Background Literature and Designs

Michael Buch

December 28, 2018

1 About this document

This is a collection of references and summaries to research in the field of meta-programming/supercompilation/partial evaluation

2 Examples

Examples drawn from paper on collapsing towers [1]:

- Regular expression matcher <- Evaluator <- Virtual Machine
  - Generate low-level VM code for a matcher specialized to one regex (through arbitrary number of intermediate interpreters)
- Modified evaluator <- Evaluator <- Virtual Machine
  - Modified for tracing/counting calls/be in CPS
  - Under modified semantics ”interpreters become program transformers”. E.g. CPS interpreter becomes CPS transformer

3 Methodologies

- Stage polymorphism [2]: “abstract over staging decisions” i.e. single program generator can produce code that is specialized in many different ways (instance of the Fourth Futamura Projection? [3])
- Multi-level base evaluator written in $\lambda\uparrow\downarrow$: supports staging operators (**polymorphic Lift**)
- Modify other interpreters: make them **stage polymorphic**, i.e. commands either evaluate code (like an interpreter) or generate code (like a translator)
- Stage only user-most interpreter: *wire tower* such that the **staging commands in L_n are interpreted directly in terms of staging commands in L_0** i.e. staging commands pass through all other layers handing down commands to layers below without performing any staging commands
- Non-reflective method: meta-circular evaluator **Pink** => collapse arbitrary levels of “self-interpretation”
- $\lambda\uparrow\downarrow$features:

  - *run residual code*
  - binding-time/stage polymorphism [4]
  - preserves execution order of future-stage expressions
  - does not require type system or static analysis
    * TDPE [5] (great explanation also at [6]): **polymorphic Lift** operator turns static values into dynamic (future-stage) expressions

3.1 Towers of Interpreters Project Overview

3.1.1 Scala

- base.scala: implements definitional interpreter for $\lambda\uparrow\downarrow$

4 Results

- Able to achieve compilation of stack-machine on top the Pink evaluator (including tracing evaluator etc.)
- Compilation i.e. collapsing through explicit staging annotations requires intricate knowledge of infrastructure and does not support all data structures e.g. stacks

5 Problems

A useful analogy is the one presented in [1]: a Python interpreter running on a JavaScript emulator of a x86 CPU. What we envision (with reference to this hypothetical setting) is handling the two following cases:

1. A one-off run of a python script on top of this stack should be collapsed by bypassing the emulator interpretation
2. A continuously running emulator evaluating a continuously running python interpreter should collapse individual runs of interpretation while respecting the dynamically changing environment
   - In literature, the closest to compiling a dynamically changing tower is [7, 1] (for a *reflective* language Black) and GraalVM [8]

To tackle the first of these problems we construct a similar yet condensed form of the setting as shown in 1.

6 Contributions

- Extension of the core $\lambda\uparrow\downarrow$to support side effects, combining previous insights into multi-level λ [9] and work on side-effects in partial evaluators [7].
- Development a CESK-style abstract machine/ abstract interpreter for said extended $\lambda\uparrow\downarrow$
- Form a basis for further work on towers by providing a stage polymorphic base evaluator capable of modelling functional or imperative languages

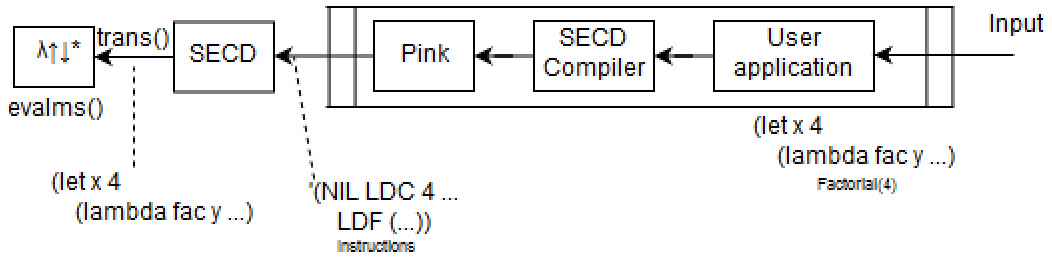

Figure 1: "Effectively functional" $\lambda \uparrow\downarrow^*$with SECD tower above it

- Mimick a practical tower through a SECD machine on top of the base evaluator and show compilation without staging commands throughout the tower
- Theoretical proposal of how one might achieve collapsing in practice

References

[1] N. Amin and T. Rompf, "Collapsing towers of interpreters," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 52, 2017.

[2] G. Ofenbeck, T. Rompf, and M. Püschel, "Staging for generic programming in space and time," in *ACM SIGPLAN Notices*, vol. 52, no. 12. ACM, 2017, pp. 15–28.

[3] R. Glück, "Is there a fourth futamura projection?" in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation.* ACM, 2009, pp. 51–60.

[4] F. Henglein and C. Mossin, "Polymorphic binding-time analysis," in *European Symposium on Programming.* Springer, 1994, pp. 287–301.

[5] O. Danvy, "Type-directed partial evaluation," in *Partial Evaluation.* Springer, 1999, pp. 367–411.

[6] B. Grobauer and Z. Yang, "The second futamura projection for type-directed partial evaluation," *Higher-Order and Symbolic Computation*, vol. 14, no. 2-3, pp. 173–219, 2001.

[7] K. Asai, H. Masuhara, and A. Yonezawa, *Partial evaluation of call-by-value λ-calculus with side-effects.* ACM, 1997, vol. 32, no. 12.

[8] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One vm to rule them all," in *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software.* ACM, 2013, pp. 187–204.

[9] F. Nielson and H. R. Nielson, "Multi-level lambda-calculi: an algebraic description," in *Partial evaluation.* Springer, 1996, pp. 338–354.