HEAD-ORDER TECHNIQUES AND OTHER PRAGMATICS OF LAMBDA CALCULUS GRAPH REDUCTION

Nikos B. Troullinos

DISSERTATION.COM

Boca Raton

Head-Order Techniques and Other Pragmatics of Lambda Calculus Graph Reduction

Dissertation.com
Boca Raton, Florida
USA • 2011

ISBN-10: 1-61233-757-0
ISBN-13: 978-1-61233-757-9

Abstract

In this dissertation Lambda Calculus reduction is studied as a means of improving the support for declarative computing. We consider systems having reduction semantics; i.e., systems in which computations consist of equivalence-preserving transformations between expressions. The approach becomes possible by reducing expressions beyond weak normal form, allowing expression-level output values, and avoiding compilation-centered transformations. In particular, we develop reduction algorithms which, although not optimal, are highly efficient.

A minimal linear notation for lambda expressions and for certain runtime structures is introduced for explaining operational features. This notation is related to recent theories which formalize the notion of substitution.

Our main reduction technique is Berkling's Head Order Reduction (HOR), a delayed substitution algorithm which emphasizes the extended left spine. HOR uses the de Bruijn representation for variables and a mechanism for artificially binding relatively free variables. HOR produces a lazy variant of the head normal form, the natural midway point of reduction. It is shown that beta reduction in the scope of relative free variables is not hard. Full normalization suggests new applications by not relegating partial evaluation to a meta level. Variations of HOR are presented, including a conservative "breadth-first" one which takes advantage of the inherent parallelism of the head normal form.

A reduction system must be capable of sharing intermediate results. Sharing under HOR has not received attention to date. In this dissertation variations of HOR which achieve sharing are described. Sharing is made possible via the special treatment of expressions referred to by head variables. The reduction strategy is based on normal order, achieves low reduction counts, but is shown to be incomplete.

Head Order Reduction with and without sharing, as well as other competing algorithms are evaluated on several test sets. Our results indicate that reduction rates in excess of one million reductions/second can be achieved on current processors in interpretive mode and with minimal pre- and post-processing. By extending the efficient algorithms for the pure calculus presented in this dissertation with primitives and data structures it is now possible to build useful reduction systems. We present some suggestions on how such systems can be designed.

Dedication

Στους γονείς μου, Μαίρη και Βασίλη

Preface

Foremost among all my teachers, Klaus Berkling taught me to question every assumption and not to rest until all aspects of a problem have been considered. For the last five years he has guided me through the maze of the practical aspects of Lambda Calculus reduction. With his judgment he reinforced my conviction that minimality and elegance are not luxuries but bare necessities.

Improving the practicality of Michael Hilton's *THOR/HORSE* was a strong influence and a well-defined first goal. It turned out that the problems are of a greater scope than I anticipated. Our interaction during the early stages of my study is greatly appreciated.

Inspiration, and the incentive for edging a bit closer to the "science" of computer science, was provided by the classes, seminars and talks that I was exposed to during my long career as a graduate student at Syracuse University. Ken Bowen, Lockwood Morris, John Reynolds, Alan Robinson, Ernest Sibert and Ed Storm come to mind, among many others. Academic committee members, Shiu-Kai Chin and Peter O'Hearn provided valuable feedback at every step.

Chuck Stormon and Coherent Research, Inc. afforded me the uncommon opportunity of continuing my graduate work while helping forge the future of this remarkable company. All my colleagues at CRi are to be thanked for their continuous interest, encouragement and good cheer.

Bruce Berra, Gideon Frieder and Brad Strait kindly provided financial assistance during my days at ECE, CIS and at the CASE Center. Their support is gratefully acknowledged.

From one-third across the globe away, loving parents Mairi and Vasilis were a vital source of encouragement, support and patience; enterprising sister Maria ensured that I stayed focused and balanced most of the time; Liebling Susanne E.(mail) Bohl was instrumental in making the journey more enjoyable. Lightheartedly, some credit also goes to a certain telecommunications company whose tariffs made it a bit more affordable to live and work in one continent while having so many of one's "friends & family" in another.

Table of Contents

List of Figures

List of Tables

Chapter 1

Lambda Calculus and Reduction Computing

1.1 Introduction - Motivation

An intriguing direction in symbolic computing is to base an architecture (hardware or software) on equivalence preserving transformations of expressions. In this dissertation we show that this viewpoint can be made practical. The computational model is the Lambda Calculus—a simple model which views functions as rules. Our motivation is the belief that direct Lambda Calculus reduction can be done efficiently. Beta reduction in the scope of relative free variables is not difficult with proper representations. Strong normalization[1] opens up new applications by not relegating abstract interpretation to a meta level.

Berkling's Head Order Reduction (HOR) is the main tool of this approach. It is a delayed substitution algorithm with emphasis on the extended left spine. It uses the *de Bruijn* coding for variables, *unbound variable counts* for "neutralizing" relatively free variables and proceeds to an operational variant of the natural midway point of reduction, the *head normal form*. The machinery of Head Order Reduction appears to be a direct and minimal match to the requirements of strong reduction. Among its operational properties are extremely low space requirements and independence from pointer representations for variables. Michael Hilton's *THOR/HORSE,* an applied derivative system, is a good example of the overall direction and goals.

[1] In this thesis this term simply means that reduction proceeds to full β-normal form.

THOR/HORSE has demonstrated that systems based on the preceding principles are not only feasible but can also attain levels of efficiency previously thought unreachable. Hence, we downplay transformation techniques like combinators and strong typing that are commonly employed solely for the purpose of compilation to von Neumann code. It is our belief that specialized hardware or a low-level software "interpreter" for a proper operational casting of the Lambda Calculus can serve large classes of problems well, both in terms of expressiveness and efficiency.

But a pure normal-order strategy like that of *THOR/HORSE* is unacceptable for practical work if it is not augmented with sharing. A large part of our efforts concentrates on augmenting HOR with sharing. Initially, the task seemed intractable; early reducers were prone to subtle binding errors. As our understanding improved, we realized that no single technique is a complete solution to the problem and free of disadvantages. But progress has been made. By utilizing combinations of essential techniques provided by Berkling's earlier work the state of affairs described in this dissertation was reached. The initial goal of adding effective full sharing in a system having the scope of *THOR/HORSE* is unfulfilled.

The essentials of Head Order Reduction were implicitly present as early as 1972 in de Bruijn's *Indagationes Mathematicae* publication of a Lambda Calculus with integer "Nameless Dummies" instead of the usual named variable identifiers. Retrospectively, one can state that the logical progression that leads to the operational HOR is the addition of delayed substitutions via an environment and the emphasis on head normal forms. Historically, the development took a different course via Berkling's "in-the-large" reduction rules presented for the first time at the *Santa Fe Graph Reduction Workshop* in 1986.

The quest for grasping the interrelationships and intricacies of reduction methods led us to review several of the well-known weak and strong algorithms. Our attention was limited to the pure, untyped Lambda Calculus to keep complications to a minimum. It turned out that the weak form of HOR is identical in strategy with the so called "RTLF" strategy of Wadsworth and Aiello–Prini and other less well known algorithms. Detailed comparisons of all the popular algorithms demonstrate the directness and efficiency of HOR.

Our work represents a shift in the point of view of traditional research on functional programming. It is based on the realization that the Lambda Calculus, which is commonly regarded as one of the foundations of programming, is not intrinsically oriented towards producing a value. Therefore, not all languages and systems should be limited in this regard. We believe that much may be gained in the realm of software technology if the first class objects are full expressions rather than ground values of some output domain.

We attempt to derive architectural principles while remaining close to the inherent mechanisms suggested by the underlying theory. The sort of computations that benefit from the existence of a reduction system are those in which one is not only interested in getting a final ground result but also insists that the steps taken in arriving at such result are as transparent and direct as possible. If this line of thinking is followed, then the Lambda Calculus appears indeed to be a solid choice because most of the thorny issues surrounding declarative computation of the functional kind surface here in their purest form. How practical or influential this path may be is something that will be determined eventually but for the moment it seems exciting work to pursue.

Combinator techniques are sidestepped because we regard them as indirect. With combinators one forgoes the random reference ability of variables and replaces it by argument propagation systems. From an operational standpoint this is cumbersome and circuitous. In addition, such techniques make it very difficult to do *incremental* reduction and to produce recognizable intermediate results. The efficiency of reducers like HOR may be largely attributed to using binding indices for variables together with a correction mechanism for dynamically-created abstractions that operates in an incremental and local manner. This way one can avoid the problems of scope and name capturing of variables and thus render the detour via combinators much less attractive.

Our overall emphasis is, by necessity, the invention of reduction schemes which are economical from a practical rather than a theoretical standpoint. Hence, we reject the usual notion of optimality—which equates excess work to excess reductions—as too limiting and not very helpful for reduction system design. The Lambda Calculus can model situations of arbitrary complexity. A reduction strategy which always avoids excess reductions requires bookkeeping which is more substantial than the effort required for performing "excess" reductions. A more reasonable goal is to attempt to match or exceed the natural savings of applicative order while avoiding, if possible, its incompleteness. Opportunities for parallelism are plentiful, but again, the orchestration overhead can outweigh the benefits derived over simpler sequential tactics. Source program transformations can help in exposing the inherent parallelism and research in this area seems to us worthwhile.

The rest of this chapter highlights the reduction-based viewpoint and pinpoints the challenges that are present. Results are summarized in section 1.7.

1.2 Operational Lambda Calculus as Computational/Machine Model

Attempting to derive architectural principles from the operational characteristics of Lambda Calculus reduction is a slightly unconventional line of research. It could be likened to an aspiration to advance the techniques of modern microprocessor design by examining closely the workings of a Turing Machine. But the parallel breaks down—and hence the former does not seem as unconvincing as the latter—if one considers that a Lambda Calculus expression is a higher level of expression than, let us improvise, the transitions rules of a

Turing Machine and that it should impart more structure to a mechanization procedure. Additionally, the declarative nature of the Lambda Calculus is in sharp contrast with the sequential instructions and state-oriented nature of the von Neumann model. Consequently, since the semantic gap to be covered is larger, the journey could be more rewarding.

Lambda Calculus as Basis of a Reduction System

Our interest in the Lambda Calculus stems from two basic premises: First, research efforts on making high-order declarative languages work better and serve larger classes of practical computing tasks should intensify[2]. Second, a significant portion of our current difficulties with the building of software is attributed at to the low-level nature of the popular programming language and environments and the batch style of most of real world development tools. Higher-order programming, symbolic computation, partial evaluation and incremental development hold serious promises in advancing common computing practice[3]. Programming environments could become more fluid and the so called "Computer-Aided Software Engineering" tools would be able to focus more on the actual artifact.

One could regard computation as proceeding under two rather distinct flavors. First, functions are applied to arguments in a pure sense by fusing them together which means propagating arguments to the proper places at the appropriate time. Second, ground data objects, like encodings for numbers, characters or graphics primitives, are transformed in well-defined and predesigned ways; for example, multiplying two integers together or shifting the value of a register by a fixed amount. Symbolic computation often has a stronger first component while "number crunching" is nearly a pure form of the second. The Lambda Calculus is one of the simplest settings where one can experience these two flavors in a distilled manner and offers the opportunity of discovering techniques that may be useful in more general settings. The considered application areas are reduction languages, semantics-directed machine architecture and high-level software development tools.

One cannot avoid touching upon the potential criticism that the Lambda Calculus is inherently a model of *sequential* computing and hence efforts based on it may not be as applicable to future systems[4]. Exploring concurrent computation in a similar spirit, starting

[2] We are particularly interested in function-based languages. But efficient Lambda Calculus reduction can have implications for languages with side effects due to the generality of the concepts of abstraction and application.

[3] The literature has a wealth of research on each of these topics and it is outside of the scope of this thesis to attempt further explications. For high-order functional programming see, for instance, the paper by J. Hughes [Hughes 90] and the proposal by J. Backus et al [Backus 90].

[4] The major difficulty is that from a foundational viewpoint the pure calculus can not "escape" an undefined argument of a disjunction; i.e., there is no mechanism for expressing

maybe with Milner's *Calculus of Communicating Systems* (*CCS*) [Milner 80] or Hoare's *Communicating Sequential Processes* (*CSP*) [Hoare 85] or some other concurrent process modeling theory, would have been closer to recent trends. The author feels though, that progress remains to be made at the level of sequential expression. As it will become evident in section 8.4, the core of the Head-Order strategy has important implications for automatic *run-time* parallelism given the structural invariance of the operational head normal forms.

An Example of Reduction Semantics

A reduction system is envisioned as a delivery platform for exploratory programming environments of declarative or well-structured imperative nature. Solid representation choices and direct support from a specialized instruction set can result in a high-level organization, compared to the von Neumann one. Compilation is limited to optimizing but equivalence-preserving transformations on portions of a program that consist of arithmetic, logical and other low-level hardware-supported operations.

To demonstrate this sort of system let us visit a well-known high-order function like *reduce*. In the syntax of *THOR, reduce* has a definition identical to that of a *Scheme* function. It also behaves identically to a *Scheme* function as seen in the first and second examples of Table 1.1. But a vital, and commonly considered prohibitively expensive, change of semantics is evident in the third and fourth runs. When *reduce* is supplied with fewer arguments than those implied by its definition or with enough but unbound ones, *HORSE* (the programming environment for *THOR*) is perfectly content to perform the available reductions without any of the usual error messages. This is akin to abstract interpretation but it is accomplished with nothing more than the ordinary reduction mechanisms of the underlying calculus; there is nothing "abstract" or "meta" about such behavior—it is simply that the rules of the underlying calculus are followed to a greater extent.

A major theme of our work is that the cost of such fluidity is not as high as commonly thought; the proviso is, of course, that efficient reduction algorithms and systems are available. Fundamentally, not much is new or radical with this approach. The novelty is the recognition of the importance of *expression-based* computation which advances beyond weak normalization.

---

the meaning of a "parallel or". Applied variants of the pure calculus like *THOR* can and do include such mechanisms by treating, for instance, truth values specially. At this level there is no such constraint; in fact, the side-effect free nature of the calculus makes it relatively easy to add parallelism. Graph transformations, the natural mechanism for reduction, do not impose any sequentiality by themselves.