From Stack Traces to Lazy Rewriting Sequences

EXTENDED ABSTRACT

Stephen Chang
Northeastern University

Eli Barzilay
Northeastern University

John Clements
California Polytechnic State University

Matthias Felleisen
Northeastern University

Communicating Author: stchang@ccs.neu.edu

Abstract

Debugging lazy functional programs poses serious challenges. Due to the complicated nature of lazy evaluation, some debugging tools abandon laziness altogether. Other debuggers preserve laziness but present it in a way that may confuse programmers because the focus of evaluation jumps around in a seemingly random manner.

In this paper, we introduce the algebraic program stepper as a new debugging tool for lazy programs. We conjecture that our tool is suitable for clarifying the confusing nature of laziness. Preliminary classroom experiences have confirmed a prototype implementation of the stepper as a useful tool for novice programmers and programmers new to lazy programming.

Mathematically speaking, our stepper renders lazy computations as the standard rewriting sequences of a program rewriting system. Our lazy semantics introduces lazy evaluation as a form of parallel program rewriting. The semantics resembles graph reduction but remains intuitive for programmers because it emphasizes the source syntax. As a *syntactic semantics*, our rewriting system represents a compromise between Launchbury's store-based semantics and a simple, axiomatic description of lazy computation as sharing-via-parameters. We prove an equivalence between our system and both of these semantics.

The stepper's implementation leverages Racket's continuation marks for stack trace generation. We can therefore exploit existing models of continuation marks and a correctness proof of Racket's eager algebraic stepper to prove the correctness of our lazy stepper.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.2 [*Language Classification*]: Applicative (functional) languages; D.3.3 [*Processors*]: Debuggers

General Terms lazy programming; debugging and stepping; lazy lambda calculus

1. How Functional Programming Works

Hughes (1989) explains why lazy functional programming matters. Laziness helps programmers create programs from reusable and composable modules. Unfortunately, laziness also increases the distance between a programmer and the underlying machinery. Specifically, laziness reduces a programmer's ability to predict when certain expressions are evaluated during program execution. As long as things work, this cognitive dissonance poses no problems. When a program exhibits erroneous behavior, however, programmers are often at a loss. A programmer can turn to debugging tools for help, but the evaluation of lazy programs is often confusing enough that some debugging tools resort to hiding laziness from the programmer in order to display useful information (Wallace et al. 2001; Ennals and Peyton Jones 2003; Allwood et al. 2009).

To present an accurate portrayal of laziness, an ideal debugger should not modify the execution model of a program. Some maintainers of GHC (Peyton Jones et al. 1992) seemingly share this sentiment, since the bundled GHCi debugger abides by this ideal. The authors of the GHCi debugger (Marlow et al. 2007) state that their debugger "lets the programmer see the effects of laziness," and therefore, "shows the programmer what is actually happening in their program at runtime." The authors acknowledge, however, that their debugger presents lazy computations in a way that is difficult to follow, mostly due to seemingly random jumps from one place to another in the program.

In this paper we introduce algebraic stepping as a supplemental debugging mechanism for lazy languages. Given a functional program, an algebraic stepper presents the evaluation of the program directly as a manipulation of the source syntax, similar to the algebraic calculations of a student of mathematics. Formally, the manipulations create a rewriting sequence, which in turn, is based on a formal lazy semantics. Based on preliminary classroom experiences, algebraic stepping is an intuitive approach that illuminates the nature of lazy evaluation. Our experience especially suggests that this kind of tool benefits novice programmers when they try to understand small programs, and programmers who are new to a language and are trying to explore some linguistic feature.

PLT's DrRacket (Findler et al. 2002) comes with an algebraic stepper (Clements et al. 2001) for the call-by-value Racket language. This stepper utilizes Racket's continuation marks, a portable, lightweight stack inspection mechanism, to generate a stack trace from which the by-value reduction sequence is then reconstructed. We leverage this existing framework to build a stepper for Lazy Racket (Barzilay and Clements 2005), an untyped call-by-need language that uses the same evaluation mechanism as Haskell. A Lazy Racket program macro-expands its surface syntax into a plain Racket program enriched with appropriate `delay` and `force` constructs (Hatcliff and Danvy 1997). Lazy Racket is used primarily in educational settings, where we have tested a prototype of the stepper so far. Since continuation marks are easily ported to any language runtime, the techniques presented in this paper are applicable to any language and are not specific to Racket.

The final version of this paper will present: (1) the stepper for Lazy Racket; (2) its underlying semantics, a novel lazy program rewriting system; (3) and a proof that the stepper correctly implements the standard rewriting semantics of the system. Our novel lazy rewriting system introduces the idea of using *parallel* reductions to simulate *shared* reductions. The system resembles graph reduction, except we emphasize reductions on the source syntax, which is more intuitive for programmers, instead of first compiling

to combinators. As a syntactic semantics, our system is a compromise between Launchbury's natural semantics (Launchbury 1993) and Ariola et al.'s call-by-need λ-calculus (Ariola et al. 1995). Our semantics operates at a higher-level of abstraction than the natural semantics, because we don't have an explicit store. Our semantics is also easier to understand than Ariola et al.'s calculus, on which Gibbons and Wansbrough (1996) built a lazy debugging tool, because the calculus includes unusual reshuffling reductions that are not present in any lazy language implementations.

Besides being more intuitive than existing semantics, our rewriting system has two additional advantages. First, there exists an equivalence between the standard rewriting semantics of our system and both Launchbury's and Ariola et al.'s semantics, and as part of our correctness proof, we demonstrate these correspondences. Second, our rewriting system provides a convenient basis for a correctness proof of the stepper. We consider correctness proofs for debugging tools nearly as important as proofs for compilers, because both are critical elements of a programmer's tool chain. For the correctness proof of our stepper, we construct a model of the `delay`-and-`force` implementation, further enriched with continuation marks (Clements et al. 2001), show that it bisimulates the standard rewriting semantics, and finally, exploit a strategy from Clements (2006) for the rest of the proof.

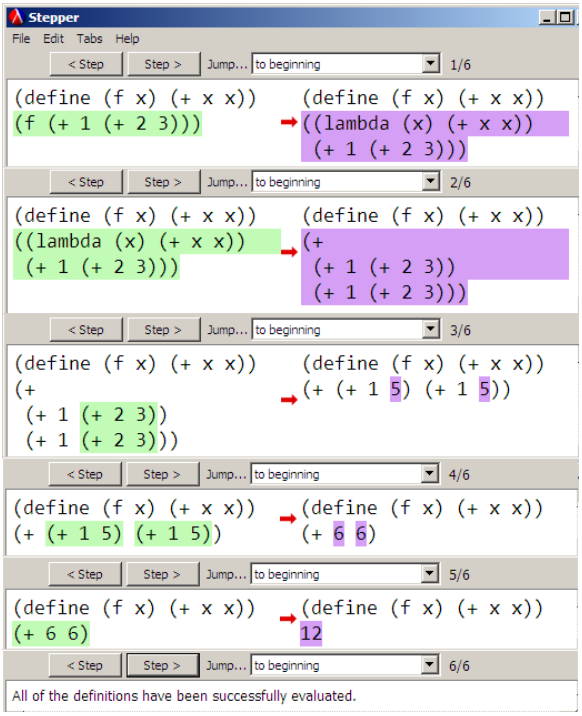

Figure 1. Lazy Stepper Example

2. Lazily Stepping Through a Small Example

Figure 1 shows the sequence of steps that the Lazy Racket stepper displays for the following simplistic program:

```
#lang lazy

(define (f x) (+ x x))
(f (+ 1 (+ 2 3)))
```

The Lazy Racket stepper comes with the DrRacket IDE. When a programmer invokes the stepper, DrRacket brings up a separate window of the program with navigation buttons that allow the programmer to browse through a sequence of images. Each image corresponds to one step in the reduction sequence.

A green box in an image indicates the location of the current redex on the left-hand side of a reduction step. Similarly, a purple box highlights the contractum on the right-hand side. The surroundings of a redex is a context, i.e., a term with potentially several holes. Each of the holes contains a reference to the current machine instruction. The context with respect to the leftmost-outermost hole is a textual reconstruction of the lazy stack trace. The formal model of the stepper explains this relationship in detail and also shows how the stepper handles more advanced language features like higher-order functions and cyclic structures.

Let us walk through the sequence in figure 1 in some detail. In step 2, evaluation of the function argument is delayed, meaning the unevaluated argument replaces each instance of the variable `x` in the function body of `f`. Next the strictness of `+` demands the evaluation of the summand `(+ 2 3)`, a part of the unevaluated argument that replaced the leftmost occurrence of `x`. Step 3 shows how the expression is forced in two different places and step 4 shows a second instance of this form of simultaneous replacement.

Since the second occurrence of `x` refers to the same delayed computation, its value is already available when the time comes to evaluate it. At this point, it is trivial to compute the final outcome of the function call; see step 5. In short, *the stepper explains evaluation as an algebraic process using a form of parallel reduction*.

References

T. O. Allwood, S. Peyton Jones, and S. Eisenbach. Finding the needle: stack traces for GHC. In *Proc. 2nd Symp. on Haskell*, pages 129–140, 2009.

Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. In *Proc. 22nd Symp. on Principles of Programming Languages*, pages 233–246, 1995.

E. Barzilay and J. Clements. Laziness without all the hard work. In *Proc. Works. on Functional and Declarative Programming in Education*, pages 9–13, 2005.

J. Clements. *Portable and High-level Access to the Stack with Continuation Marks*. PhD thesis, Northeastern University, 2006.

J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *Proc. 10th European Symp. on Programming*, pages 320–334, 2001.

R. Ennals and S. Peyton Jones. HsDebug: debugging lazy programs by not being lazy. In *Proc. Works. on Haskell*, pages 84–87, 2003.

R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *J. of Functional Programming*, 12(2):159–182, 2002.

J. Gibbons and K. Wansbrough. Tracing lazy functional languages. In *Proc. Computing: The Australasian Theory Seminar*, pages 11–20, 1996.

J. Hatcliff and O. Danvy. Thunks and the λ-calculus. *J. of Functional Programming*, 7(3):303–319, 1997.

J. Hughes. Why functional programming matters. *Computer J.*, 32(2):98–107, 1989.

J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th Symp. on Principles of Programming Languages*, pages 144–154, 1993.

S. Marlow, J. Iborra, B. Pope, and A. Gill. A lightweight interactive debugger for Haskell. In *Proc. Works. on Haskell*, pages 13–24, 2007.

S. Peyton Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview, 1992.

M. Wallace, O. Chitil, T. Brehm, and C. Runciman. Multiple-view tracing for Haskell: a new hat. In *Proc. Works. on Haskell*, pages 151–170, 2001.