. The proof identifies step-functions with pattern-matching functions.

Where are the lambdas?

In principle, this approach should apply to λSF-calculus, but deciding if an abstraction is a compound is very complex.

Recent work avoids this by giving a reduction-preserving translation of a λ-calculus (closure calculus) to SF-calculus.

What is the domain theory of closure calculus?
What happens when S and F are added?