Programs as Data Structures in λSF-Calculus

Barry Jay
University of Technology, Sydney

March 23, 2016

Abstract

Programs have a dual nature, as functions to be used in applications, and as data structures to be analysed and optimised during compilation. This talk introduces the lambda-SF-calculus, in which both natures are realised through the equations

program = closed normal form = data structure.

The second equation is already a theorem, since the internal structure of closed normal forms, even abstractions, is fully exposed by factorisation, mediated by the operator F. The first equation must account for recursive programs, represented by fixpoint functions. The fixpoint function has been redefined so that it preserves normality. It remains to find a way of separating the programs (which have normal forms) from their computations (which may fail to terminate).

The Wizard of Oz

Dorothy meets three new friends, each of which is missing something, and a wizard who can help.

The Friends Abilities

 the friends	 heart	 courage	 brains
 Tinman	✗	✓	✓
 Lion	✓	✗	✓
 Scarecrow	✓	✓	✗
 Wizard	✓	✓	✓

The Abilities (technical)

 the friends	 equations	 prog=data	 abstraction
 Tinman	✗	✓	✓
 Lion	✓	✗	✓
 Scarecrow	✓	✓	✗
 Wizard	✓	✓	✓

The Friends (technical)

 the friends	 equations	 prog=data	 abstraction
 Turing model	✗	✓	✓
 λ-calculus	✓	✗	✓
 SF-calculus	✓	✓	✗
 λSF-calculus	✓	✓	✓

For Turing machines, programs are strings. Their equality is trivial. Self-interpretation corresponds, more or less, to the existence of a universal Turing machine. Equational reasoning is out, and abstraction is difficult.

 the friends	 equations	 prog=data	 abstraction
 Turing model	✗	✓	✓

λ-calculus supports abstraction and equational reasoning, but abstractions are not data.
No definable equality, Gödelisation or quotation.

 the friends	 equations	 prog=data	 abstraction
 λ-calculus	✓	✗	✓

Pattern Calculus

Pattern calculus (Springer, 2009) supports a generic equality function for data structures:

$$
\begin{array}{l}
\text{let rec equal} = \\
\quad \mid x_1\ x_2 \longrightarrow (\ \mid y_1\ y_2 \longrightarrow (\text{equal } x_1\ y_1)\ \&\&\ (\text{equal } x_2\ y_2) \\
\qquad\qquad\qquad\quad \mid y \longrightarrow \text{false}) \\
\quad \mid x \qquad\ \longrightarrow (\ \mid y \longrightarrow \text{eqop } x\ y)
\end{array}
$$

so

$$\text{equal}\,(\text{Cons } h_1\ t_1)\,(\text{Cons } h_2\ t_2) \longrightarrow^* (\text{equal } h_1\ h_2)\&\&(\text{equal } t_1\ t_2)\ .$$

So a better account of data, but abstractions are not data, so no program analysis

$$\text{equal }(\lambda x.x)\ (\lambda x.x) \longrightarrow^* \text{False}$$

SF-calculus

The terms and reduction rules of SF-calculus (JSL, 2011) are

$$\begin{array}{rrcll} & M, N & ::= & x \mid S \mid F \mid M\ N & \\ (S) & SMNP & \longrightarrow & MP(NP) & \\ (K) & FOMN & \longrightarrow & M & O \text{ an operator, } S \text{ or } F \\ (F) & FPMN & \longrightarrow & NP^l P^r & P \text{ a compound of } P^l \text{ and } P^r \end{array}$$

The compounds are terms of the form SM, SMN, FM, FMN, i.e. head normal forms. $K = FF$ since $KMN = FFMN \longrightarrow M$. Any SKX is an identity function since

$$SKXM \longrightarrow KM(XM) \longrightarrow M\ .$$

However, F is **not** definable in terms of S, K and I since F can recover X from SKX, while SKI-calculus is *extensional*.

Pattern-Matching in *SF*-calculus

Now define equal by the pseudo-combinator

$$\begin{array}{l} \text{let rec equal } x\ y = \\ F\ x\ (\text{eqop } x\ y)\ (\lambda x_1.\lambda x_2. \\ \quad F\ y\ \text{False}\ (\lambda y_1.\lambda y_2.(\textit{equal}\ x_1\ y_1)\&\&(\text{equal } x_2\ y_2))) \end{array}$$

The fixpoints, abstractions etc. can be defined using traditional combinatory techniques.
Pattern-matching can also define a self-interpreter (ICFP '11).
But no abstractions.

 the friends	 equations	 prog=data	 abstraction
 *SF*-calculus	✓	✓	✗

λSF-Calculus is a Mashup

The terms and reduction rules of λSF-calculus are

$$\begin{array}{rcll}
M, N & ::= & x \mid S \mid F \mid \lambda x.M \mid M\ N & \\
(\lambda x.M)N & \longrightarrow & \{N/x\}M & \\
SMNP & \longrightarrow & MP(NP) & \\
FOMN & \longrightarrow & M & O \text{ an operator, } S \text{ or } F \\
FPMN & \longrightarrow & NP^l P^r & P \text{ a compound of } P^l \text{ and } P^r
\end{array}$$

The compounds now include abstractions.

 the friends	 equations	 prog=data	 abstraction
 λSF-calculus	✓	✓	✓

Factoring Abstractions

Factorisation is consistent with equational reasoning because it does not break any redexes. The compounds are defined to ensure this. As well as SMN etc., they include the head normal forms of λ-calculus, such as $\lambda x.\lambda y.x\ M\ N$ and some mixed terms such as $\lambda x.F\ x\ MN$.

If $\lambda x.M$ is a compound then its components are defined using two tricks, one new and one old.

$$\begin{array}{rcl} (\lambda x.M)^l & = & SKF \\ (\lambda x.M)^r & = & \lambda^* x.M\ . \end{array}$$

The left component records the presence of an abstraction, since $(SKF)(\lambda^* x.M)$ reduces. The right component $\lambda^* x.M$ replaces a λ by operators in traditional style, e.g. $\lambda^* x.x = I$.

Some Theorems

1. Reduction is confluent (so at most one normal form)
2. All closed normal forms are either operators or compounds (and so are factorable)
3. There is a definable conversion of closed normal forms to combinators that are extensionally and intensionally equivalent.

Extensional equivalence is defined in terms of $\beta\eta SK$-reduction.
Intensional equivalence means no information loss:
the conversion can be reversed.

All proofs have been verified in Coq.

Recursive Programs

The identification of programs with closed normal forms, or closed strongly normalising terms is:

- perfectly natural in the Turing model, where a program is a string, but
- counter-intuitive in the λ-calculus since
- strongly normalising λ-calculi require the addition of a fixpoint operator to become Turing complete. However,
- there is a fixpoint combinator such that fix f is strongly normalising if f is, with
- non-termination becoming possible only when fix f is applied to another argument, which
- suggests how to ensure programs are closed normal forms.

Fresh Approaches

The full power of combinators is revealed by factorisation. The λSF-calculus suggests fresh approaches to many topics in language design and implementation, including:

- Gödelisation
- Self-interpretation
- Term constructors
- Pattern calculus
- Type checking
- Evaluation strategy
- Partial evaluation
- Domain specific languages

Conclusions

λSF-calculus combines the best properties of the Turing model, λ-calculus and SF-calculus within a single calculus that supports

- equational reasoning
- λ-abstraction and
- program analysis

through the identifications

program = closed normal form = data structure