ENCODING THE FACTORISATION CALCULUS
REPRESENTING THE INTENSIONAL IN THE EXTENSIONAL

Reuben N. S. Rowe
EXPRESS/SOS, Madrid,
Monday 31st August 2015

Programming Principles, Logic & Verification Group
Department of Computer Science
University College London
Motivation

- We are interested in the relationship between the *Factorisation Calculus* and more familiar models of computation (viz. the $\lambda$-calculus)

- Factorisation Calculus is:
  - A combinatory rewrite system
  - A basis for a general theory of pattern matching
  - A model of *intensional* computations
    - cf. $\lambda$-calculus is an *extensional* theory of functions
The Factorisation Calculus

• Introduced by Jay and Given-Wilson (2011)
• A combinatory calculus comprising two operators: $S$ and $F$
• We identify two ‘special’ sets of terms:
  • Atomic terms: unapplied operators, i.e. \{S, F\}
  • Compound terms: partially applied operators, e.g. $S(FF)S$
• $S$ is the familiar combinator from Combinatory Logic:

\[
SXYZ \rightarrow XZ(YZ)
\]

• The $F$ operator distinguishes atomic terms from compounds, also factorising the latter:

\[
FXMN \rightarrow M \quad \text{if } X \text{ atomic}
\]
\[
F(PQ)MN \rightarrow NPQ \quad \text{if } PQ \text{ compound}
\]
The Factorisation Calculus: Important Properties

• It is *combinatorially complete*, since \( FF \) represents \( K \):

\[
FFXY \rightarrow X
\]

• The *internal structure* of terms can be analysed, so:
  • *Intensionally distinct* terms can be distinguished
  • The equality predicate on normal forms is representable

• Compare with Combinatory Logic (and so \( \lambda \)-calculus):
  • Equality of arbitrary normal forms *not* representable
  • Factorisation of combinators is not representable
    • e.g. there is no CL term \( T \) such that \( T(SKX) \rightarrow^* X \) for any \( X \)
Consider using arbitrary linear normal terms as patterns for matching, e.g.

\[
\{ SM (FN S)/S x (Fy S) \} = [x \mapsto M, y \mapsto N] \\
\{ SM N/F x y \} = \text{fail}
\]

A case \( G(P) = M \) (for pattern \( P \) and term \( M \)) defines a symbolic function \( G \) on combinators:

\[
G(U) = \begin{cases} 
\sigma(M) & \text{if } \{U/P\} = \sigma \\
\text{some default term} & \text{if } \{U/P\} = \text{fail}
\end{cases}
\]

A combinatory calculus is structure complete if every case \( G \) is represented by some term \( G \), i.e. \( GU \equiv_\beta G(U) \) for all \( U \)

**Theorem:** Factorisation Calculus is structure complete
How to Interpret the Characterisation?

- Jay and Given-Wilson use structural completeness as a way to characterise the expressive power of Factorisation Calculus.
- Factorisation Calculus is structurally complete; CL isn’t.
  - Conclusion: Factorisation Calculus is more expressive.
- There are symbolic functions representable in Factorisation Calculus but not in CL.
  - e.g. Factorisation, equality of normal forms.
- So, does the Factorisation Calculus compute more things?
  - The standard way to answer this is by showing the (non-)existence of an encoding.
Overview of our Encoding

- Factorisation Calculus
- λ-calculus

We use a construction due to Berrarducci and Böhm which encodes certain types of term rewriting system in λ-calculus. We show how to implement Factorisation Calculus as a suitable rewrite system. The encoding is faithful, i.e. preserves both reduction and termination.
• We use a construction due to Berrarducci and Böhm which encodes certain types of term rewriting system in $\lambda$-calculus.
• We use a construction due to Berrarducci and Böhm which encodes certain types of term rewriting system in $\lambda$-calculus
• We show how to implement Factorisation Calculus as a suitable rewrite system
We use a construction due to Berrarducci and Böhm which encodes certain types of term rewriting system in $\lambda$-calculus.

We show how to implement Factorisation Calculus as a suitable rewrite system.

The encoding is *faithful*.

- i.e. preserves both reduction and termination.
The Berrarducci-Böhm Representation Result

- A rewrite system $\mathcal{R}$ over a signature $\Sigma$ is canonical if:
  - $\Sigma = \Sigma_C \uplus \Sigma_F$ with every rewrite rule of the form:
    $$f(c(x_1, \ldots, x_n), y_1, \ldots, y_m) \rightarrow t \quad (c \in \Sigma_C \text{ and } f \in \Sigma_F)$$
  - That is, $\Sigma$ comprises constructors $\Sigma_C$ and programs $\Sigma_F$

- Berrarducci and Böhm (1992) showed that every such $\mathcal{R}$ has a representation $\phi_\mathcal{R}$ in $\lambda$-calculus, i.e.
  $$t \rightarrow_\mathcal{R} t' \Rightarrow \phi_\mathcal{R}(t) \rightarrow_\lambda \phi_\mathcal{R}(t')$$

- Moreover, for closed terms, $\phi_\mathcal{R}$ preserves strong normalisation
A Canonical Rewrite System for Factorisation

- Application is a *constructor-driven program*:

  \[
  \text{app} (S_0, x) \rightarrow S_1 (x) \quad \text{app} (F_0, x) \rightarrow F_1 (x) \\
  \text{app} (S_1 (x), y) \rightarrow S_2 (x, y) \quad \text{app} (F_1 (x), y) \rightarrow F_2 (x, y) \\
  \text{app} (S_2 (x, y), z) \rightarrow \text{app} (\text{app} (x, z), \text{app} (y, z)) \\
  \text{app} (F_2 (x, y), z) \rightarrow \text{factorise} (x, y, z)
  \]

- Factorisation is a program too:

  \[
  \text{factorise} (S_0, y, z) \rightarrow y \\
  \text{factorise} (S_1 (q), y, z) \rightarrow \text{app} (\text{app} (z, S_0), q) \\
  \text{factorise} (S_2 (p, q), y, z) \rightarrow \text{app} (\text{app} (z, \text{app} (S_0, p)), q) \\
  + \text{symmetric rules for } F_0, F_1, F_2
  \]
The translation into our rewrite system $SF_\oplus$ is straightforward:

$$[[S]]_\oplus = S_0 \quad [[F]]_\oplus = F_0 \quad [[MN]]_\oplus = \text{app}([[M]]_\oplus, [[N]]_\oplus)$$

We have shown that $[[\cdot]]_\oplus$ also preserves reduction and strong normalisation.

Thus $[[\cdot]]_\lambda = \phi_{SF_\oplus} \circ [[\cdot]]_\oplus$ is a faithful encoding of Factorisation Calculus in $\lambda$-calculus.
Some Observations

• Our encoding is **compositional**:

\[
[MN]_\lambda = \phi_{SF@}(\text{app} ([M]_\@, [N]_\@)) = \phi_{SF@}(\text{app}) [M]_\lambda [N]_\lambda
\]

• It is not a **homomorphism** ... however:
  • It looks like an instance of an **applicative morphism** (Longley)

• The ‘classical’ interpretation is that our encoding constitutes an equivalence
  • We need to look further to understand the notion of expressiveness captured by structural completeness
Felleisen’s Framework for Comparing Expressiveness

• Felleisen (1991) defined a formal expressiveness criterion based on the concept of *eliminability* in logic
  • A logic $L$ is more expressive than logic $L'$ if:
    1. $L$ is a conservative extension of $L'$
    2. $L$ contains a non-eliminable symbol

• By analogy, language $L$ is more expressive than language $L'$ if:
  1. it is a superset of $L'$
  2. it contains some construct which cannot be translated to $L'$ using a *macro*

• Consider SKF-calculus as a more expressive superset of CL, since $F$ is not representable using $S$ and $K$ (i.e. as a macro)
• Take computational models to be pairs: a (semantic) domain and a set of functions (the *extensionality*).
  • A larger extensionality = more expressive.

• Maps between domains induce *simulations* (i.e. encodings).
  • But some maps allow for simulations of strictly larger extensionalities!

• Different restrictions on the mappings between domains yield notions of (in)equivalence of varying strength.

• Our encoding shows a weak form of equivalence.

• Existing results would seem to imply inequivalence at a stronger level.
CONCLUSIONS & FUTURE WORK

• Factorisation Calculus is a recent fundamental model of computation with expressive intensional properties

• We have demonstrated the existence of a faithful encoding of the Factorisation Calculus in the $\lambda$-calculus

• Our results point towards a nuanced relationship between the two paradigms which requires further investigation

• We believe that research into the denotational semantics of Factorisation Calculus is a logical next step