Proving Type Soundness

Paul Stansifer

April 15, 2010

Turner demonstrates that the lambda calculus can be compiled to the SKI combinatory calculus, and shows how to evaluate the result efficiently.

Significance: By adding the C, B, and Y combinators (in addition to primitive operations), Turner shows how even a limited graph-reducing machine can be a practical way to implement call-by-need.

Hughes demonstrates how to compile code into supercombinators: individual combinators, representing whole lambda expressions, to be formed into a much smaller code tree.

Significance: Supercombinators tie the combinator graph and source together more closely, and make a number of low-level optimizations possible.
Johnsson describes the G-Machine, a stack machine that describes supercombinators.

**Significance:** The G-Machine structures the implementation of a lazy compiler, simplifies the process of compiling to machine code, and makes certain optimizations possible, especially adding eagerness.

@article{Burn1988,
  author = {Burn, G. L. and Jones, S. L. Peyton and Robson, J. D.},
  journal = {Conference on LISP and Functional Programming},
  keywords = {for-hopl-combcomp},
  mendeley-tags = {for-hopl-combcomp},
  title = {{The spineless G-machine}},
  url = {http://portal.acm.org/citation.cfm?id=62717},
  year = {1988}
}

The Spineless G-Machine is an improvement of the G-Machine that translates the left spine of the graph into a stack, and performs stack operations on it directly. In order to maintain the sharing property of call-by-need, it has to at certain points extract the stack into the graph.

**Significance:** Further performance improvements are made, and the resulting machine made much more “spiritually imperative” (and thus suitable for translation to machine code), even while evaluating a side-effect-free, call-by-need language.