Partial Evaluation and Normalisation by Traversals

Joint work by Daniil Berezun* and Neil D. Jones†

January 23, 2016

Game semantics re-examined. A starting point: the game semantics for PCF can be thought of as a PCF interpreter. In game semantics papers [1, 2, 3, 4, 5, 6, 12, 13, 14] the denotation of an expression is a game strategy. When played, the game results in a traversal[1]. Ong's recent paper [14] normalises a simply typed λ-expression using traversals.

A surprising consequence: it is possible to build a lambda calculus interpreter with **none** of the traditional implementation machinery: β-reduction; environments binding variables to values; and "closures" and "thunks" for function calls and parameters. (This was implicitly visible in early work on full abstraction for PCF.)

A new angle on game semantics: It looks very promising to study its operational consequences. Further, this may give a new line of attack on an old topic: *semantics-directed compiler generation* [7, 16].

An idea: specialise a traversal-based normaliser. Ong's algorithm [14] is defined by structural recursion on the syntax of (the eta-long form of) a λ-expression M. Consequence: the algorithm can be *specialised* with respect to the sub-λ-expressions of M. (Specialisation is also known as *partial evaluation*, see [9].)

An intermediate step: a low-level semantic language LLL**.** A partial evaluator, given a program p and the *static* portion s of its input data, will precompute the parts of p's computation that depend only on s, and generate residual code for all other parts of p. In the current context: specialisation is used to *factor* a given traversal algorithm $trav : \Lambda \to \textit{Traversals}$ into two stages:

$$trav = travgen\,;[\![\ ]\!]^{LLL} \text{ where } travgen : \Lambda \to \text{LLL} \text{ and } [\![\ ]\!]^{LLL} : \text{LLL} \to \textit{Traversals}$$

The specialised traversal-builder is a residual output program in language LLL. The output program contains no lambda-syntax; only target code to construct the traversal.

Traversals for $\Lambda^{simplytyped}$.

We programmed Ong's traversal algorithm in both HASKELL and SCHEME. The HASKELL version includes typing (Algorithm W, given user-defined types for free variables); conversion to eta-long form; the traversal algorithm itself; and construction of the residual λ-expression. The SCHEME version is (at the time of writing) nearly in form suitable for automatic specialisation. We will use the system UNMIX (Sergei Romanenko).

We have implemented an LLL-generator. Given an input λ-expression M, the generator produces as output an LLL program p_M that, when run, will yield the traversals of M. Symbolically: $[\![M]\!] = [\![[\![p_M]\!]]\!]$.

A well-known fact: the traversal of M may be much larger than M. (By Statman's results it may be larger by a "non-elementary" amount!). It is possible, though, to construct p_M so $|p_M| = O(|M|)$, i.e., M's LLL equivalent has size that is only linearly larger than M itself.

For specialisation, all calls of the traversal algorithm to itself that do not progress from one M subexpression to a proper subexpression are annotated as "dynamic". The motivation is increased efficiency: no such recursive calls in the traversal-builder will be unfolded while producing the generator; but all other calls will be unfolded.

The current implementation regards LLL as a subset of SCHEME, so the output p_M is currently produced in the form of a SCHEME program. (This will soon be changed, replacing SCHEME by a tiny subset of HASKELL.)

Traversals for $\Lambda^{untyped}$. A traversal algorithm for *untyped* λ-expressions M has been implemented in HASKELL. It is more complex than Ong's evaluator, using four different kinds of back pointers. The net effect is that an arbitrary untyped λ-expression can be translated into LLL. A correctness proof is pending.

As with Ong's evaluator, this algorithm is also defined by structural recursion on its input λ-expression's syntax. Current work: apply partial evaluation to the traversal algorithm for untyped λ-expressions.

Next steps: (a) More on languages, partial evaluation and implementation. (b) Find a way to separate programs from data. Regard a computation of λ-expression M on input d as a *game* between the LLL-codes for M and d. (c) Study the utility of LLL as an intermediate language for a semantics-directed compiler generator.

*JetBrains and St. Petersburg State University (Russia)

†DIKU, University of Copenhagen (Denmark)

[1] Let a *token* be any subexpression of M, the lambda expression being evaluated. A *traversal* is a sequence of occurrences of tokens. Some tokens have *back pointers* to earlier positions in the current traversal. A token may occur more than once, or not at all in a traversal. The size of the traversals: of the order of the length of the expression's head linear reduction sequence.

References

[1] S. Abramsky and G. McCusker. Game semantics. In *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer Verlag, 1999.

[2] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 1–15, 1994.

[3] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

[4] William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logic Methods in Computer Science*, 5(1), 2009.

[5] William Blum and Luke Ong. A concrete presentation of game semantics. In *Galop 2008:Games for Logic and Programming Languages*, 2008.

[6] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[7] Neil D. Jones, editor. *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*. Springer, 1980.

[8] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.*, 11(1):5–94, 2001.

[9] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.

[10] Neil D. Jones and Steven S. Muchnick. The complexity of finite memory programs with recursion. *J. ACM*, 25(2):312–321, 1978.

[11] Neil D. Jones and Steven S. Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*, volume 66 of *Lecture Notes in Computer Science*. Springer, 1978.

[12] Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong. Innocent game models of untyped lambda-calculus. *Theor. Comput. Sci.*, 272(1-2):247–292, 2002.

[13] Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 353–364, 2012.

[14] C.-H. Luke Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.

[15] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[16] David A. Schmidt. State transition machines for lambda calculus expressions. In Jones [7], pages 415–440.