ON THE RELATIONSHIP BETWEEN LAZINESS AND STRICTNESS

STEPHEN CHANG

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

to the

Faculty of the
College of Computer and Information Science
Northeastern University
Boston, Massachusetts

May 2014

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS APPROVAL FORM

THESIS TITLE: On the Relationship Between Laziness and Strictness

AUTHOR: Stephen Chang

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

Thesis Advisor	*Date* 14 May 2014
Thesis Reader	*Date* 5/14/2014
Thesis Reader	*Date* 5/14/2014
Thesis Reader	*Date* 5/14/2014

GRADUATE SCHOOL APPROVAL:

Director, Graduate School	*Date* 5/15/2014

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature	*Date* 5/20/2014

Distribution: Once completed, this form should be scanned and attached to the front of the electronic dissertation document (page 1). An electronic version of the document can then be uploaded to the Northeastern University-UMI website.

Dedicated to Jen, Stella, and Evan.

The computing scientist's main challenge is not to get confused by the complexities of his own making.

— E. W. Dijkstra

ABSTRACT

Lazy evaluation empowers programmers with the ability to create efficient, modular programs. Unfortunately, laziness also comes with some drawbacks. On one hand, lazy constructs incur time and space overhead so programmers should minimize their use when they are not needed. Thus, lazy-by-default languages are often not the best means to utilize laziness. On the other hand, programs with too little laziness may pay its costs without receiving any benefit in return. This is because laziness is a global property of programs and thus its effects are not immediately proportionate to the amount of laziness in the program. Specifically, laziness propagates according to a program's value flow, so for example, introducing laziness in one part of a program may require additional laziness in other parts before any benefits are realized. Thus, determining the appropriate amount of laziness for a program is a tricky endeavor that requires balancing the costs with the benefits.

Researchers have proposed various strategies for determining a balance of laziness and strictness in a program. Unfortunately, most previous strategies have failed to appropriately consider either the costs or the benefits involved. Thus the preferred way to manage laziness in practice today is via manual annotations, which is an error-prone task that requires extensive experience to get right. In this dissertation, we introduce two techniques, one static and one dynamic, that take a strict-first perspective on managing laziness and improve on the state of the art in two ways. First, our techniques weigh the costs of laziness against the benefits with models that capture the expertise typically required when reasoning about laziness placement, and thus our techniques more precisely identify which parts of a program should be strict and which should be lazy. Second, we implement concrete tools that utilize our strategies so programmers can compute where laziness is needed in a more automated manner.

ACKNOWLEDGMENTS

First and foremost, I owe a debt of gratitude to my advisor, Matthias Felleisen, who saw a PhD student in me and gave me a chance when I did not deserve one. Of course, this is unsurprising for someone who has been a visionary leader in the field for the past three decades. As a mentor, Matthias goes above and beyond the call of duty when teaching his students about the art of research. Most remarkably, he does so in the most selfless manner possible, readily accepting blame for students' failures but eagerly bestowing credit for successes. I am privileged to have worked with such an extraordinary person.

Also, this dissertation would not exist if not for the support I received from the Northeastern PRL group. The first question that a visiting prospective student always asks is, "what separates Northeastern's PL program from the others?" The unequivocal answer is the collaborative nature of its members. If someone is working towards a paper deadline, everyone else is reading drafts. If another lab member is preparing a talk, everyone else drops what they are doing and "tortures" the talk until it's ready. In no particular order, thanks to Asumu Takikawa, Vincent St-Amour, Ian Johnson, Jonathan Schuster, Christos Dimoulas, Dan Brown, Aaron Turon, Zane Shelby, Tony Garnock-Jones, Stevie Strickland, Carl Eastlund, Sam Tobin-Hochstadt, David Van Horn, Eli Barzilay, Phillip Mates, Billy Bowman, Jamie Perconti, Jesse Tov, Justin Slepak, Claire Alvis, Bryan Chadwick, Ryan Culpepper, Felix Klock, Erik Silkensen, Paul Stansifer, Ahmed Abdelmeged, Dimitris Vardoulakis, Mitch Wand, Olin Shivers, Amal Ahmed, Will Clinger, Riccardo Pucella, and Viera Proulx.

Thanks to my coauthors, David Van Horn, John Clements, and Eli Barzilay, for guidance and patience while I learned how to write a research paper. Thanks to my thesis committee, Amal Ahmed, David Van Horn, Eli Barzilay, and Greg Morrisett, who provided lots of helpful feedback on my ideas. Thanks to Greg Morrisett and Norman Ramsey, whose courses introduced me to the exciting world of programming language research.

Thanks to my parents, who are my heroes and have always supported me no matter what. If I can achieve a mere fraction of what they have accomplished, then my life can be considered a success.

Finally, thanks to my family Jen, Stella, and Evan. Jen, thanks for being my inspiration and for making me strive to be a better person in all aspects of life. Stella and Evan, thanks for providing meaning and purpose in life, in a way that I previously did not know was possible. I love you all.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

BALANCING LAZY AND STRICT EVALUATION

1 INTRODUCTION

1.1 THE ADVANTAGES AND DISADVANTAGES OF LAZINESS

Experts have extolled the benefits of laziness for decades now. Landin introduced the idea of streams in 1965 [45] and later, Steele and Sussman's Lambda papers [75] popularized the notion that strict programs can profit from such lazy constructs. Indeed, since then many well-known algorithms and data structures implemented in strict languages have taken advantage of laziness [27, 35, 36, 62].

In the mid-80s, lazy functional *languages* began to grow in prominence. As Hughes explained [40], lazy functional languages naturally support the construction of modular reusable components and their composition into reasonably efficient programs. For example, the solution to a puzzle may consist of a generator that produces an easily-constructed stream of all *possible* solutions and a filter that extracts the desired *valid* solutions. Due to laziness, only the relevant portion of the possible solutions are explored. Put differently, lazy composition appears to naturally recover the desired degree of efficiency without imposing a contorted programming style.

Eventually, enthusiasm in lazy evaluation waned as researchers began to understand its drawbacks [38, 49]. Lazy evaluation in practice makes it difficult to reason about resource consumption. It also potentially incurs significant resource overhead.

All hope is not lost, however, because researchers also observed that most programs actually need only a small amount of laziness [20, 23, 51, 55, 69]. Thus, the new research focus is to find a balance of laziness and strictness that maximizes the benefits while minimizing the costs. Specifically, the lazy language community offers many strategies for determining when to automatically remove laziness [20, 23, 51, 57] from lazy programs, though with mixed success. In particular, all lazy languages today deploy some form of strictness analysis [16, 57]. However, it often leaves too much laziness behind and thus, lazy programmers more often than not resort to manually managing strictness via strictness annotations such as Haskell's `seq` [63], which requires significant experience and expertise to get right.[1].

Modern strict programming languages continue to recognize the benefits of laziness as well, and most enable programmers to wield laziness in some

1 The introduction of `seq` also changes the observational equivalence relation of the language though we ignore this problem.

form, either via data structure libraries or primitive lazy annotations such as `delay` and `force`, as pioneered by Scheme [68] and adopted by ML [2]. However, few to none of these languages provide any additional guidance on how to use these constructs, and deploying laziness in strict languages requires extensive expertise, just like managing strictness in lazy languages.

Thus, the issue of how best to deploy laziness in practice remains an open problem. Deploying laziness is a tricky endeavor because on one hand, inserting too little laziness results in no benefits. Specifically, laziness propagates according to value flow in a program, so inserting laziness at one location may spawn additional laziness requirements elsewhere in the program. If these demands are not met, then the addition of laziness merely increases the overhead costs. On the other hand, inserting too much laziness may yield some benefit, but may also incur excessive resource overhead which may eliminate the benefits of laziness altogether. In other words, utilizing laziness in practice is a delicate balancing act that requires significant expertise, which can take years of experience to acquire.

1.2 LAZINESS FLOW

1.2.1 *Propagating Delayed Function Arguments*

The following Scala [61] code snippet from a 2009 blog post[2] illustrates one instance of the problem of how programmers can easily insert too little laziness, and thus achieve no benefits. Scala delays method arguments whose type is marked with `=>`, as in:[3]

```
def foo[A,B](a: A, b: => B): B = ...
```

When `foo` is called, its second argument is not evaluated until its value is needed inside the function body. However, if another function, `bar`, calls `foo`:

```
def bar[C,A,B](c: C, a: A, b: B): B = { ... foo(a, b) }
```

the `b` argument is evaluated when `bar` is called, thus negating the benefit of laziness in `foo`. To recover it, we must delay the third argument to `bar`:

```
def bar[C,A,B](c: C, a: A, b: => B): B = ...
```

If yet another function calls `bar` then that function must delay its argument as well. For programs with complex call graphs, the required delay points may be scattered throughout the program, making programmer errors more likely.

2 http://pchiusano.blogspot.com/2009/05/optional-laziness-doesnt-quite-cut-it.html

3 The `=>` syntax specifies "by-name" parameter passing for this position but the distinction between "by-name" and "lazy" is inconsequential here.

1.2.2 *Reorganizations Interfere with Laziness*

Laziness flow can also introduce subtle bugs during code refactoring. This subsection tells a recent story involving the Racket [29] language and a senior Racket developer.

A game tree is a data structure representing all possible sequences of moves in a game. It is frequently employed in AI algorithms to calculate an optimal next move, and it is also useful for game developers wishing to experiment with the rules of a game. For anything but the simplest games, however, the multitude of available moves at each game state results in an unwieldy or even infinite game tree. Thus, laziness is frequently utilized to manage such trees.

The Racket code to generate a game tree might look like the program in figure 1. A game tree is created with the `gen-GT` function, which takes a game state and the current active player. If the given state is a final state, then a `GT-Leaf` node is created. Otherwise, a `GT-Node` is created with the current game state, the current player, and a list of moves from the given game state. The `calc-next-moves` function creates a list of `Move` structures, where each move contains a new game tree starting from the game state resulting from the move.

A recent Racket-based programming book, Realm of Racket [26] utilizes such a game tree. Initially, only a small game is implemented, so `Move` is defined as a strict constructor. As the book progresses, however, the game tree becomes unwieldy as more features are added to the game. In response, the third argument of the `Move` structure is delayed, meaning the call to the `Move` constructor implicitly wraps the third argument with a `delay`.[4] With the lazy `Move` constructor, the code above generates only the first node of a game tree.

To prepare the book for typesetting, an author reorganized the definition of `calc-next-moves` in a seemingly innocuous fashion to fit it within the margins of a page:

The underlined code pulls the generation of the game tree into a separate definition. As the astute reader will recognize, the new game tree is no longer created lazily. Even though the `Move` constructor is lazy in the third position, the benefits of laziness are lost. Even worse, such a performance bug is easily unnoticed because the program still passes all unit tests (especially because the same test suite is used for both the strict and the lazy version).

4 Specifically, `Move` becomes a macro that expands to a private constructor call where the third argument is delayed. This is a common idiom in Lisp-like languages [30].

```
;; A GameTree (short: GT) is one of:
;; -- (GT-Leaf GameState)
;; -- (GT-Node GameState Player [ListOf Move])

;; A Move is a (Move Name Position GameTree)

(struct GT-Leaf (state))
(struct GT-Node (state player moves))
(struct Move (name pos gt))

;; gen-GT : GameState Player -> GameTree
(define (gen-GT game-state player)
  (if (final-state? game-state)
      (GT-Leaf game-state)
      (GT-Node game-state
               player
               (calc-next-moves game-state player))))

;; calc-next-moves : GameState Player -> [ListOf Move]
(define (calc-next-moves game-state player)
  ⟨⟨for each possible attacker and target in game-state⟩⟩:
     (define new-state ...)
     (define new-player ...)
     (Move attacker target (gen-GT new-state new-player)))
```

Figure 1: Racket Program That Utilizes a Game Tree.

```
;; calc-next-moves : GameState Player -> [ListOf Move]
(define (calc-next-moves game-state player)
  ⟨⟨for each possible attacker and target in game-state⟩⟩:
     (define new-state ...)
     (define new-player ...)
     (define new-gt (gen-GT new-state new-player))
     (Move attacker target new-gt))
```

Figure 2: Refactored Game Tree Code.

1.3 TOO MUCH LAZINESS

It is well understood now that laziness is not desirable all the time. The use cases where laziness is particularly burdensome, like iteration using accumulators [51, 83], are well known. This subsection demonstrates that even canonical uses of laziness can benefit from less laziness. Specifically, we present an implementation of the n-queens program, a generate-and-filter-style example that precisely benefits from the style of modular, lazy programming advocated by Hughes.

The *n*-queens problem makes an illustrative playground for advertising lazy programming because an idiomatic lazy solution to such a puzzle may consist of just two parts: a part that places *n* queens at arbitrary positions on an *n* by *n* chess board, and a part for deciding whether a particular placement is a solution to the puzzle. Given these two components, a one-line function calculates a solution:

```
let nqueens n = hd (filter isValid all_placements)
```

The `all_placements` variable stands for a stream of all possible placements of n queens; `filter isValid` eliminates placements with conflicting queens; and `hd` picks the first valid one. Lazy evaluation guarantees that `filter isValid` traverses `all_placements` for just enough placements to find the first solution.

The approach cleanly separates two distinct concerns. While the code for generating `all_placements` may ignore the rules of the puzzle, it is the task of `isValid` to enforce them. If the components were large, two different programmers could tackle them in parallel. All they would have to agree on is the representation of queen placements, for which we choose a list of board coordinates (r, c). Here is `all_placements`, in OCaml-style functional pseudocode:

```
let process_row r qss_so_far =
  foldr
    (fun qs new_qss ->
      (map (fun c -> (r,c)::qs) (rng n)) @ new_qss)
    []
    qss_so_far

let all_placements = foldl process_row [[]] (rng n)
```

Brackets denote lists, `rng n` is `[1...n]`, `::` is infix `cons`, and `@` is infix append. All possible placements are generated by adding one coordinate at a time.

The `process_row` function, given a row r and a list of placements *qss_so_far*, duplicates each placement in *qss_so_far* n times, adding to each copy a new coordinate of r with a different column c, and then appends all these new placements to the final list of all placements. The `process_row` function is called n times, once per row. The result of evaluating all possible placements would look like this:

```
[[(n,1);(n-1,1); ... ;(1,1)];
   ...;
 [(n,n);(n-1,n); ... ;(1,n)]]
```

where each line represents one possible placement.

Due to laziness, only a portion of the possible placements are computed, until a solution is found. Indeed, Haskell[5] returns a solution in a reasonable amount of time:[6]

```
real 0m0.130s    user 0m0.128s    sys 0m0.000s
```

Here are the timing results for the same Haskell n-queens program, however, with most of the laziness removed, demonstrating that the program still uses too much laziness:

```
real 0m0.104s    user 0m0.100s    sys 0m0.000s
```

Unfortunately, just like inserting laziness, knowing where to remove laziness is a complex problem. To get the 20% speedup above, we had to insert numerous `seq`s and bangs (`!`) [33] into the program. It is clearly tedious for programmers to determine all these places, and it typically requires significant experience and expertise to get right.

1.4 WHEN TO BE LAZY

So far, we have seen that both the strict-first and lazy-first approaches to using laziness pose significant problems. What is a programmer who wishes to utilize laziness to do? This dissertation addresses the problem of programming lazily in strict languages with two analysis techniques, one static and one dynamic. This section illustrates these techniques with a further exploration of the n-queens example.

5 Compiled with GHC 7.4.1 with -O2 optimizations to include a strictness analyzer.
6 Run on an Intel i7-2600k, 16GB memory machine.

1.4.1 n-*queens, Strictly*

We use the same n-queens example as before, except implemented in Racket [29]. Since Racket is strict, using `all_placements` with the `nqueens` function from earlier generates all possible placements before testing each one of them for validity. This computation is obviously time consuming and performs far more work than necessary. For reference, here is the timing for $n = 8$ queens:[7]

```
avg cpu time: 11676ms  avg real time: 11723ms  avg gc time: 7036ms
```

We know that we can speed up this strict program by adding some laziness. But where should the laziness be placed? If the programmer switches to lazy lists to represent `all_placements` (as strict programmers looking to utilize laziness often do [62]), then only a portion of the possible placements should be explored. Specifically, all instances of `cons` (`::`) are replaced with its lazy variant, represented with $::_{lz}$ below. It is also necessary to add `force`s where appropriate.[8] For example, here is append (`@`) and `map` with lazy `cons` (`[]` also represents the empty lazy list). For clarity, we keep the `delay`s explicit.

```
let rec (@) lst1 lst2 =
  match force lst1 with
  | [] -> lst2
  | x::lzxs -> x::lzdelay (xs @ lst2)

let rec map f lst =
  match force lst with
  | [] -> []
  | x::lzxs -> f x::lzdelay (map f xs)
```

Running this program surprises our lazy-strict programmer:

```
avg cpu time: 15068ms  avg real time: 15111ms  avg gc time: 9628ms
```

With lazy `cons` and `force`, the program runs even slower than the strict version. Using lazy `cons` naïvely does not seem to generate the expected performance gains. Additional `delay`s and `force`s are required, though it is not

7 Note that the times in this section cannot be compared to the previous section, as far as determining the best placement of laziness is concerned. The absolute times in this section are greater than those for Haskell, presumably due to less efficient implementation of laziness and different optimizations. But we can still draw meaningful conclusions of the effects of laziness by comparing programs written in the same language with each other.

8 "Appropriate" here means we avoid Wadler et al.'s [80] "odd" errors.

immediately obvious where to insert them. Part iii of this dissertation presents a static-analysis-based refactoring tool that assists programmers with laziness by transforming the program to add additional required `delay`s and `force`s. In this particular case, the static analysis would suggest an additional `delay` in the `foldr` function:

```
let rec foldr f base lst =
  match force lst with
  | [] -> base
  | x::lzxs -> f x (delay (foldr f base xs))
```

This unobvious `delay` is needed because `f`'s second argument eventually flows to a lazy `cons` in append (`@`). Without this `delay`, the list of all queen placements is evaluated prematurely. With this refactoring, and an appropriate insertion of `force`s, the lazy-strict programmer sees a dramatic improvement over the original timings:

```
avg cpu time: 3328ms  avg real time: 3334ms  avg gc time: 1104ms
```

1.4.2 *More Improvement*

Frustratingly, the n-queens example from the previous section (1.4.1) is still too lazy. The example actually requires laziness only in *foldr* and *filter*. Figure 3 summarizes the running times for the 8-queens program implemented in Racket [29] and varying degrees of laziness. The figure reports that the strict version is slow and that adding the laziness suggested in the section 1.4.1 produces roughly a four-fold speedup. However, adding only the laziness annotations in *foldr* and *filter* produces a program that is roughly *fifteen times faster* than the strict version. The overapproximation of the statically-computed laziness was due to a "seed" laziness requirement. In order for the static tool to compute its suggestions, eager lists are first converted to lazy lists, as strict programmers looking to add laziness commonly do. It turns out, however, that this inserts *too much laziness*. *Only filter* needs additional laziness; the other lists should be evaluated eagerly.

We see that even strict programmers can insert too much laziness, illustrating the difficulty of determining exactly which parts of a program should be lazy. This raises the question of when exactly an expression in a strict program should be lazy. Part v of this dissertation presents a dynamic metric, called *laziness potential*, that predicts exactly what parts of a program benefit from laziness. We then present a profiler that computes laziness potential

Implementation Description	Time (ms)
no laziness	11676
static tool-based suggestions	3328
dynamic profiler-based suggestions	784

Figure 3: Running times for an 8-queens program.

and presents laziness suggestions to a programmer. In this case, the profiler would suggest only the `delays` in *foldr* and *filter*, which is also the laziness arrangement we used in the improved Haskell examples from section 1.3.

1.5 THIS DISSERTATION AND ITS THESIS

To this day, deploying laziness in practice remains difficult, as illustrated by the described examples. Worse, there has been little mitigation of the high level of expertise required to use laziness, with both strict-first and lazy-first programmers resorting to manual annotations as the current "best practice". This dissertation's goal is to lower the barrier of deploying laziness in real-world programs via automatic and semi-automatic tools. We demonstrate that *designing lazy algorithms and data structures in strict languages benefits from static and dynamic tool support*.

1.6 OUTLINE OF THE DISSERTATION

- Part ii of this dissertation briefly explores the history of laziness and relates previous attempts at taming it to our own.
- Part iii presents a static analysis that approximates the flow of laziness through a program. The analysis automatically computes, based on some initial so-called "seed" laziness and a control-flow analysis, what other parts of a program should be lazy. The requirement for some initial "seed" annotations, however, contributes imprecision. Additionally, because the tool considers only static information, it merely approximates where laziness is needed and often produces spurious suggestions.
- Part iv explores a criteria for when exactly an expression in a program should be lazy, via a series of examples.

- Part v Formalizes the criteria from part iv and also presents a dynamic profiler that computes a special metric, *laziness potential*, and presents suggestions to the programmer.
- Finally, part vi concludes with some ideas for future work.

Part II

PREVIOUS WORK

2 RELATED WORK

2.1 A SHORT HISTORY OF LAZINESS

In the early 1970s, researchers proposed the call-by-need [30, 34, 81] parameter passing mechanism that purported to combine the best of both the call-by-value and call-by-name models. While call-by-value always evaluates the argument of a function, call-by-name evaluates the argument every time it is needed. Hence, if an argument (or some portion) is never needed, call-by-name wins; otherwise call-by-value is superior because it avoids re-evaluation of arguments. Call-by-need initially proceeds like call-by-name, evaluating a function's body before the argument—until the value of the argument is needed; at that point, the argument is evaluated and the resulting value is used from then onward. In short, call-by-need evaluates an argument at most once, and only if needed.

Since then, researchers have explored a number of elegant theoretical characterizations of call-by-need [4, 5, 6, 6, 14, 19, 21, 31, 32, 43, 44, 47, 52, 54, 58, 64, 67, 71].[1] However, most are too abstract to be of practical use for programmers trying to determine the proper balance of laziness and strictness in their programs, because the models do not reason about resource consumption, i.e. the costs of laziness. For example, the reductions in the call-by-need lambda calculus [6] do not address how to delay expressions. Only the lowest-level machine semantics [21, 31, 44, 64, 71] begin to portray the true resources required by lazy evaluation but even these models have limited utility since they do not help programmers to reason about laziness in terms of the surface syntax.

2.2 BALANCING LAZY AND STRICT

Programmers have learned that the key to deploying laziness is moderation, because laziness severely complicates reasoning about a program's resource consumption. Approaches to taming laziness come from two different directions. From one direction, language researchers have devised strategies for determining when to safely *remove* laziness [16, 20, 23, 51, 57]. Programmers also annotate their programs with strictness annotations such as Haskell's `seq` [63] and GHC's bang (`!`) patterns [33]. From the other direction, strict

1 For my own treatment of call-by-need as a lambda calculus, see the appendix A.

programmers typically *add* laziness via annotations such as `delay` and `force`, for example to manage large data structures [62], delay possibly unneeded tasks [27], or leverage other lazy design patterns [78].

2.2.1 *Subtracting Laziness*

The most well-known technique to remove laziness from lazy programs is strictness analysis [10, 16, 37, 39, 42, 57, 79, 84], which statically removes laziness by estimating which thunks would be forced during evaluation without introducing non-termination.

Others have introduced additional static techniques that build on strictness analysis to remove even more laziness. Faxén [23] uses a whole-program analysis that utilizes labeled types and type inference [22] to identify "cheap eagerness." The analysis maps labeled expressions to "costs" and expressions with low enough cost are evaluated strictly. Boquist [9] uses an analysis to identify promises that can be removed in case expressions.

Researchers have also explored the idea of combining strict and lazy evaluation in a dynamic setting. In "stingy" evaluation [77] expressions are evaluated strictly, but only for a short period of time, thus yielding only a small occasional improvement. Maessen [51] and Ennals and Peyton Jones [20] additionally introduced similar strict-by-default mechanisms. In both their systems, programs are evaluated eagerly but various runtime heuristics determine when to suspend the computation to preserve lazy semantics.

Id [60] and pH [1], are two languages that try to exploit parallelism in programs using a technique called lenient evaluation, which is a blend of lazy and strict evaluation. To leniently evaluate a function call, the arguments and function body are eagerly evaluated at the same time. When a computation needs the value from another computation, it blocks if the other value is not ready. If some computation, a function argument for example, is not used, then the program never waits for its completion, so it does not affect evaluation of the final answer, even if that computation never terminates. Therefore, though function calls are evaluated eagerly, lenient evaluation ultimately implements a non-strict semantics.

The previous techniques experienced varying degrees of success. Faxén's cheap eagerness seems to gain the most improvement with regard to fixing space leaks during tail-recursive iterations. The dynamic strategies [20, 51] successfully meet their goal of efficient iteration while still handling infinite data structures (i.e., non-termination) with a reasonable amount of overhead. But these strategies sometimes remove too much laziness, for example where laziness is beneficial in finite programs [62]. Strictness analysis is the sole technique that sees practical use today, though it too struggles to handle tail-recursive iteration and leaves too much laziness in the program.

Overall, it seems that the prior strategies have not quite found an ideal balance of lazy and strict, as real-world lazy programmers still resort to manually annotating their programs with strictness annotations such as `seq` [63] and bang (`!`).

2.2.2 *Adding Laziness*

Scheme [68] and ML [2] have supported manual stream programming with `delay` and `force` for decades. Using `delay` and macros, a programmer can easily turn an eager, Lisp-style list `constructor` into a lazy one [30], while `force` retrieves the value from a delayed computation.

Wadler et al. [80] present a technique to automatically insert laziness annotations into a strict program. However, they assume that the programmer wants laziness everywhere and thus the result of their refactoring suffers from the same excessive resource consumption as lazy-by-default languages.

While the primary contribution of the Wadler et al. [80] paper does not apply to our work, there is one aspect of the paper that is relevant. The paper points out an "off by one" error that can occur when lazy `cons` is used with a certain pattern of `force`s in a strict language. They dub this erroneous usage the "odd" style. For example, here is the "odd" `map` definition from their paper:

```
define map(f lst) =
  if null? lst
     null
     lcons (f (first lst))
           (map f (force (rest lst)))
```

Assume that the `lcons` form delays only its tail argument. This definition forces the tail thunk before the recursive call to `map`. However, promises should not be forced until needed, as defined by the language's strictness points, so one should not be surprised that the premature forcing of promises here can result in more evaluation than necessary.

Ideally, the forcing of a promise should be postponed until it is known where its value is needed. This is the only pattern of `force`s used in this dissertation, a pattern dubbed the "even" style by Wadler et al. and thus programs refactored via the techniques in this dissertation cannot suffer from any odd-style errors. Here is `map` in the "even" style:

```
define map(f lst) =
  if null? (force lst)
```

```
null
lcons (f (first (force lst)))
      (map f (rest (force lst)))
```

In this definition, the (potential) promise is not forced unless its value is required, indicated by the underlines.[2] We note a limitation of our techniques in that given a program *already* written in odd-style, our tools do not suggest a switch to the even-style. However, any laziness inserted by our tools has accompanying `force`s in the "even" style. Enhancing our tool with erroneous "odd" style detection would require a flow-sensitive analysis and is interesting future work.

Sheard [72] shares the vision of a strict language that is also practical for programming lazily. While his language does not require explicit `force`s, the language provides no guidance when it comes to the non-trivial task of inserting laziness annotations at the proper places. In addition, the language does not calculate where forces are required and instead just inserts forces at every possible strictness point. Naturally his method of `force` insertion produces the correct results, but it inevitably inserts too many `force`s. Our static analysis computes where to insert `force`s only when needed.

2.2.3 *Laziness By Need: This Dissertation's Contribution*

This dissertation tackles the laziness problem from the relatively-unexplored, strict-evaluation-first end of the spectrum. We are not aware of any work that looks to automatically insert laziness into strict programs. Its goal is to insert laziness in a balanced way such that the program benefits, yet the costs are minimized. In other words, we look to inject laziness on a by-need basis only.

Starting with a strict language is advantageous in several ways. Many disadvantages of lazy evaluation such as excessive space and time resource consumption, which researchers have battled with many sophisticated techniques, naturally do not pose a problem in strict languages since we effectively start with "zero" cost. In addition, as many researchers have observed that promises in lazy languages are not needed [20, 23, 51, 55, 69], it makes more sense to start without laziness and then to compute where to add a handful of `delay` annotations, rather than to start with everything lazy and then try to remove the bulk of the added laziness.

2 Wadler et al.'s solution differs slightly due to their use of a typed language

2.3 VALUE USE

Merely switching from eager constructors to lazy ones in a strict program is often not enough to achieve the performance benefits of laziness because the insertion of one `delay` tends to require additional `delay`s elsewhere in the program to achieve the desired lazy behavior. Since these additional `delay` insertions depend on the value flow of the program, it can be difficult to determine where to insert them, especially in the presence of higher-order functions.

Thus prior work concerning both static and dynamic variants of the notion of value "use" are relevant to our work. We present a static analysis that defines static "strict positions" when determining where to insert `force`s. We also present a dynamic profiler that relies on "usage contexts" when computing laziness potential, where usage contexts are contexts that require a specific (kind of) value in order to proceed with evaluating the rest of the program. These concepts are related to, and derived from, several pieces of previous works.

2.3.1 *Static Use Analysis*

Compiler writers routinely use static analyses to predict the use of variables and the values they represent. If a program does not use a variable from a certain point on, the compiler might discard the computation of its value. Shivers's [74] useless-variable elimination, and Wand's subsequent extension [82], employ a higher-order use analyses similar to ours. The primary difference is that we are concerned with the usage of thunks, and thus our technique can be considered an extension of variable use. For example, we must also consider thunks originating from data constructors while the original work focuses only on variables (i.e., function arguments).

2.3.2 *Strictness Analysis*

From the dynamic side, a concept related to "use" is the denotational notion of strictness [16, 57] mentioned earlier in this chapter. Strictness is defined via the value $\bot$: a function `f` is strict if `f` $\bot = \bot$. In a lazy language, a strict function's argument does not need to be delayed. In the strict world, however, there is no analogous $\bot$-like value that extensionally indicates when to delay an expression. We must instead consider intensional value flows, which is precisely what our notion of usage contexts capture.

2.3.3 *Low-Utility Data Structures*

Xu et al. [22] present a system related to our dynamic profiler that assigns costs to bytecode instructions in order to identify unimportant parts of a program. On the surface, Xu et al.'s system differs from ours because the former analyzes the utility of objects in imperative, object-oriented programs, while we deal with the injection of laziness into functional programs. The primary difference, however, concerns the cost metrics. Xu et al. measure cost at the primitive level of memory accesses and bytecode instructions. In contrast, we formulate the cost of a wide variety of high-level constructs at source level. Xu et al. assume that average programmers can interpret the results of their low-level measurements back to high-level language constructs but do not explain how, and we do not accept this to be trivial. We leave the precise relation between the two cost models to future work.

Part III

A STATIC APPROACH

ML, Scheme, and other strict functional languages have supported lazy stream programming with `delay` and `force` for several decades. Today, nearly every modern programming language provides some facility to leverage laziness in programs. Unfortunately, these languages come with little to no guidance on how to design for laziness.

To remedy this situation, we present a semantics-based refactoring that helps strict programmers manage manual lazy programming. The refactoring uses a static analysis to identify where additional `delay`s and `force`s might be needed to achieve the desired simplification and performance benefits, once the programmer has added the initial lazy data constructors. We also present a correctness argument for the underlying transformations and some preliminary experiences with a prototype tool implementation.

Chapter 3 presents the analysis-based program transformation, and chapter 4 argues its correctness. Chapter 5 sketches a prototype implementation, and chapter 6 describes real-world applications.[3]

3 The material for part iii of this dissertation appeared previously in Chang [2013]: *Laziness By Need*, In: ESOP '13: Proceedings of the 22nd European Symposium on Programming.

3

REFACTORING FOR LAZINESS

The heart of our refactoring is a whole-program analysis that calculates where values may flow. Our transformation uses the results of the analysis to insert `delay`s and `force`s. Section 3.1 describes the core of our strict language. We then present our analysis in three steps: section 3.2 explains the analysis rules for our core language; section 3.3 extends the language and analysis with lazy forms: `delay`, `force`, and lazy `cons` (`lcons`); and section 3.4 extends the analysis again to calculate the potential insertion points for `delay` and `force`. Finally, section 3.5 defines the refactoring transformation function.

3.1 LANGUAGE SYNTAX

Our starting point is an untyped[1] functional core language. The language is *strict* and uses a standard expression notation:

$$\begin{aligned} e \in \mathit{Exp} = {} & n \mid b \mid x \mid \lambda(x \ldots).e \mid e\ e \ldots \mid o\ e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\ & \mid \texttt{zero?}\ e \mid \texttt{not}\ e \mid \texttt{if}\ e\ e\ e \\ & \mid \texttt{null} \mid \texttt{cons}\ e\ e \mid \texttt{first}\ e \mid \texttt{rest}\ e \mid \texttt{null?}\ e \end{aligned}$$

$$n \in \mathbb{Z},\quad b \in \mathit{Bool} = \texttt{true} \mid \texttt{false},\quad x \in \mathit{Var},$$
$$o \in \mathit{Op} = + \mid - \mid * \mid / \mid < \mid > \mid = \mid \texttt{or} \mid \texttt{and}$$

There are integers, booleans, variables, λs, applications, boolean and arithmetic primitives, conditionals, (non-recursive) lets, and eager lists and list operations. Here are the values, where both components of a non-empty list must be values:

$$v \in \mathit{Val} = n \mid b \mid \lambda(x \ldots).e \mid \texttt{null} \mid \texttt{cons}\ v\ v$$

A program p consists of two pieces: a series of mutually referential function definitions and an expression that may call the functions:

$$p \in \mathit{Prog} = d \ldots e \qquad\qquad d \in \mathit{Def} = \texttt{define}\ f(x \ldots) = e$$

1 Standard type systems cannot adequately express the flow of laziness and thus cannot solve the `delay`-insertion problems from Chapter 1. A type error can signal a missing `force`, but a type system will not suggest where to add performance-related `delay`s. Thus for convenience we omit types from our analysis language.

3.2 A STATIC ANALYSIS, STEP 1: OCFA

Our initial analysis is based on oCFA [41, 70, 73]. The analysis assumes that each subexpression has a unique label ℓ, also drawn from *Var*, but that the set of labels and the set of variables in a program are disjoint. The analysis computes an abstract environment $\widehat{\rho}$ that maps elements of *Var* to sets of abstract values:

$$\widehat{\rho} \in \mathit{Env} = \mathit{Var} \to \mathcal{P}(\widehat{v})$$
$$\ell \in \mathit{Var}$$
$$\widehat{v} \in \widehat{\mathit{Val}} = \texttt{val} \mid \lambda(x \ldots).\ell \mid \texttt{cons}\ \ell\ \ell$$

A set $\widehat{\rho}(x)$ or $\widehat{\rho}(\ell)$ represents an approximation of all possible values that can be bound to x or observed at ℓ, respectively, during evaluation of the program.

The analysis uses abstract representations of values, $\widehat{v}$, where `val` stands for all literals in the language. In addition, $\lambda(x \ldots).\ell$ are abstract function values where the body is represented with a label, and $(\texttt{cons}\ \ell\ \ell)$ are abstract list values where the ℓ's are the labels of the respective pieces. We also overload the $\widehat{\cdot}$ notation to denote a function that converts a concrete value to its abstract counterpart:

$$\boxed{\widehat{\cdot} : \mathit{Val} \to \widehat{\mathit{Val}}}$$

$$\widehat{n} = \texttt{val} \qquad \widehat{b} = \texttt{val} \qquad \widehat{\texttt{null}} = \texttt{val}$$
$$\widehat{\lambda(x \ldots).e^{\ell}} = \lambda(x \ldots).\ell \qquad \widehat{\texttt{cons}\ v_1^{\ell_1}\ v_2^{\ell_2}} = \texttt{cons}\ \ell_1\ \ell_2$$

We present our analysis with a standard [59], constraints-based specification, where notation $\widehat{\rho} \models p$ means $\widehat{\rho}$ is an acceptable approximation of program p. Figures 4 and 5 show the analysis for programs and expressions, respectively.

$$\widehat{\rho} \models d \ldots e \qquad [\mathit{prog}]$$
$$\quad \text{iff} \quad \widehat{\rho} \models_d d \ \wedge\ \ldots\ \wedge\ \widehat{\rho} \models_e e$$
$$\widehat{\rho} \models_d \texttt{define}\ f(x \ldots) = e^{\ell} \qquad [\mathit{def}]$$
$$\quad \text{iff} \quad \lambda(x \ldots).\ell \in \widehat{\rho}(f) \ \wedge\ \widehat{\rho} \models_e e^{\ell}$$

Figure 4: oCFA analysis on programs

The [*prog*] rule specifies that environment $\widehat{\rho}$ satisfies program $p = d \ldots e$ if it satisfies all definitions $d \ldots$ as well as the expression e in the program. The

$$\widehat{\rho} \models_e n^{\ell} \quad \text{iff} \quad \texttt{val} \in \widehat{\rho}(\ell) \qquad [num]$$

$$\widehat{\rho} \models_e b^{\ell} \quad \text{iff} \quad \texttt{val} \in \widehat{\rho}(\ell) \qquad [bool]$$

$$\widehat{\rho} \models_e x^{\ell} \quad \text{iff} \quad \widehat{\rho}(x) \subseteq \widehat{\rho}(\ell) \qquad [var]$$

$$\widehat{\rho} \models_e (\lambda(x \ldots).e_0^{\ell_0})^{\ell} \qquad [lam]$$

$$\text{iff} \quad \lambda(x \ldots).\ell_0 \in \widehat{\rho}(\ell) \ \wedge \ \widehat{\rho} \models_e e_0^{\ell_0}$$

$$\widehat{\rho} \models_e (e_f^{\ell_f}\ e_1^{\ell_1} \ldots)^{\ell} \qquad [app]$$

$$\begin{aligned}\text{iff} \quad & \widehat{\rho} \models_e e_f^{\ell_f} \ \wedge \ \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \ldots \ \wedge \\ & (\forall \lambda(x_1 \ldots).\ell_0 \in \widehat{\rho}(\ell_f) : \\ & \quad \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1) \ \wedge \ \ldots \ \wedge \\ & \quad \widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell))\end{aligned}$$

$$\widehat{\rho} \models_e (\texttt{let}\ x = e_1^{\ell_1}\ \texttt{in}\ e_0^{\ell_0})^{\ell} \qquad [let]$$

$$\begin{aligned}\text{iff} \quad & \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x) \ \wedge \\ & \widehat{\rho} \models_e e_0^{\ell_0} \ \wedge \ \widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell)\end{aligned}$$

$$\widehat{\rho} \models_e (o\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \qquad [op]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho} \models_e e_2^{\ell_2} \ \wedge \ \texttt{val} \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{zero?}\ e_1^{\ell_1})^{\ell} \qquad [zero?]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \texttt{val} \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{not}\ e_1^{\ell_1})^{\ell} \qquad [not]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \texttt{val} \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{if}\ e_1^{\ell_1}\ e_2^{\ell_2}\ e_3^{\ell_3})^{\ell} \qquad [if]$$

$$\begin{aligned}\text{iff} \quad & \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho} \models_e e_2^{\ell_2} \ \wedge \ \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell) \ \wedge \\ & \qquad\qquad \widehat{\rho} \models_e e_3^{\ell_3} \ \wedge \ \widehat{\rho}(\ell_3) \subseteq \widehat{\rho}(\ell)\end{aligned}$$

$$\widehat{\rho} \models_e \texttt{null}^{\ell} \quad \text{iff} \quad \texttt{val} \in \widehat{\rho}(\ell) \qquad [null]$$

$$\widehat{\rho} \models_e (\texttt{null?}\ e_1^{\ell_1})^{\ell} \qquad [null?]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \texttt{val} \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{cons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \qquad [cons]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho} \models_e e_2^{\ell_2} \ \wedge \ (\texttt{cons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{first}\ e_1^{\ell_1})^{\ell} \qquad [first]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ (\forall(\texttt{cons}\ \ell_2\ \_) \in \widehat{\rho}(\ell_1) : \ \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$

$$\widehat{\rho} \models_e (\texttt{rest}\ e_1^{\ell_1})^{\ell} \qquad [rest]$$

$$\text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ (\forall(\texttt{cons}\ \_\ \ell_2) \in \widehat{\rho}(\ell_1) : \ \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$

Figure 5: Step 1: oCFA analysis on expressions

[*def*] rule says that $\widehat{\rho}$ satisfies a definition if the corresponding abstract λ-value is included for variable `f` in $\widehat{\rho}$, and if $\widehat{\rho}$ satisfies the function body as well.

In figure 5,[2] the [*num*], [*bool*], and [*null*] rules show that `val` represents these literals in the analysis. The [*var*] rule connects variables x and their labels ℓ, specifying that all values bound to x should also be observable at ℓ. The [*lam*] rule for an ℓ-labeled λ says that its abstract version must be in $\widehat{\rho}(\ell)$ and that $\widehat{\rho}$ must satisfy its body. The [*app*] rule says that $\widehat{\rho}$ must satisfy the function and arguments in an application. In addition, for each possible λ in the function position, the arguments must be bound to the corresponding parameters of that λ and the result of evaluating the λ's body must also be a result for the application itself. The [*let*] rule has similar constraints. The [*op*], [*zero?*], [*not*], and [*null?*] rules require that $\widehat{\rho}$ satisfy a primitive's operands and uses `val` as the result. The [*if*] rule requires that $\widehat{\rho}$ satisfy the test expression and the two branches, and that any resulting values in the branches also be a result for the entire expression. The [*cons*] rule for an ℓ-labeled, eager `cons` requires that $\widehat{\rho}$ satisfy both arguments and that a corresponding abstract `cons` value be in $\widehat{\rho}(\ell)$. Finally, the [*first*] and [*rest*] rules require satisfiability of their arguments and that the appropriate piece of any `cons` arguments be a result of the entire expression.

3.3 A STATIC ANALYSIS, STEP 2: ADDING `delay` AND `force`

Next we extend our language and analysis with lazy forms:

$$e \in \mathit{Exp} = \ldots \mid \texttt{delay}\ e \mid \texttt{force}\ e \mid \texttt{lcons}\ e\ e$$

$$\text{where } \texttt{lcons}\ e_1\ e_2 \stackrel{\text{df}}{\equiv} \texttt{cons}\ e_1\ (\texttt{delay}\ e_2)$$

The language is still strict but `delay` introduces promises. A `force` term recursively forces all nested `delay`s. Lazy `cons` (`lcons`) is only lazy in its rest argument and `first` and `rest` work with both `lcons` and `cons` values so that `rest` (`lcons` v e) results in (`delay` e).

We add promises and lazy lists to the sets of values and abstract values:

$$v \in \mathit{Val} = \ldots \mid \texttt{delay}\ e \mid \texttt{lcons}\ v\ e$$

$$\widehat{v} \in \widehat{\mathit{Val}} = \ldots \mid \texttt{delay}\ \ell \mid \texttt{lcons}\ \ell\ \ell$$

2 Some rules in figure 5 differ slightly from the presentation in Nielson et al. [59] in that the former analyzes all function bodies while the latter analyzes only applied function bodies. We deviate because we wish to be able to analyze libraries of (possibly) unapplied functions.

$$\widehat{\rho} \models_e (\texttt{delay}\ e_1^{\ell_1})^{\ell} \qquad [\textit{delay}]$$
$$\quad \text{iff} \quad (\texttt{delay}\ \ell_1) \in \widehat{\rho}(\ell) \ \wedge\ \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(\ell)$$
$$\widehat{\rho} \models_e (\texttt{force}\ e_1^{\ell_1})^{\ell} \qquad [\textit{force}]$$
$$\quad \text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge\ (\forall \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay} : \widehat{v} \in \widehat{\rho}(\ell))$$
$$\widehat{\rho} \models_e (\texttt{lcons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \qquad [\textit{lcons}]$$
$$\quad \text{iff} \quad \widehat{\rho} \models_e e_1^{\ell_1} \ \wedge\ \widehat{\rho} \models_e e_2^{\ell_2} \ \wedge\ (\texttt{lcons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell)$$
$$\widehat{\rho} \models_e (\texttt{first}\ e_1^{\ell_1})^{\ell} \qquad [\textit{first}]$$
$$\quad \text{iff} \quad \ldots \ \wedge\ (\forall(\texttt{lcons}\ \ell_2\ \_) \in \widehat{\rho}(\ell_1) : \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$
$$\widehat{\rho} \models_e (\texttt{rest}\ e_1^{\ell_1})^{\ell} \qquad [\textit{rest}]$$
$$\quad \text{iff} \quad \ldots \ \wedge\ (\forall(\texttt{lcons}\ \_\ \ell_2) \in \widehat{\rho}(\ell_1) :$$
$$(\texttt{delay}\ \ell_2) \in \widehat{\rho}(\ell) \ \wedge\ \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$

Figure 6: Step 2: Analysis with lazy forms.

The $\widehat{\cdot}$ function is similarly extended:

$$\boxed{\widehat{\cdot} : \mathit{Val} \to \widehat{\mathit{Val}}}$$

$$\ldots$$
$$\widehat{\texttt{delay}\ e^{\ell}} = \texttt{delay}\ \ell$$
$$\widehat{\texttt{lcons}\ v_1^{\ell_1}\ e_2^{\ell_2}} = \texttt{lcons}\ \ell_1\ \ell_2$$

The abstract representation of a `delay` replaces the labeled delayed expression with just the label and the abstract `lcons` is similar.

Figure 6 presents the new and extended analysis rules. The [*delay*] rule specifies that for an ℓ-labeled `delay`, the corresponding abstract `delay` must be in $\widehat{\rho}(\ell)$ and $\widehat{\rho}$ must satisfy the delayed subexpression. In addition, the values of the delayed subexpression must also be in $\widehat{\rho}(\ell)$. This means that the analysis approximates evaluation of a promise with both a promise and the result of forcing that promise. We discuss the rationale for this constraint below. The [*force*] rule says that $\widehat{\rho}$ must satisfy the argument and that non-`delay` arguments are propagated to the outer ℓ label. Since the [*delay*] rule already approximates evaluation of the delayed expression, the [*force*] rule does not have any such constraints.

We also add a rule for `lcons` and extend the [*first*] and [*rest*] rules to handle `lcons` values. The [*lcons*] rule requires that $\widehat{\rho}$ satisfy the arguments and requires a corresponding abstract `lcons` at the expressions's ℓ label. The [*first*] rule handles `lcons` values just like `cons` values. For the [*rest*] rule, a `delay`

with the `lcons`'s second component is a possible result of the expression. Just like the [*delay*] rule, the [*rest*] rule assumes that the lazy component of the `lcons` is both forced and unforced, and thus there is another constraint that propagates the values of the (undelayed) second component to the outer label.

Implicit Forcing.

In our analysis, `delays` are both evaluated and unevaluated. We assume that during evaluation, a programmer does not want an unforced `delay` to appear in a strict position. For example, if the analysis discovers an unforced `delay` as the function in an application, we assume that the programmer forgot a `force` and analyze that function call anyway. This makes our analysis quite conservative but minimizes the effect of any laziness-related errors in the computed control flow. On the technical side, implicit forcing also facilitates the proof of a safety theorem for the transformation (see subsection 4.3).

3.4 A STATIC ANALYSIS STEP 3: LAZINESS ANALYSIS

Our final refinement revises the analysis to calculate three additional sets, which are used to insert additional `delays` and `forces` in the program:

$$\widehat{\mathcal{D}} \in \mathit{DPos} = \mathcal{P}(\mathrm{Var})$$
$$\widehat{\mathcal{S}} \in \mathit{SPos} = \mathcal{P}(\mathrm{Var})$$
$$\widehat{\mathcal{F}} \in \mathit{FPos} = \mathcal{P}(\mathrm{Var} \cup (\mathrm{Var} \times \mathrm{Var}))$$

Intuitively, $\widehat{\mathcal{D}}$ is a set of labels representing function arguments that flow to lazy positions and $\widehat{\mathcal{S}}$ is a set of labels representing arguments that flow to strict positions. Our transformation then delays arguments that reach a lazy position but not a strict position. Additionally, $\widehat{\mathcal{F}}$ collects the labels where a delayed value may appear—both those manually inserted by the programmer and those suggested by the analysis—and is used by the transformation to insert `forces`.

We first describe how the analysis computes $\widehat{\mathcal{D}}$. The key is to track the flow of arguments from an application into a function body and for this, we introduce a special abstract value $(\texttt{arg}\ \ell)$, where ℓ labels an argument in a function call:

$$\widehat{v} \in \widehat{\mathit{Val}} = \ldots \mid \texttt{arg}\ \ell$$

Figure 7 presents revised analysis rules related to $\widehat{\mathcal{D}}$. To reduce clutter, we express the analysis result as $(\widehat{\rho}, \widehat{\mathcal{D}})$, temporarily omitting $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{F}}$. In the new [*app*] and [*let*] rules, additional constraints (box 1) specify that for each labeled argument, an `arg` abstract value with a matching label must be in $\widehat{\rho}$ for the corresponding parameter. We are interested in the flow of arguments

only within a function's body, so the result-propagating constraint filters out `arg` values (box 2).

Recall that $\widehat{\mathcal{D}}$ is to contain labels of arguments that reach lazy positions. Specifically, if an $(\texttt{arg}\ \ell)$ value flows to a `delay` or the second position of an `lcons`, then ℓ must be in $\widehat{\mathcal{D}}$ (box 3)[3]. If an ℓ-labeled argument reaches a lazy position, the transformation *may* decide to delay that argument, so the analysis must additionally track it for the purposes of inserting `force`s. To this end, we introduce another abstract value $(\texttt{darg}\ \ell)$,

$$\widehat{v} \in \widehat{Val} = \ldots \mid \texttt{darg}\ \ell$$

and insert it when needed (box 4). While $(\texttt{arg}\ \ell)$ can represent any argument, $(\texttt{darg}\ \ell)$ represents only arguments that reach a lazy position (i.e., $\ell \in \widehat{\mathcal{D}}$).

Figure 8 presents revised analysis rules involving $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{F}}$. These rules use the full analysis result $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}})$. Here, $\widehat{\mathcal{S}}$ represents arguments that reach a strict position so the new [*force*] rule dictates that if an $(\texttt{arg}\ \ell)$ is the argument of a `force`, then ℓ must be in $\widehat{\mathcal{S}}$ (box 5). However, a `force` is not the only expression that requires the value of a promise. There are several other contexts where a `delay` should not appear and the [*strict*] rule deals with these strict contexts S: the operator in an application, the operands in the primitive operations, and the test in an `if` expression. Expressions involving these strict positions have three additional constraints. The first specifies that if an $(\texttt{arg}\ \ell_1)$ appears in any of these positions, then ℓ_1 should also be in $\widehat{\mathcal{S}}$ (box 5). The second and third additional constraints show how $\widehat{\mathcal{F}}$ is computed. Recall that $\widehat{\mathcal{F}}$ determines where to insert `force`s in the program. The second [*strict*] constraint says that if any `delay` flows to a strict position ℓ, then ℓ is added to $\widehat{\mathcal{F}}$ (box 6). This indicates that a programmer-inserted `delay` has reached a strict position and should be forced. Finally, the third constraint dictates that if a $(\texttt{darg}\ \ell_2)$ value flows to a strict label ℓ, then a pair (ℓ, ℓ_2) is required to be in $\widehat{\mathcal{F}}$ (box 7), indicating that the analysis *may* insert a `delay` at ℓ_2, thus requiring a `force` at ℓ.

3.5 THE REFACTORING TRANSFORMATION

Figure 9 specifies our refactoring as a function φ that transforms a program p using analysis result $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}})$. The φ_e function wraps expression e^ℓ with $\texttt{delay}^*$ if ℓ is in $\widehat{\mathcal{D}}$ and not in $\widehat{\mathcal{S}}$. In other words, e is delayed if it flows to a lazy position but not a strict position. With the following correctness section in mind, we extend the set of expressions with $\texttt{delay}^*$, which is exactly like `delay` and merely distinguishes programmer-inserted `delay`s from those inserted by the our transformation. The new $\texttt{delay}^*$ expression is given a

3 The function $fv(e)$ calculates free variables in e

$$
\begin{array}{l}
(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (e_f^{\ell_f}\ e_1^{\ell_1}\ \ldots)^{\ell} \qquad [\textit{app}] \\
\quad \text{iff} \quad (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_f^{\ell_f} \ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1} \ \wedge\ \ldots\ \wedge \\
\qquad (\forall \lambda(x_1\ \ldots).\ell_0 \in \widehat{\rho}(\ell_f) : \\
\qquad\quad \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1) \ \wedge\ \ldots\ \wedge \\
\qquad\quad \boxed{(\texttt{arg}\ \ell_1) \in \widehat{\rho}(x_1) \ \wedge\ \ldots}_1 \ \wedge \\
\qquad\quad \boxed{(\forall \widehat{v} \in \widehat{\rho}(\ell_0), \widehat{v} \notin \texttt{arg} : \widehat{v} \in \widehat{\rho}(\ell))}_2) \\
(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{let}\ x = e_1^{\ell_1}\ \texttt{in}\ e_0^{\ell_0})^{\ell} \qquad [\textit{let}] \\
\quad \text{iff} \quad (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1} \ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x) \ \wedge \\
\qquad \boxed{(\texttt{arg}\ \ell_1) \in \widehat{\rho}(x)}_1 \ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_0^{\ell_0} \ \wedge \\
\qquad \boxed{(\forall \widehat{v} \in \widehat{\rho}(\ell_0), \widehat{v} \notin \texttt{arg} : \widehat{v} \in \widehat{\rho}(\ell))}_2 \\
(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{delay}\ e_1^{\ell_1})^{\ell} \qquad [\textit{delay}] \\
\quad \text{iff} \quad (\texttt{delay}\ \ell_1) \in \widehat{\rho}(\ell) \ \wedge \\
\qquad (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1} \ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(\ell) \ \wedge \\
\qquad (\forall x \in \mathrm{fv}(e_1) : (\forall(\texttt{arg}\ \ell_2) \in \widehat{\rho}(x) : \\
\qquad\quad \boxed{\ell_2 \in \widehat{\mathcal{D}}}_3 \wedge \boxed{(\texttt{darg}\ \ell_2) \in \widehat{\rho}(x)}_4)) \\
(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{lcons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \qquad [\textit{lcons}] \\
\quad \text{iff} \quad (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1} \ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_2^{\ell_2} \ \wedge \\
\qquad (\texttt{lcons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell) \ \wedge \\
\qquad (\forall x \in \mathrm{fv}(e_2) : (\forall(\texttt{arg}\ \ell_3) \in \widehat{\rho}(x) : \\
\qquad\quad \boxed{\ell_3 \in \widehat{\mathcal{D}}}_3 \wedge \boxed{(\texttt{darg}\ \ell_3) \in \widehat{\rho}(x)}_4))
\end{array}
$$

Figure 7: Step 3a: Calculating flow to lazy positions.

$$(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e (\texttt{force}\ e_1^{\ell_1})^{\ell} \qquad [\textit{force}]$$

$$\text{iff} \quad (\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e e_1^{\ell_1} \ \wedge$$

$$(\forall \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay} : \widehat{v} \in \widehat{\rho}(\ell)) \ \wedge$$

$$\boxed{(\forall(\texttt{arg}\ \ell_2) \in \widehat{\rho}(\ell_1) : \ell_2 \in \widehat{\mathcal{S}})}_5$$

$$(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e S[e^{\ell}] \qquad [\textit{strict}]$$

$$\text{iff} \quad \ldots \ \wedge \ \boxed{(\forall(\texttt{arg}\ \ell_1) \in \widehat{\rho}(\ell) : \ell_1 \in \widehat{\mathcal{S}})}_5 \ \wedge$$

$$\boxed{(\exists \texttt{delay} \in \widehat{\rho}(\ell) \Rightarrow \ell \in \widehat{\mathcal{F}})}_6 \ \wedge$$

$$\boxed{(\forall(\texttt{darg}\ \ell_2) \in \widehat{\rho}(\ell) : (\ell,\ell_2) \in \widehat{\mathcal{F}})}_7$$

---

where $S \in \mathit{SCtx} = [\,]\ e \ldots \mid o\ [\,]\ e \mid o\ v\ [\,] \mid \texttt{if}\ [\,]\ e_1\ e_2$
$\mid \texttt{zero?}\ [\,] \mid \texttt{not}\ [\,] \mid \texttt{null?}\ [\,] \mid \texttt{first}\ [\,] \mid \texttt{rest}\ [\,]$

Figure 8: Step 3b: Calculating flow to strict positions.

fresh label ℓ_1. In two cases, φ_e inserts a `force` around an expression e^{ℓ}. First, if ℓ is in $\widehat{\mathcal{F}}$, it means ℓ is a strict position and a programmer-inserted `delay` reaches this strict position and must be forced. Second, an expression e^{ℓ} is also wrapped with `force` if there is some ℓ_2 such that (ℓ,ℓ_2) is in $\widehat{\mathcal{F}}$ and the analysis says to delay the expression at ℓ_2, i.e., $\ell_2 \in \widehat{\mathcal{D}}$ and $\ell_2 \notin \widehat{\mathcal{S}}$. This ensures that transformation-inserted `delay`*s are also properly forced. All remaining clauses in the definition of φ_e, represented with ellipses, traverse the structure of e in a homomorphic manner.

$$\boxed{\varphi : \mathit{Prog} \times \mathit{Env} \times \mathit{DPos} \times \mathit{SPos} \times \mathit{FPos} \to \mathit{Prog}}$$

$$\varphi[\![(\texttt{define}\ \mathsf{f}(x\ldots) = e_1)\ldots e]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = (\texttt{define}\ \mathsf{f}(x\ldots) = \varphi_e[\![e_1]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}})\ldots\varphi_e[\![e]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}$$

$$\boxed{\varphi_e : \mathit{Exp} \times \mathit{Env} \times \mathit{DPos} \times \mathit{SPos} \times \mathit{FPos} \to \mathit{Exp}}$$

$$\varphi_e[\![e^\ell]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = (\texttt{delay}^*\ (\varphi_e[\![e]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}})^\ell)^{\ell_1}, \qquad (\dagger)$$

$$\text{if } \ell \in \widehat{\mathcal{D}},\ \ell \notin \widehat{\mathcal{S}},\ \ell_1 \notin \mathrm{dom}(\widehat{\rho})$$

$$\varphi_e[\![e^\ell]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = (\texttt{force}\ (\varphi_e[\![e]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}})^\ell)^{\ell_1}, \qquad (\ddagger)$$

$$\text{if } \ell \in \widehat{\mathcal{F}},\ \ell_1 \notin \mathrm{dom}(\widehat{\rho}),$$

$$\quad \text{or } \exists \ell_2.(\ell, \ell_2) \in \widehat{\mathcal{F}},\ \ell_2 \in \widehat{\mathcal{D}},\ \ell_2 \notin \widehat{\mathcal{S}},\ \ell_1 \notin \mathrm{dom}(\widehat{\rho})$$

$$\ldots$$

Figure 9: Transformation function φ.

4

STATIC ANALYSIS CORRECTNESS

Our refactoring for laziness is not semantics-preserving. For example, non-terminating programs may be transformed into terminating ones or exceptions may be delayed indefinitely. Nevertheless, we can prove our analysis sound and the φ transformation safe, meaning that unforced promises cannot cause exceptions.

4.1 LANGUAGE SEMANTICS

To establish soundness, we use Flanagan and Felleisen's [28] technique, which relies on a reduction semantics. The semantics is based on evaluation contexts, which are a subset of expressions with a hole in place of one subexpression:

$$\begin{aligned} E \in Ctx = {} & [\,] \mid v \ldots\ E\ e \ldots \mid o\ E\ e \mid o\ v\ E \mid \texttt{let}\ x = E\ \texttt{in}\ e \\ & \mid \texttt{if}\ E\ e\ e \mid \texttt{zero?}\ E \mid \texttt{not}\ E \mid \texttt{force}\ E \mid \texttt{null?}\ E \\ & \mid \texttt{cons}\ E\ e \mid \texttt{cons}\ v\ E \mid \texttt{lcons}\ E\ e \mid \texttt{first}\ E \mid \texttt{rest}\ E \end{aligned}$$

A reduction step $\longmapsto$ is defined as follows, where $\rightarrow$ is specified in figure 10:

$$E[e] \longmapsto E[e'] \quad \text{iff} \quad e \rightarrow e'$$

A conventional δ function evaluates primitives and is elided. We again assume that subexpressions are uniquely labeled but since labels do not affect evaluation, they are implicit in the reduction rules, though we do mention them explicitly in the theorems. Since our analysis does not distinguish memoizing promises from non-memoizing ones, neither does our semantics. To evaluate complete programs, we parameterize $\longmapsto$ over definitions $d \ldots$, and add a look-up rule:

$$E[f] \longmapsto_{d\ldots} E[\lambda(x \ldots).e], \qquad \text{if } (\texttt{define}\ f(x \ldots) = e) \in d \ldots$$

Thus, the result of evaluating a program $p = d \ldots e$ is the result of reducing e with $\longmapsto_{d\ldots}$. We often drop the $d \ldots$ subscript to reduce clutter.

Exceptions

Our $\rightarrow$ reduction thus far is partial, as is the (elided) δ function. If certain expressions show up in the hole of the evaluation context, e.g., `first null` or

$$
\begin{array}{rll}
(\lambda(x \ldots).e)\ v \ldots & \to e\{x := v, \ldots\} & \text{(ap)} \\
o\ v_1\ v_2 & \to \delta\ o\ v_1\ v_2 & \text{(op)} \\
\texttt{let}\ x = v\ \texttt{in}\ e & \to e\{x := v\} & \text{(let)} \\
\texttt{if false}\ e_1\ e_2 & \to e_2 & \text{(iff)} \\
\texttt{if}\ v\ e_1\ e_2 & \to e_1,\ v \neq \texttt{false},\ v \neq \texttt{delay}\ e & \text{(if)} \\
\texttt{zero?}\ 0 & \to \texttt{true} & \text{(zo)} \\
\texttt{zero?}\ v & \to \texttt{false},\ v \neq 0,\ v \neq \texttt{delay}\ e & \text{(z)} \\
\texttt{not false} & \to \texttt{true} & \text{(notf)} \\
\texttt{not}\ v & \to \texttt{false},\ v \neq \texttt{false},\ v \neq \texttt{delay}\ e & \text{(not)} \\
\texttt{null? null} & \to \texttt{true} & \text{(nuln)} \\
\texttt{null?}\ v & \to \texttt{false},\ v \neq \texttt{null},\ v \neq \texttt{delay}\ e & \text{(nul)} \\
\texttt{first}\ (\texttt{cons}\ v_1\ v_2) & \to v_1 & \text{(fstc)} \\
\texttt{first}\ (\texttt{lcons}\ v\ e) & \to v & \text{(fstlc)} \\
\texttt{rest}\ (\texttt{cons}\ v_1\ v_2) & \to v_2 & \text{(rstc)} \\
\texttt{rest}\ (\texttt{lcons}\ v\ e) & \to \texttt{delay}\ e & \text{(rstlc)} \\
\texttt{force}\ (\texttt{delay}\ e) & \to \texttt{force}\ e & \text{(ford)} \\
\texttt{force}\ v & \to v,\ v \neq \texttt{delay}\ e & \text{(forv)} \\
 & & \\
v\ v_1 \ldots & \to \texttt{exn},\ v \neq \lambda(x \ldots).e & \text{(apx)} \\
\texttt{first}\ v & \to \texttt{exn},\ v \notin \texttt{cons or lcons} & \text{(fstx)} \\
\texttt{rest}\ v & \to \texttt{exn},\ v \notin \texttt{cons or lcons} & \text{(rstx)} \\
S[\texttt{delay}\ e] & \to \texttt{dexn} & \text{(strictx)} \\
S[\texttt{delay}^*\ e] & \to \texttt{dexn}^* & \text{(strictx}^*\text{)}
\end{array}
$$

Figure 10: Call-by-value reduction semantics.

division by 0, we consider the evaluation stuck. To handle stuck expressions, we add an exception `exn` to our semantics. We assume that δ returns `exn` for invalid operands of primitives and we extend $\rightarrow$ with the exception-producing reductions in the bottom part of figure 10.

The (apx) rule says that application of non-λs results in an exception. The (fstx) and (rstx) rules state that reducing `first` or `rest` with anything but a non-empty list is an exception as well. The (strictx) and (strictx*) reductions specify that an exception occurs when an unforced promise appears in a context where the value of that promise is required. These contexts are exactly the strict contexts S from figure 8. We introduce `dexn` and `dexn`* to indicate when a `delay` or `delay`* causes an exception; otherwise these tokens behave just like `exn`. We also extend $\longmapsto$:

$$\mathsf{E}[\mathtt{exn}] \longmapsto \mathtt{exn}$$

A conventional well-definedness theorem summarizes the language's semantics.

Theorem 1 (Well-Definedness)**.** *A program* p *either reduces to a value* v*; starts an infinitely long chain of reductions; or reduces to* `exn`*.*

Proof. The reduction relation satisfies unique decomposition, which means that an expression is either a value, `exn`, or it can be uniquely partitioned into an evaluation context and a $\rightarrow$ redex or `exn`. Using a progress-and-preservation approach, we can then prove that this subject is preserved across all reductions. □

4.2 SOUNDNESS OF THE ANALYSIS

Before stating the soundness theorem, we first extend our analysis to exceptions:

$$(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models_e \mathtt{exn}^{\ell} \qquad \text{[exn]}$$

Lemma 1 states that $\rightarrow$ preserves the $\models_e$ relation, and corollary 1 states that $\longmapsto$ preserves $\models_e$. We use notation $\widehat{\rho} \models_e e$ when we are not interested in $\widehat{\mathcal{D}}$, $\widehat{\mathcal{S}}$, and $\widehat{\mathcal{F}}$, which are only used for transformation. This means $\widehat{\rho}$ satisfies only the constraints from sections 3.2 and 3.3.

Lemma 1 (Preservation: $\rightarrow$)**.** *If* $\widehat{\rho} \models_e e$ *and* $e \rightarrow e'$*, then* $\widehat{\rho} \models_e e'$*.*

Proof. By cases on $\rightarrow$, using the corresponding analysis rules. □

Corollary 1 (Preservation: $\longmapsto$)**.** *If* $\widehat{\rho} \models_e e$ *and* $e \longmapsto e'$*, then* $\widehat{\rho} \models_e e'$*.*

We now state our soundness theorem, where $\longmapsto\!\!\!\!\!\rightarrow$ is the reflexive-transitive closure of $\longmapsto$. The theorem says that if an expression in a program reduces to an ℓ-labeled value, then any acceptable analysis result $\widehat{\rho}$ correctly predicts that value.

Theorem 2 (Soundness). *For all* $\widehat{\rho} \models p$, $p = d \ldots e$, *if* $e \longmapsto\!\!\!\!\!\rightarrow_{d\ldots} E[v^{\ell}]$, $\widehat{v} \in \widehat{\rho}(\ell)$.

Proof. By induction on the length of $\longmapsto\!\!\!\!\!\rightarrow$, using Corollary 1. □

4.3 SAFETY OF REFACTORING

We show that refactoring for laziness cannot raise an exception due to a `delay` or `delay`* reaching a strict position. To start, we define the function ξ which derives a satisfactory abstract environment for a φ-transformed program:

$$\boxed{\xi : Env \times Prog \to Env}$$

$$\xi[\![\widehat{\rho}]\!]_p = \widehat{\rho}', \text{ where}$$

$$\forall \ell, x \in \mathrm{dom}(\widehat{\rho}): \tag{1}$$
$$\widehat{\rho}'(\ell) = \widehat{\rho}(\ell) \cup \{(\texttt{delay}^*\ \ell_1) \mid (\texttt{darg}\ \ell_1) \in \widehat{\rho}(\ell), (\texttt{delay}^*\ e_1^{\ell_1}) \in p\}$$
$$\widehat{\rho}'(x) = \widehat{\rho}(x) \cup \{(\texttt{delay}^*\ \ell_1) \mid (\texttt{darg}\ \ell_1) \in \widehat{\rho}(x), (\texttt{delay}^*\ e_1^{\ell_1}) \in p\}$$
$$\forall (\texttt{delay}^*\ e_1^{\ell_1})^{\ell} \in p, \ell \notin \mathrm{dom}(\widehat{\rho}): \tag{2}$$
$$\widehat{\rho}'(\ell) = \widehat{\rho}(\ell_1) \cup \{(\texttt{delay}^*\ \ell_1)\} \cup \{(\texttt{delay}^*\ \ell_2) \mid (\texttt{darg}\ \ell_2) \in \widehat{\rho}(\ell_1), (\texttt{delay}^*\ e_2^{\ell_2}) \in p\}$$
$$\forall (\texttt{force}\ e_1^{\ell_1})^{\ell} \in p, \ell \notin \mathrm{dom}(\widehat{\rho}): \tag{3}$$
$$\widehat{\rho}'(\ell) = \{\widehat{v} \mid \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay}\}$$

The ξ function takes environment $\widehat{\rho}$ and a program p and returns a new environment $\widehat{\rho}'$. Part 1 of the definition copies $\widehat{\rho}$ entries to $\widehat{\rho}'$, except `darg` values are replaced with `delay`*s when there is a corresponding `delay`* in p. Parts 2 and 3 add new $\widehat{\rho}'$ entries for `delay`*s and `force`s not accounted for in $\widehat{\rho}$. When the given p is a φ-transformed program, then the resulting $\widehat{\rho}'$ satisfies that program.

Lemma 2. *If* $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models p$, *then* $\xi[\![\widehat{\rho}]\!]_{\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}} \models \varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}$.

Proof. Let $\widehat{\rho}' = \xi[\![\widehat{\rho}]\!]_{\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}}$. Proceed by cases on the result of $\varphi_e[\![e^{\ell}]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}$, where e is a subexpression in p. We drop the subscript arguments of φ_e for conciseness.

- If $\varphi_e[\![e^\ell]\!] = (\texttt{delay}^*\ \varphi_e[\![e]\!]^\ell)^{\ell_1}$, by the [*delay*] analysis rule, three constraints must hold:
  1. $(\texttt{delay}^*\ \ell) \in \widehat{\rho}'(\ell_1)$
  2. $\widehat{\rho}' \models_e \varphi_e[\![e]\!]^\ell$
  3. $\widehat{\rho}'(\ell) \subseteq \widehat{\rho}'(\ell_1)$

  Constraints 1 and 3 hold by ξ, part (2). For constraint 2, we know from $(\dagger)$ in φ_e and the analysis rules that an environment satisfying $\varphi_e[\![e]\!]$ has $\texttt{delay}^*$ values that may not be in $\widehat{\rho}$. However, we also see that any inserted $\texttt{delay}^*$s in $\varphi_e[\![e]\!]$ is exactly tracked by corresponding `darg` values in $\widehat{\rho}$. Thus constraint 2 holds as well, due to part (1) of ξ.
- If $\varphi_e[\![e^\ell]\!] = (\texttt{force}\ \varphi_e[\![e]\!]^\ell)^{\ell_1}$, by the [*force*] analysis rule, two constraints must hold:
  1. $\widehat{\rho}' \models_e \varphi_e[\![e]\!]^\ell$
  2. $\forall \widehat{v} \in \widehat{\rho}'(\ell), \widehat{v} \notin \texttt{delay} : \widehat{v} \in \widehat{\rho}'(\ell_1)$

  Constraint 1 holds by the same reasoning as in the $\texttt{delay}^*$ case, using $(\ddagger)$ in φ_e, and constraint 2 holds by part (3) of ξ's definition.
- All the other cases follow similar reasoning.

□

Finally, theorem 3 states the safety property.[1] It says that evaluating a transformed program cannot generate an exception due to `delay`s or $\texttt{delay}^*$s.

Theorem 3 (Safety)**.**
For all p *and* $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models p$, *if* $\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = d \ldots e$, *then* $e \mapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{d\ldots}$ *dexn, and* $e \mapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{d\ldots}$ *dexn**.

Proof. Let $\widehat{\rho}' = \xi[\![\widehat{\rho}]\!]_{\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}}}$. By lemma 2, $\widehat{\rho}' \models p'$.

1. Since `dexn` can only be generated if a `delay` value appears in a strict position, it is sufficient to show:

   $$e \mapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{d\ldots} E[S[(\texttt{delay}\ e_1^{\ell_1})^\ell]]$$

   We prove the claim by contradiction.

   Suppose $e \longmapsto\!\!\!\!\!\!\rightarrow_{d\ldots} E[S[(\texttt{delay}\ e_1^{\ell_1})^\ell]]$. By soundness (theorem 2) applied to $\widehat{\rho}'$ and p', $(\texttt{delay}\ \ell_1) \in \widehat{\rho}'(\ell)$. Then from the definition of ξ, $(\texttt{delay}\ \ell_1) \in \widehat{\rho}(\ell)$. From the analysis rules in figure 8, since we are at a strict position, a `delay` in $\widehat{\rho}(\ell)$ implies $\ell \in \widehat{\mathcal{F}}$, so φ would have inserted a `force` around ℓ. However, $\texttt{force}\ [\,] \notin S$ so we have a contradiction.

1 We conjecture that a stronger safety theorem holds but we leave proving it for future work: *For all* $p = d \ldots e$, *if* $e \longmapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{d\ldots} v_1$, $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models p$, $\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = d' \ldots e'$, $e' \longmapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{d'\ldots} v_2$, *and* $v_1, v_2 \neq$ *delay*, *then* $v_1 = v2$.

2. Since $\texttt{dexn}^*$ can only be generated if a $\texttt{delay}^*$ value appears in a strict position, it is sufficient to show:

$$e \mapsto\!\!\!\!\!\!\rightarrow\!\!\!\!\!\!\rightarrow_{\mathrm{d}\ldots} \mathsf{E}[\mathsf{S}[(\texttt{delay}^*\ e_1^{\ell_1})^{\ell}]]$$

We prove the claim by contradiction.

Suppose $e \longmapsto\!\!\!\!\!\!\rightarrow_{\mathrm{d}\ldots} \mathsf{E}[\mathsf{S}[(\texttt{delay}^*\ e_1^{\ell_1})^{\ell}]]$. By soundness (theorem 2), $(\texttt{delay}^*\ \ell_1) \in \widehat{\rho}'(\ell)$. Then from the definition of ξ, $(\texttt{darg}\ \ell_1) \in \widehat{\rho}(\ell)$. From the analysis rules in figure 8, since we are at a strict position, $(\texttt{darg}\ \ell_1) \in \widehat{\rho}(\ell)$ implies $(\ell, \ell_1) \in \widehat{\mathcal{F}}$. From the definition of φ, the existence of $(\texttt{delay}^*\ e_1^{\ell_1})$ also implies that $e_1^{\ell_1}$ is an argument in a function call and that $\ell_1 \in \widehat{\mathcal{D}}$ and $\ell_1 \notin \widehat{\mathcal{S}}$. Finally, since $(\ell, \ell_1) \in \widehat{\mathcal{F}}$, $\ell_1 \in \widehat{\mathcal{D}}$, and $\ell_1 \notin \widehat{\mathcal{S}}$, φ would have inserted a `force` around ℓ. However, $\texttt{force}\ [\,] \notin \mathsf{S}$, so we have a contradiction.

□

4.4 IDEMPOTENCY

In contrast to conventional software engineering refactorings, our transformation is not idempotent. Indeed, it may be necessary to refactor a program multiple times to get the "right" amount of laziness. Consider the following example:

```
let x = ⟨long computation⟩
    y = ⟨short computation involving x⟩
in (delay y)
```

The long computation should be delayed but applying our transformation once delays only the short computation. To delay the long computation, a second transformation round is required. In practice, we have observed that one round of laziness refactoring suffices to handle the majority of cases. However, chapter 6 presents a real-world example requiring multiple transformations so our tool currently allows the programmer to decide how often to apply the refactoring.

5 STATIC TOOL IMPLEMENTATION

We have implemented refactoring for laziness for Racket [29],[1] in the form of a prototype plugin tool for the DrRacket IDE. It uses laziness analysis to automatically insert `delay` and `force` expressions as needed. It also displays graphical justification when desired.

5.1 CONSTRAINT SOLVING ALGORITHM

Computing our laziness analysis requires two stages: (1) the generation of a set of constraints from a program, and (2) the search for the least solution using a conventional worklist algorithm [59]. The graph nodes are the variables and labels in the program, plus one node each for $\widehat{\mathcal{D}}$, $\widehat{\mathcal{S}}$, and $\widehat{\mathcal{F}}$. Without loss of generality, we use only labels for the nodes and $\widehat{\rho}$ for the analysis result in our description of the algorithm. There exists an edge from node ℓ_1 to ℓ_2 if there is a constraint where $\widehat{\rho}(\ell_2)$ depends on $\widehat{\rho}(\ell_1)$; the edge is labeled with that constraint. Thus one can view a node ℓ as the endpoint for a series of data flow paths. To compute $\widehat{\rho}(\ell)$, it suffices to traverse all paths from the leaves to ℓ, accumulating values according to the constraints along the way.

The analysis result is incrementally computed in a breadth-first fashion by processing constraints according a worklist of nodes. Processing a constraint entails adding values to $\widehat{\rho}$ so the constraint is satisfied. The algorithm starts by processing all constraints where a node depends on a value, e.g., $\texttt{val} \in \widehat{\rho}(\ell)$; the nodes on the right-hand side of these constraints constitute the initial worklist. Nodes are then removed from the worklist, one at a time. When a node is removed, the constraints on the out-edges of that node are processed and a neighbor ℓ of the node is added to the worklist if $\widehat{\rho}(\ell)$ was updated while processing a constraint. A node may appear in the worklist more than once, but only a finite number of times, as shown by the following termination argument.

1 Racket's semantics differs slightly from the semantics in figure 10. In contexts that expect either false or non-false values (e.g., `if`), Racket treats `delay`s as non-false while the semantics in figure 10 results in a exception. Thus, the analysis may be unsound for Racket programs that depend on this behavior.

Termination and Complexity of Constraint Solving

Inspecting the constraints from section 3 reveals that an expression requires recursive calls only for subexpressions. Thus, a finite program generates a finite number of constraints. For a finite program with finitely many labels and variables, the set of possible abstract values is also finite. Thus, a node can appear in the worklist only a finite number of times, so algorithm must terminate.

We observe in the constraint-solving algorithm that, (1) a node ℓ is added to the worklist only if $\widehat{\rho}(\ell)$ is updated due to a node on which it depends being in the worklist, and (2) values are always added to $\widehat{\rho}$; they are never removed. For a program of size n, there are $O(n)$ nodes in the dependency graph. Each node can appear in the worklist $O(n)$ times, and a data flow path to reach that node could have $O(n)$ nodes, so it can take $O(n^2)$ node visits to compute the solution at a particular node. Multiplying by $O(n)$ total nodes, means the algorithm may have to visit $O(n^3)$ nodes to compute the solution for all nodes.

5.2 LAZINESS REFACTORING TOOL

Our prototype tool uses the result of the analysis and the φ function from section 3.5 to insert additional `delay`s and `force`s. In contrast to the mathematical version of φ, its implementation avoids inserting `delay`s and `force`s around values and does not insert duplicate `delay`s or `force`s.

We evaluated a number of examples with our tool including the n-queens problem from chapter 1. Figure 11 shows the program in Racket, including timing information and a graphical depiction of the answer. Despite the use of `lcons`,[2] the program takes as long as an eager version of the same program (not shown) to compute an answer. Figure 12 shows the program after our tool applies the laziness transformation. When the tool is activated, it: (1) computes an analysis result for the program, (2) uses the result to insert `delay`s and `force`s, highlighting the added `delay`s in yellow and the added `force`s in blue, and (3) adds arrows originating from each inserted `delay`, pointing to the source of the laziness, thus explaining its decision to the programmer in an intuitive manner. Running the transformed program exhibits the desired performance.

2 Though `lcons` is not available in Racket, we simulate it with a macro to match the syntax of this dissertation. The macro wraps a `delay` around the second argument of a `cons`.

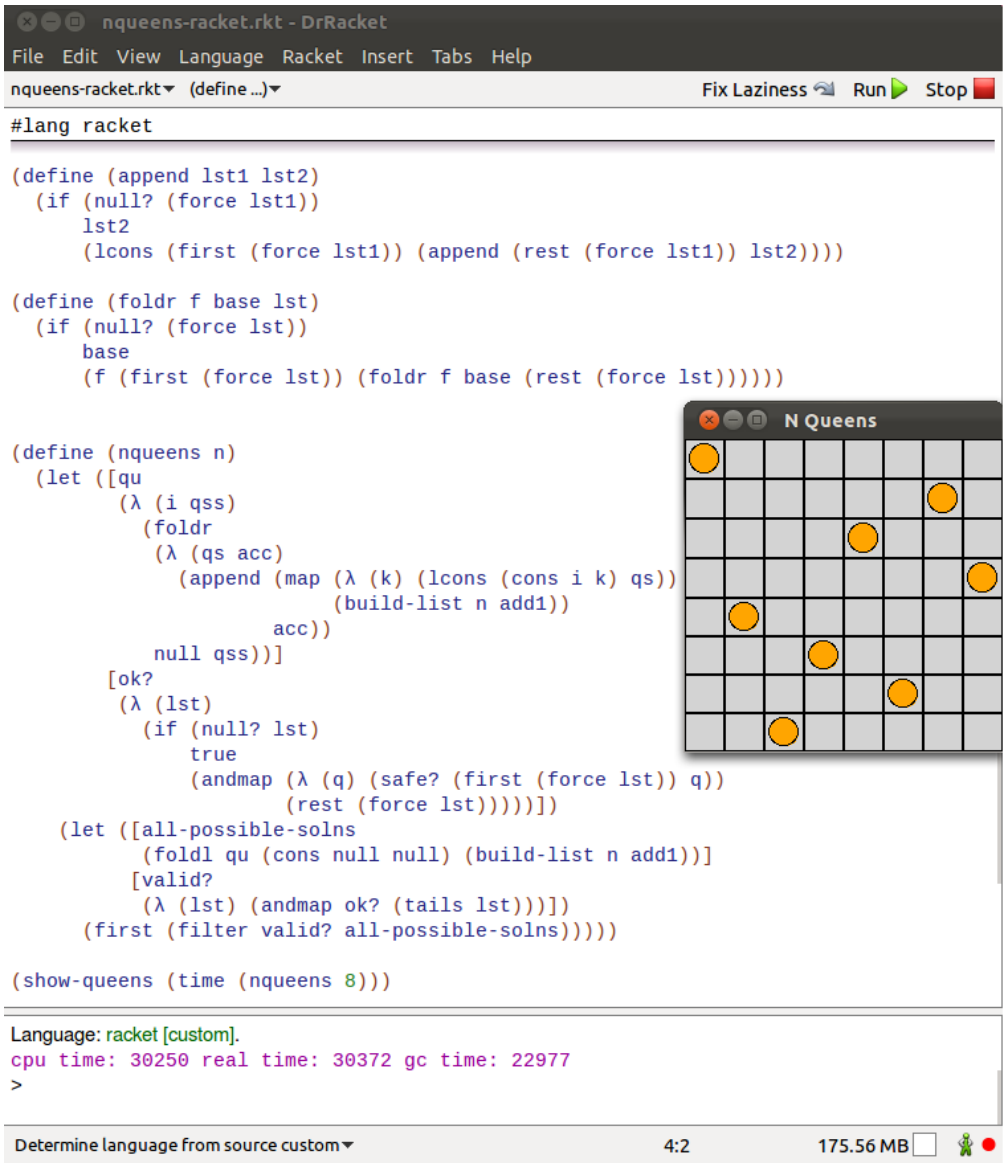

Figure 11: Evaluating n-queens in Racket with only lazy cons.

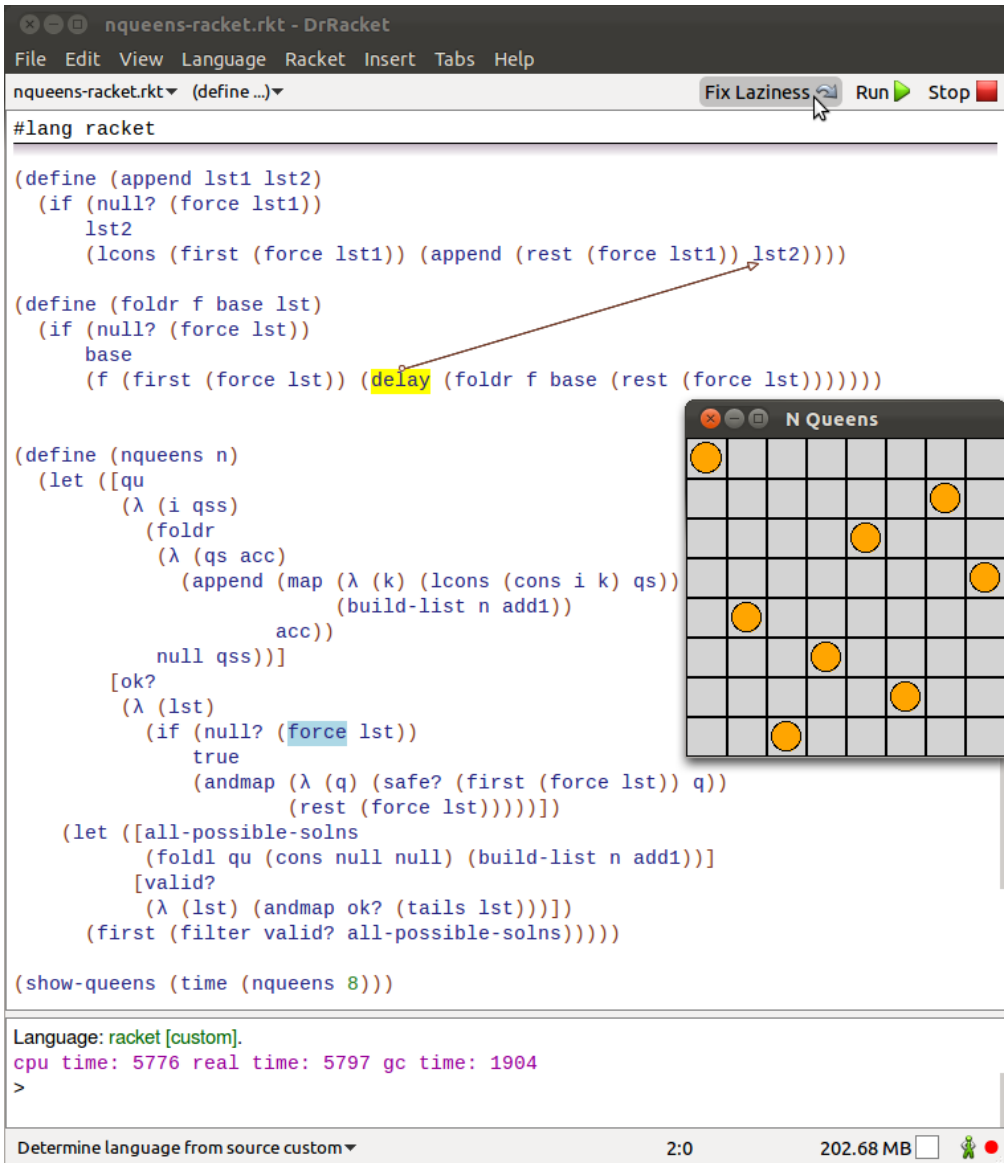

Figure 12: Evaluating n-queens in Racket after refactoring.

6 STATIC TOOL EVALUATION

To further evaluate our idea and our tool, we examined the Racket code base and some user-contributed packages for manual uses of laziness. We found several erroneous attempts at adding laziness and we verified that our tool would have prevented many such errors.[1] We consider this investigation a first confirmation of the usefulness of our tool. The rest of the section describes two of the examples.

The DMdA languages [17] allow students to write contracts for some data structures. These contracts are based on Findler et al.'s lazy contracts [27]. The contracts are primarily implemented via a constructor with a few lazy fields. Additionally, several specialized contract constructors for various data structures call the main constructor. However, since the specialized constructors are implemented with ordinary strict functions, the programmer must manually propagate the laziness to the appropriate arguments of these functions to preserve the intended lazy behavior. The situation is similar to the Scala example from chapter 1. Thus, a small amount of laziness in the main contract constructor requires several more `delays` scattered all throughout the program. Adding these `delays` becomes tedious as the program grows in complexity and unsurprisingly, a few were left out. Our tool identified the missing `delays`, which the author of the code confirmed and corrected with commits to the code repository.

A second example concerns queues and deques [66] implemented in Typed Racket [76], based on implicit recursive slowdown [62, Chapter 11], where laziness enables fast amortized operations and simplifies the implementation. The library contained several performance bugs, as illustrated by this code snippet from a deque enqueue function:

```
define enqueue(elem dq) =  ...
  let strictprt = ⟨extract strict part of dq⟩
      newstrictprt = ⟨combine elem and strictprt⟩
      lazyprt = force ⟨extract lazy part of dq⟩
      lazyprt1 = ⟨extracted from lazyprt⟩
      lazyprt2 = ⟨extracted from lazyprt⟩
  in Deque newstrictprt
           (delay ⟨combine lazyprt1 and lazyprt2⟩)
```

1 The examples were first translated to work with the syntax in this paper.

The function enqueues `elem` in deque `dq`, which has a lazy part and a strict part. In one execution path, the lazy part is extracted, forced, and separated into two additional pieces. Clearly, the forcing is unnecessary because neither of the pieces are used before they are inserted back into the new deque. Worse, the extra forcing slows the program significantly. For this example, activating our tool *twice* fixes the performance bug. For a reasonably standard benchmark, the fix reduced the running time by an order of magnitude. The authors of the code acknowledged the bug and merged our fix into the code repository.

Part IV

INTERLUDE: A CRITERIA FOR LAZINESS

The static analysis from part iii assumes that a programmer has partially annotated a program. Based on these annotations and a control-flow analysis, it suggests additional `delay` annotations. Because the tool considers only static information, it merely approximates where laziness is needed and often produces spurious suggestions. The requirement for some initial "seed" annotations contributes additional imprecision. On occasion, the tool must also be run more than once to find additional laziness but our research offers no guidance on this process.

The gaps in the static approach raise an fundamental question: "what exactly is the ground truth when it comes to laziness?" In other words, is there a precise criteria to use when determining whether an expression in a program should be lazy and when it should be strict. Since deploying laziness is a balancing act, with too little laziness causing extra complexity with no added benefits but too much laziness resulting in the costs overwhelming the gains, such a criteria would benefit programmers by informing them to what degree the program benefits from laziness at each location.[2]

Coming up with such a metric must rely on a programmer's intuition about the use of values in a program. An initial criteria might suggest that if an expression's result is not used, we should delay its evaluation. Informally, a value is *used* if it reaches a position that requires a specific (kind of) value to continue the evaluation process. Examples of such positions are the operands to primitive functions, the test in a conditional, and the function position in an application. In contrast, positions such as arguments to a programmer's functions and data constructors do not "use" their values. Thus the underlined argument in the following expression should be delayed:

$$(\lambda x.1)\ \underline{2+3}$$

Our initial laziness criterion does not cover the case where a program evaluates an expression more than once, e.g., via recursion, producing multiple values. We could extend our criterion to delay expressions where *none* of its values are used, but obviously this binary classification is useless for prac-

2 The material for part iv of this dissertation appeared previously in Chang and Felleisen [2014]: *Profiling for Laziness*, In: POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

tical cases. Consider the following pseudocode, which represents a function definition and its use in a typical strict functional language:

```
def rng n m =                                  (1)
  if n ⩾ m then nil                            (2)
  else n::rng (n+1) m                          (3)
                                               (4)
let lst = rng 0 1000                           (5)
in (second lst) + (third lst)                  (6)
```

A call to *rng* n m produces the list of integers in $[n, m)$. In this example, the (underlined) recursive call to *rng* is evaluated multiple times and produces multiple values. Although `second` and `third` use two of these values, the program is likely to benefit from delaying the underlined expression because most of the list is unused.

The example suggests a quantitative criterion for the injection of laziness into strict programs. Specifically, it hints that the ratio of the number of used results to the total number of results per expression might pinpoint delayable expressions. In the example, the recursive call to *rng* is evaluated $m - n = 1000$ times and two of the values are used, giving it a ratio of $2/1000$.

An expression with such a low ratio seems like a promising candidate for laziness, but as the following (boxed) changes to the example show, this first quantitative criterion is still insufficient:

```
def rng [f] n m =                              (1)
  if n ⩾ m then nil                            (2)
  else [f n]::rng [f] (n+1) m                  (3)
                                               (4)
let lst = rng [add1] 0 1000                    (5)
in (second lst) + (third lst)                  (6)
```

Here is some more accounting data:

```
f n [ln 3]: 2/1000 values used
rng f (n+1) m [ln 3]: 2/1000 values used
```

The expressions `f` n and *rng* `f` (n+1) m have equal ratios, seemingly indicating that the program benefits equally from delaying each. A programmer immediately understands, however, that this recommendation is misleading because delaying the call to *rng* would prevent most calls to `f`.

To combine these factors, we focus only on the unused values, and additionally weigh them by the number of *child-values* created during the creation of an unused value. Essentially, this *weight* is correlated to the size of the dynamic value-dependency tree whose root is the unused value. The weight of an *expression* then, which we dub its *laziness potential*, is roughly the average of the weights of all its unused values:

```
rng f (n+1) m [ln 3]: 2/1000 values used
- delaying 998 unused avoids 2990 subvalues, weight=2991
f n [ln 3]: 2/1000 values used
- delaying 998 unused avoids 0 subvalues, weight=1
```

A higher weight indicates a greater benefit from laziness. The exact calculation of laziness potential is discussed in chapter 7. The important takeaway here is that a metric designer must consider delaying the call to *rng* more beneficial than delaying the call to `f`, which aligns with a programmer's intuition.

Laziness potential seems promising as a criterion but the following "generate and filter" example demonstrates yet another problem:

```
def rng f n m =                                          (1)
  if n ⩾ m then nil                                      (2)
  else f n::rng f (n+1) m                                (3)
def filter p? lst =                                      (4)
  if null? lst then nil                                  (5)
  else let x = first lst                                 (6)
      in if p? x then x::filter p? (rest lst)            (7)
         else filter p? (rest lst)                       (8)
                                                         (9)
let lst = filter even? (rng add1 0 1000)                 (10)
in (second lst) + (third lst)                            (11)
```

For example, our informal metric would suggest delaying the first recursive call to *filter*, because it has a large weight:

```
filter p? (rest lst) [ln 7]: 2/500 used
- delaying 498 unused avoids 2486 subvalues, wgt=2487
```

Intuitively, however, the recursive call to *rng* should be delayed as well, but it does not appear to play a role in the current metric because *filter* "uses" the entire list.

In response, we should compute the metric iteratively:

1. after measuring the laziness potential, we next simulate delaying the expression with the highest laziness potential, and then
2. recalculate usages to possibly uncover additional unused values.

An iterative calculation of laziness potential would look like this:

```
~~~~~~~~~~~~~~ Profiling Summary: Round 0 ~~~~~~~~~~~~~~
  filter p? (rest lst) [ln 7]: 2/500 values used
  - delaying 498 unused avoids 2486 subvalues, wgt=2487

~~~~~~~~~~~~~~ Profiling Summary: Round 1 ~~~~~~~~~~~~~~
  rng f (n+1) m [ln 3]: 4/1000 values used
  - delaying 996 unused avoids 984 subvalues, wgt=2985
  f n [ln 3]: 5/1000 values used
  - delaying 995 unused avoids 0 subvalues, wgt=1
```

After round 0, the analysis would simulate a delay of the call to *filter* and then recomputes all usages, revealing unused values from the call to *rng* in round 1. This process would repeat until there are no more unused expressions. The summary report finally suggests delaying both the calls to *filter* and *rng*, because it would benefit program performance the most.

Part V

A DYNAMIC APPROACH

Part v of this dissertation translates our intuitive ideas about a criteria for laziness from part iv into a formal model and the implementation of a profiler that utilizes this model to suggest locations for laziness to a programmer. Chapter 7 formally describes the notion of laziness potential, chapter 8 presents our prototype profiler implementation, and chapter 9 demonstrates its effectiveness.[3]

3 The material for part v of this dissertation appeared previously in Chang and Felleisen [2014]: *Profiling for Laziness*, In: POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

7 LAZINESS POTENTIAL

7.1 PARTIALLY-LABELED λ-CALCULUS

As our informal presentation in part iv suggests, we consider *laziness potential* the key to determining where to insert effective laziness annotations in strict programs. We present a formal model for laziness potential in two stages. The first stage defines a labeled calculus for λ-calculus terms. The second stage extends the language with more realistic language constructs.

Our model starts from a labeled version of the untyped λ-calculus:

$$
\begin{aligned}
e \in Exp &= x \mid v \mid e\ e \mid {}^{\boldsymbol{\ell}}e \mid {}^{\boldsymbol{\ell}\cdot\ell}e \\
v \in LabVal &= w^{\vec{\ell}} \\
w \in Val &= \lambda x.e \\
x \in Var,\ \boldsymbol{\ell} &\in eLab,\ \ell \in vLab
\end{aligned}
$$

The syntax employs two kinds of labels. Static labels $\boldsymbol{\ell}$ (in bold) name a syntactic expression in the source program. Dynamic labels ℓ (plain) name a value during evaluation, i.e., the result of one runtime execution of an expression with a particular set of bindings for its free variables. Let a program be a closed expression e with only static labels. As the grammar indicates, static labels are *optional* decorations. During evaluation, a statically labeled expression may be paired with a dynamic label at the upper left. Evaluation may also tag *values* with any number of dynamic labels, all positioned at the upper right and denoted with $\vec{\ell}$. We use w when we wish to refer to a value without labels.

A program may evaluate an expression multiple times, e.g., via recursion, so a statically $\boldsymbol{\ell}$-labeled expression may be associated with a set of dynamic labels representing the values that this expression produces. Further, we wish to compute *usage* of these values, which requires tracking value flow, so values themselves may accumulate multiple dynamic ℓ labels to enable identification of the value's source expression(s). In the example $\underline{(\lambda x.x)\ (}\underline{(\lambda z.z)\ \lambda y.y}\underline{)}$, $\lambda y.y$ is the result of evaluating both underlined expressions and thus must have two dynamic labels.

Evaluation contexts [25] specify the order of evaluation in our labeled λ-calculus. They are the standard by-value evaluation contexts, plus an additional labeled context:

$$E \in Ctx = [\,] \mid E\ e \mid v\ E \mid {}^{\boldsymbol{\ell}\cdot\ell}E$$

These contexts dictate that reductions take place within a labeled expression only if it comes with a static and one dynamic label.

The $\longmapsto$ reduction relation specifies one step of evaluation. It is generated from three basic notions of reduction: β_v and two additional reductions that introduce and manipulate value labels:

$$\begin{array}{rcll} E[(\lambda x.e)^{\vec{\ell}}\ v] & \longmapsto & E[e\{x ::= v\}] & (\beta_v) \\ E[^{\boldsymbol{\ell}}e] & \longmapsto & E[^{\boldsymbol{\ell}\cdot\ell}e],\ \ell \notin E[^{\boldsymbol{\ell}}e] & (\mathit{vstart}) \\ E[^{\boldsymbol{\ell}\cdot\ell}w^{\vec{\ell}}] & \longmapsto & E[w^{\ell,\vec{\ell}}] & (\mathit{vend}) \end{array}$$

The semantics satisfies a conventional well-definedness theorem.

Theorem 4 (Well-Definedness)**.** *A program e either reduces to a value v or starts an infinitely long chain of reductions.*

Proof Sketch. The $\longmapsto$ relation satisfies unique decomposition, so e is a value or can be partitioned uniquely into an evaluation context and a redex. Using a progress-and-preservation approach [25], we can then show that this property is preserved for all reductions. □

From an extensional perspective, the labeled λ-calculus reduces programs just like the by-value λ-calculus [65]. Theorem 5 says that the labeled calculus produces the same result, modulo labels, as the unlabeled by-value semantics generated by β_v. The $\longmapsto_v$ relation is the by-value standard reduction and $\longmapsto\!\!\!\!\rightarrow_v$ is its reflexive, transitive closure. The function ξ (not defined) strips labels from programs.

Theorem 5 (Label Non-interference)**.**
For any labeled program e_1*, if* $e_1 \longmapsto\!\!\!\!\rightarrow e_2$*, then* $\xi(e_1) \longmapsto\!\!\!\!\rightarrow_v \xi(e_2)$*.*

Proof Sketch. Show, by cases on $\longmapsto$, that if $e_1 \longmapsto e_2$, then either $\xi(e_1) \longmapsto_v \xi(e_2)$ or $\xi(e_1) = \xi(e_2)$. □

From an intensional perspective, labels play a critical role. Expressions may be labeled in three different ways and the two label-related reductions specify transitions between labelings. Suppose a redex is statically labeled with $\boldsymbol{\ell}$. Before it is evaluated, a unique, dynamic label ℓ is generated and placed on the upper-left, as indicated by the *vstart* reduction. When such a labeled subexpression is reduced to a value, a *vend* reduction shifts the value label from the left-hand side to the right-hand side and discards the static label.

7.2 LABELING A PROGRAM

The labeled λ-calculus does not specify which expressions in a program to label statically. Hence, the calculus is applicable in a variety of scenarios, depending on the specific labeling strategy.

For the purposes of finding candidates for lazy evaluation, we employ a function $\mathcal{L}$ that maps unlabeled expressions to expressions with labels on arguments that are not already values:

$$\boxed{\mathcal{L} : Exp \to Exp}$$

$$\begin{aligned}
\mathcal{L}[\![x]\!] &= x \\
\mathcal{L}[\![\lambda x.e]\!] &= \lambda x.\mathcal{L}[\![e]\!] \\
\mathcal{L}[\![e_1\ e_2]\!] &= \mathcal{L}[\![e_1]\!]\ \mathcal{L}_{arg}[\![e_2]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}_{arg}[\![x]\!] &= x \\
\mathcal{L}_{arg}[\![\lambda x.e]\!] &= \lambda x.\mathcal{L}[\![e]\!] \\
\mathcal{L}_{arg}[\![e_1\ e_2]\!] &= \ell(\mathcal{L}[\![e_1]\!]\ \mathcal{L}_{arg}[\![e_2]\!])
\end{aligned}$$

The $\mathcal{L}$ function labels only applications because delaying values or variables is pointless. Furthermore, only applications in an argument position receive a label. Again, it is not beneficial to delay applications in a usage context because the created suspension is immediately forced. We also do not label function bodies. We prefer to delay its application to avoid spoiling the proper implementation of tail calls for the execution of annotated programs.

7.3 USAGE CONTEXTS

To determine whether a particular value is used during evaluation, we define usage contexts U:

$$\begin{aligned}
u &\in uCtx = [\,]\ e \\
U &\in UCtx = [\,] \mid E[u]
\end{aligned}$$

Intuitively, a context is a usage context if creating a redex requires a specific (kind of) value placed in the hole. In our core calculus, the function position of an application is the only usage context because it requires a λ value. In addition, the top-level context is a usage context and thus a program result is considered "used."

7.4 EXTENDING THE MODEL

By design, our model smoothly generalizes to constructs found in practical languages. Almost any evaluation-context-based reduction semantics can easily be extended to an appropriate labeled reduction system. The only requirements are to extend the usage contexts and labeling strategy.

Here we extend our model with typical features. Specifically, we add constants, let, conditionals, primitive arithmetic and boolean operations, lists and

$$
\begin{aligned}
e &= \ldots \mid n \mid b \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
&\quad \mid o\ e\ e \mid \texttt{and}\ e\ e \mid \texttt{or}\ e\ e \mid \texttt{not}\ e \\
&\quad \mid \texttt{cons}\ e\ e \mid \texttt{first}\ e \mid \texttt{rest}\ e \mid \texttt{nil} \mid \texttt{null?}\ e \\
&\quad \mid \texttt{delay}\ e \mid \texttt{force}\ e \\
o &= + \mid - \mid * \mid /,\ n \in \mathbb{N},\ b = \texttt{\#t} \mid \texttt{\#f} \\
w &= \ldots \mid n \mid b \mid \texttt{cons}\ v\ v \mid \texttt{nil} \mid \texttt{delay}\ e \\
E &= \ldots \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \\
&\quad \mid o\ E\ e \mid o\ v\ E \mid \texttt{and}\ E\ e \mid \texttt{and}\ v\ E,\ v \neq \texttt{\#f} \\
&\quad \mid \texttt{or}\ E\ e \mid \texttt{or}\ \texttt{\#f}\ E \mid \texttt{not}\ E \\
&\quad \mid \texttt{cons}\ E\ e \mid \texttt{cons}\ v\ E \mid \texttt{first}\ E \mid \texttt{rest}\ E \mid \texttt{nil?}\ E \\
&\quad \mid \texttt{force}\ E
\end{aligned}
$$

Figure 13: Syntax for an extended by-value language.

list operations, and `delay` and `force` laziness constructs; see figures 13 and 14. The reductions are conventional [25]:

- The true and false boolean literals are `#t` and `#f`. The semantics treats all non-`#f` values as true, as is standard in untyped (and some typed) languages.
- The `and` and `or` primitives are short-circuiting, meaning that the second argument is only evaluated if necessary. This behavior is reflected in the evaluation contexts for `and` and `or`.
- The `force` form is recursive, specified by the *frcd* rule, meaning that applying a single `force` to a suspension returns the underlying value no matter how many `delay`s it is wrapped in.[1].
- Untyped `force` is idempotent, as seen in the *frcv* rule. Applying `force` to a non-delayed value returns that value.

Here is the extended set of usage contexts:

$$
\begin{aligned}
u &= \ldots \mid \texttt{if}\ [\,]\ e\ e \mid o\ [\,]\ e \mid o\ v\ [\,] \\
&\quad \mid \texttt{and}\ [\,]\ e \mid \texttt{and}\ v\ [\,],\ v \neq \texttt{\#f} \mid \texttt{or}\ [\,]\ e \mid \texttt{or}\ \texttt{\#f}\ [\,] \\
&\quad \mid \texttt{not}\ [\,] \mid \texttt{first}\ [\,] \mid \texttt{rest}\ [\,] \mid \texttt{nil?}\ [\,] \mid \texttt{force}\ [\,]
\end{aligned}
$$

1 There is not one consensus among language designers whether `force` should be recursive, so not all languages behave according to our model.

$$
\begin{array}{rcll}
\mathsf{E}[\texttt{let}\ x = v\ \texttt{in}\ e] & \longmapsto & \mathsf{E}[e\{x ::= v\}] & (\textit{let}) \\
\mathsf{E}[\texttt{if}\ \texttt{\#f}\ \texttt{then}\ e_1\ \texttt{else}\ e_2] & \longmapsto & \mathsf{E}[e_2] & (\textit{iff}) \\
\mathsf{E}[\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2] & \longmapsto & \mathsf{E}[e_1],\ v \neq \texttt{\#f} & (\textit{ift}) \\
\mathsf{E}[\texttt{o}\ v_1\ v_2] & \longmapsto & \mathsf{E}[\delta\ v_1\ v_2] & (\textit{prim}) \\
\mathsf{E}[\texttt{or}\ v\ e] & \longmapsto & \mathsf{E}[v],\ v \neq \texttt{\#f} & (\textit{ort}) \\
\mathsf{E}[\texttt{or}\ \texttt{\#f}\ v] & \longmapsto & \mathsf{E}[v] & (\textit{orf}) \\
\mathsf{E}[\texttt{and}\ \texttt{\#f}\ e] & \longmapsto & \mathsf{E}[\texttt{\#f}] & (\textit{andf}) \\
\mathsf{E}[\texttt{and}\ v_1\ v_2] & \longmapsto & \mathsf{E}[v_2],\ v_1 \neq \texttt{\#f} & (\textit{andt}) \\
\mathsf{E}[\texttt{not}\ \texttt{\#f}] & \longmapsto & \mathsf{E}[\texttt{\#t}] & (\textit{notf}) \\
\mathsf{E}[\texttt{not}\ v] & \longmapsto & \mathsf{E}[\texttt{\#f}],\ v \neq \texttt{\#f} & (\textit{nott}) \\
\mathsf{E}[\texttt{first}\ (\texttt{cons}\ v_1\ v_2)] & \longmapsto & \mathsf{E}[v_1] & (\textit{fst}) \\
\mathsf{E}[\texttt{rest}\ (\texttt{cons}\ v_1\ v_2)] & \longmapsto & \mathsf{E}[v_2] & (\textit{rst}) \\
\mathsf{E}[\texttt{nil?}\ \texttt{nil}] & \longmapsto & \mathsf{E}[\texttt{\#t}] & (\textit{nil}) \\
\mathsf{E}[\texttt{nil?}\ v] & \longmapsto & \mathsf{E}[\texttt{\#f}],\ v \neq \texttt{nil} & (\textit{nil}) \\
\mathsf{E}[\texttt{force}\ (\texttt{delay}\ e)] & \longmapsto & \mathsf{E}[\texttt{force}\ e] & (\textit{frcd}) \\
\mathsf{E}[\texttt{force}\ v] & \longmapsto & \mathsf{E}[v],\ v \neq \texttt{delay}\ e & (\textit{frcv})
\end{array}
$$

Figure 14: Semantics for an extended by-value language.

Finally, we extend our labeling function to mark expressions of interest, maintaining the goal of finding expressions to delay. In addition to arguments in a function application, we label the bound expression in a `let` and the arguments to `cons`. The part of the function indicated by ellipses traverse all other expressions in a homomorphic manner:

$$\begin{aligned}
\mathcal{L}[\![e_1\ e_2]\!] &= \mathcal{L}[\![e_1]\!]\ \mathcal{L}_{arg}[\![e_2]\!] \\
\mathcal{L}[\![\texttt{let}\ x = e_1\ \texttt{in}\ e_2]\!] &= \texttt{let}\ x = \mathcal{L}_{arg}[\![e_1]\!]\ \texttt{in}\ \mathcal{L}[\![e_2]\!] \\
\mathcal{L}[\![\texttt{cons}\ e_1\ e_2]\!] &= \texttt{cons}\ \mathcal{L}_{arg}[\![e_1]\!]\ \mathcal{L}_{arg}[\![e_2]\!] \\
&\dots
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}_{arg}[\![x]\!] &= x \\
\mathcal{L}_{arg}[\![v]\!] &= \mathcal{L}[\![v]\!] \\
\mathcal{L}_{arg}[\![e]\!] &= {}^{\boldsymbol{\ell}}(\mathcal{L}[\![e]\!]),\ \text{if } e \neq x \text{ or } v
\end{aligned}$$

7.5 CALCULATING LAZINESS POTENTIAL

We calculate laziness potential via functions that extract information from the propagation of labels in reduction sequences. Let *Red* be the set of finite reduction sequences. In the following definitions, a comma-separated series of expressions represents an element of *Red*, i.e., a trace. We also use $(\mathbb{T}\ e)$ to denote e's complete trace.

The $\mathbb{V}$ function takes a reduction sequence and a static label $\boldsymbol{\ell}$ and returns a set of dynamic labels representing the values generated by the $\boldsymbol{\ell}$-labeled expression over the course of reduction:

$$\boxed{\mathbb{V} : \mathit{Red} \times \mathit{eLab} \to \mathcal{P}(\mathit{vLab})}$$

$$\begin{aligned}
\mathbb{V}\ (e)\ \boldsymbol{\ell} &= \emptyset \\
\mathbb{V}\ (\mathsf{E}[e^{\ell}], \mathsf{E}[{}^{\boldsymbol{\ell}\cdot\ell}e], e', \dots)\ \boldsymbol{\ell} &= \{\ell\} \cup \mathbb{V}\ (e', \dots)\ \boldsymbol{\ell} \\
\mathbb{V}\ (e, e', \dots)\ \boldsymbol{\ell} &= \mathbb{V}\ (e', \dots)\ \boldsymbol{\ell},\ \text{if } e \neq \mathsf{E}[{}^{\boldsymbol{\ell}}e_0]
\end{aligned}$$

Intuitively, $\mathbb{V}$ inspects all *vstart* steps. In the last example from part iv, call it program `P`, if the recursive call to *filter* has static label $\boldsymbol{\ell}_1$, then a new dynamic label is created every time ${}^{\boldsymbol{\ell}_1}(\textit{filter}\ \texttt{p?}\ (\texttt{rest lst}))$ is a redex and $\mathbb{V}\ (\mathbb{T}\ \texttt{P})\ \boldsymbol{\ell}_1 = \{\ell_1, \dots, \ell_{500}\}$.

The $\mathbb{U}$ function counts how many times a specified value is used in a given reduction sequence, as dictated by usage contexts:

$$\boxed{\mathbb{U} : \mathit{Red} \times \mathit{vLab} \to \mathbb{N}}$$

$$\begin{aligned}
\mathbb{U}\ (w^{\vec{\ell}})\ \ell &= 1, \text{ if } \ell \in \vec{\ell} \\
\mathbb{U}\ (e)\ \ell &= 0, \text{ if } e \neq v, \text{ or } e = w^{\vec{\ell}} \text{ but } \ell \notin \vec{\ell} \\
\mathbb{U}\ (\mathsf{U}[w^{\vec{\ell}}], e', \ldots)\ \ell &= 1 + (\mathbb{U}\ (e', \ldots)\ \ell), \text{ if } \ell \in \vec{\ell} \\
\mathbb{U}\ (e, e', \ldots)\ \ell &= \mathbb{U}\ (e', \ldots)\ \ell, \\
&\qquad \text{if } e \neq \mathsf{U}[v], \text{ or } e = \mathsf{U}[w^{\vec{\ell}}] \text{ but } \ell \notin \vec{\ell}
\end{aligned}$$

An ℓ-labeled value is unused in $(\mathbb{T}\ e)$ if $\mathbb{U}\ (\mathbb{T}\ e)\ \ell = 0$. The **unused** function relies on $\mathbb{U}$ and $\mathbb{V}$ to compute (the labels of) all the unused values produced by a particular expression in e:

$$\boxed{\textbf{unused} : \mathit{Exp} \times \mathit{eLab} \to \mathcal{P}(\mathit{vLab})}$$

$$\textbf{unused}\ (e, \boldsymbol{\ell}) = \{\ell \mid \ell \in \mathbb{V}\ (\mathbb{T}\ e)\ \boldsymbol{\ell},\ \mathbb{U}\ (\mathbb{T}\ e)\ \ell = 0\}$$

For sample program P, if `second` and `third` are desugared to `firsts` and `rests`, then two created values reach a `first [ ]` or `rest [ ]` usage context, so for $\ell_i \in \{\ell_1, \ldots, \ell_{500}\}$, $\mathbb{U}\ (\mathbb{T}\ \mathsf{P})\ \ell_i = 1$ for two values of `i` and is otherwise 0. Thus, $\textbf{unused}\ (\mathsf{P}, \boldsymbol{\ell}_1) = 2$.

The $\mathbb{C}$ function returns the labels of all the child-values that are generated *during* the creation of a given ℓ-labeled value:

$$\boxed{\mathbb{C} : \mathit{Red} \times \mathit{vLab} \to \mathcal{P}(\mathit{vLab})}$$

$$\begin{aligned}
\mathbb{C}\ (e)\ \ell &= \emptyset \\
\mathbb{C}\ (\mathsf{E}[\overset{\boldsymbol{\ell}\cdot\ell}{e}], e', \ldots)\ \ell &= \mathbb{C}'\ (e', \ldots)\ \ell \\
\mathbb{C}\ (e, e', \ldots)\ \ell &= \mathbb{C}\ (e', \ldots)\ \ell, \text{ if } e \neq \mathsf{E}[\overset{\boldsymbol{\ell}\cdot\ell}{e_0}]
\end{aligned}$$

$$\boxed{\mathbb{C}' : \mathit{Red} \times \mathit{vLab} \to \mathcal{P}(\mathit{vLab})}$$

$$\begin{aligned}
\mathbb{C}'\ (e)\ \ell &= \emptyset \\
\mathbb{C}'\ (\mathsf{E}[w^{\ell,\vec{\ell}}], e', \ldots)\ \ell &= \emptyset \\
\mathbb{C}'\ (\mathsf{E}[\overset{\boldsymbol{\ell}}{e}], \mathsf{E}[\overset{\boldsymbol{\ell}\cdot\ell'}{e}], e', \ldots)\ \ell &= \{\ell'\} \cup \mathbb{C}'\ (e', \ldots)\ \ell \\
\mathbb{C}'\ (e, e', \ldots)\ \ell &= \mathbb{C}'\ (e', \ldots)\ \ell, \text{ if } e \neq \mathsf{E}[\overset{\boldsymbol{\ell}}{e_0}]
\end{aligned}$$

When the creation of the specified value begins, the $\mathbb{C}$ function dispatches to $\mathbb{C}'$. When this helper function encounters a reduction that creates another

value label, it adds it to its results. When evaluation of the specified value completes or the reduction sequence ends, the helper function stops collecting labels.

The **created** function uses $\mathbb{C}$ to compute (the labels of) all child-values generated during the creation of all the unused values of a given $\boldsymbol{\ell}$-labeled expression in e:

$$\boxed{\textbf{created} : \mathit{Exp} \times \mathit{eLab} \to \mathcal{P}(\mathit{vLab})}$$

$$\textbf{created}\ (e, \boldsymbol{\ell}) = \bigcup_{\ell \in \textbf{unused}\ (e, \boldsymbol{\ell})} \mathbb{C}\ (\mathbb{T}\ e)\ \ell$$

In our running example, **created** $(\mathsf{P}, \boldsymbol{\ell}_1)$ tallies the additional values created while evaluating each of $\ell_1, \ldots, \ell_{500}$, about 2,500.

Finally, the **LP** function computes the *laziness potential* of an $\boldsymbol{\ell}$-labeled expression in program e. Roughly, the function computes the "average" number of child-values generated during the creation of an unused value of expression $\boldsymbol{\ell}$, if **unused** $(e, \boldsymbol{\ell}) \neq \emptyset$:

$$\boxed{\textbf{LP} : \mathit{Exp} \times \mathit{eLab} \to \mathbb{R}}$$

$$\textbf{LP}\ (e, \boldsymbol{\ell}) = \frac{|(\textbf{unused}\ (e, \boldsymbol{\ell})) \cup (\textbf{created}\ (e, \boldsymbol{\ell}))|}{|(\textbf{unused}\ (e, \boldsymbol{\ell})) \setminus (\textbf{created}\ (e, \boldsymbol{\ell}))|}$$

The function uses **unused** to compute the (labels of the) unused values created by the specified $\boldsymbol{\ell}$-labeled expression and **created** to compute the (labels of the) child-values. Together, these two sets represent the total number of wasted values for which expression $\boldsymbol{\ell}$ is responsible. The numerator of the calculation uses the union of the two sets to avoid double-counting values, for example when expression $\boldsymbol{\ell}$ is a recursive call. To compute the desired ratio, **LP** divides the total value count by the number of unused values produced by expression $\boldsymbol{\ell}$. The denominator of the calculation additionally subtracts those unused values that are child-values of other unused values. This appropriately gives higher weight to recursive calls, which matches a programmer's intuition.

In the running example, all but one of the 498 unused values are induced by the first recursive call to *filter*. Thus the denominator in the laziness potential for $\boldsymbol{\ell}_1$ is one and **LP** $(\mathsf{P}, \boldsymbol{\ell}_1) \approx 2500$.

The **LP** function computes the laziness potential of one particular expression in a program. Delaying the expression with the highest laziness potential should generate the largest possible benefit but doing so may reveal additional opportunities for laziness. Hence, we iteratively define the following series of **LP** functions:

$$\begin{aligned}
\mathbf{LP}_0\ (e, \boldsymbol{\ell}) &= \mathbf{LP}\ (e, \boldsymbol{\ell}) \\
\mathbf{LP}_{i+1}\ (e, \boldsymbol{\ell}) &= \mathbf{LP}_i\ (C^{force}[\texttt{delay}\ ^{\boldsymbol{\ell}^{max}}e^{force}_{max}], \boldsymbol{\ell}), \\
&\qquad \text{where } e = C[^{\boldsymbol{\ell}^{max}}e_{max}] \\
&\qquad\quad \text{and } \boldsymbol{\ell}^{max} = \underset{\boldsymbol{\ell}' \in e, \mathbf{unused}\ (e, \boldsymbol{\ell}') \neq \emptyset}{\arg\max}\ \mathbf{LP}_i\ (e, \boldsymbol{\ell}')
\end{aligned}$$

An $\mathbf{LP}_{i+1}$ function first `delay`s the subexpression with (according to $\mathbf{LP}_i$) maximal laziness potential and adds appropriate forcing for the used values of that subexpression. It then uses $\mathbf{LP}_i$ on this transformed program to determine the next opportunity for laziness injection. In the definition, $\boldsymbol{\ell}^{max}$ labels e_{max}, the expression with the most laziness potential, and C is its context. Since some values produced by e_{max} may still be used, C^{force} is C augmented with `force`s and $^{\boldsymbol{\ell}^{max}}e^{force}_{max}$ is $^{\boldsymbol{\ell}^{max}}e_{max}$ augmented with `force`s around those usage contexts that require e_{max}'s values. Though it is sufficient to simply force every usage context in C, a flow analysis like in part iii may compute a more a precise placement of `force`s.

In our running example, assume $\mathbf{LP}_0$ determines that the $\boldsymbol{\ell}_1$-labeled *filter* expression has the highest laziness potential. Then $\mathbf{LP}_1$ first delays $\boldsymbol{\ell}_1$ and inserts `force`s at its use points. $\mathbf{LP}_1$ then re-runs the augmented program and recomputes laziness potential for other expressions. Assume that both arguments to `cons` in *rng* also have static labels. As mentioned in part iv, the $\mathbf{LP}_1$ computations would find that the call to both *rng* and `f` create unused values but that only the unused values of *rng* subsequently induce many more values and thus has higher laziness potential.

A Non-theorem An astute reader may wonder whether we can relate the laziness potential of an expression to a formal step-counting cost model. It would show that the suggestions are guaranteed to improve a program's performance. Unfortunately, the predictive capability of a step-counting model depends on how laziness is implemented meaning the desired theorem may hold only for semantic cost models truly close to one specific implementation.

8

PROFILER IMPLEMENTATION

An implementation of a laziness profiler cannot use the definition of $\mathbf{LP}_{\mathtt{i}}$ as a blueprint because it is highly impractical to re-run a modified program for every $\mathtt{i}$. Instead our profiler creates a value dependence graph in an online fashion during a single execution and uses this graph to perform laziness potential calculations.

Information collected during evaluation	**Math function**
ALL-VNUMS : $\mathcal{P}(vLab)$	
set of labels representing all values created during evaluation	
EXPR-VALS : $eLab \to \mathcal{P}(vLab)$	$\mathbb{V}$
maps an expression label to a set of value labels representing the values created by that expression during evaluation	
CHILD-VNUMS : $vLab \to vLab$	$\mathbb{C}$
maps a value label ℓ to a value label $\ell_{\mathtt{hi}}$ such that the set of values created while evaluating ℓ is the exclusive interval $(\ell, \ell_{\mathtt{hi}})$; i.e., the ℓ-rooted subtree in the value-dependency tree	
USES : $vLab \to \mathbb{N}$	$\mathbb{U}$
the number of times a value is used during evaluation	
USES-BY-VNUM : $vLab \to vLab \to \mathbb{N}$	
maps a value label ℓ to another map of value labels to counts, representing the value usage while evaluating ℓ only; to avoid double-counting uses, a value with label $\ell_{\mathtt{used}}$ is considered used while evaluating value ℓ only if ℓ is the immediate ancestor in the program's value-dependency tree; for example, if value ℓ is created while evaluating another value with label $\ell_{\mathtt{parent}}$, then $\ell_{\mathtt{used}}$ is considered used while evaluating ℓ but *not* while evaluating $\ell_{\mathtt{parent}}$	

Table 1: Information collected by the profiler.

8.1 USAGE INFORMATION GATHERING

Table 1 describes the collected information. It also lists the analogous mathematical function from section 7.5. The $\mathcal{I}$ function instruments a labeled λ-calculus program to collect this information:

$$\boxed{\mathcal{I} : Exp \to Exp}$$

$$\begin{aligned}
\mathcal{I}[\![x]\!] &= x \\
\mathcal{I}[\![\lambda x.e]\!] &= \lambda x.\mathcal{I}[\![e]\!] \\
\mathcal{I}[\![e_1\ e_2]\!] &= \mathcal{U}[\![\mathcal{I}[\![e_1]\!]]\!]\ \mathcal{A}[\![\mathcal{I}[\![e_2]\!]]\!]
\end{aligned}$$

$\mathcal{I}$ relies on two additional functions, $\mathcal{A}$ and $\mathcal{U}$, to instrument argument and usage positions, respectively. $\mathcal{I}$ is defined only for core λ-calculus expressions; the application of $\mathcal{A}$ exactly follows the labeling strategy from chapter 7. In other words, a subexpression in a program is instrumented as an "argument" only if it has a static label $\boldsymbol{\ell}$. In addition, a subexpression is instrumented with $\mathcal{U}$ if it resides in a usage context.

Here are $\mathcal{A}$ and $\mathcal{U}$, which inject imperative code fragments that perform the required accounting during program execution:

$$\begin{aligned}
\mathcal{A}[\![e^{\ell}]\!] = {} & \texttt{let } \ell_{new} = \texttt{next-vnum } () \texttt{ in} \\
& \quad \texttt{all-vnums} \cup\!= \{\ell_{new}\} \\
& \quad \texttt{expr-vals}[\boldsymbol{\ell}] \cup\!= \{\ell_{new}\} \\
& \quad \texttt{push-ctxt-vnum } \ell_{new} \\
& \quad \texttt{let } w^{\vec{\ell}} = e^{\ell} \texttt{ in} \\
& \quad \texttt{pop-ctxt-vnum } () \\
& \quad \texttt{child-vnums}[\ell_{new}] = \texttt{current-vnum } () \\
& \quad \texttt{return } w^{\ell_{new},\vec{\ell}}
\end{aligned}$$

$$\begin{aligned}
\mathcal{U}[\![e]\!] = {} & \texttt{let } \ell_{ctxt} = \texttt{top-ctxt-vnum } () \texttt{ in} \\
& \quad \texttt{let } w^{\vec{\ell}} = e \texttt{ in} \\
& \quad \texttt{for } \ell \in \vec{\ell}\,: \\
& \quad\quad \texttt{uses}[\ell] \mathrel{+}= 1 \\
& \quad\quad \texttt{uses-by-vnum}[\ell_{ctxt}][\ell] \mathrel{+}= 1 \\
& \quad \texttt{return } w^{\vec{\ell}}
\end{aligned}$$

In these definitions, $\texttt{let } w^{\vec{\ell}} = e$ is a pattern-matching notation that means evaluate e, bind the resulting value to w, and bind any resulting value labels to the vector $\vec{\ell}$. The $\mathcal{A}$ and $\mathcal{U}$-instrumented code generates unique dynamic labels from a counter, and it tracks which value is being evaluated via a value stack. The interfaces for the counter and the stack are described in table 2.

counter	
`current-vnum`	returns the current count
`next-vnum`	returns then increments current count
stack	
`push-ctxt-vnum`	adds a value label to the top of the stack
`pop-ctxt-vnum`	removes top label from the context stack
`top-ctxt-vnum`	returns top label but does not pop it

Table 2: Auxiliary functions for instrumenting functions.

8.2 POST-EXECUTION ANALYSIS

Before we show the loop that computes $\mathbf{LP}_i$, we introduce functions that use the collected information from table 1:

$$\texttt{unused}\ \boldsymbol{\ell} = \{\ell \mid \ell \in \textsc{expr-vals}[\boldsymbol{\ell}],\ \ell \in \textsc{all-vnums},\ \textsc{uses}[\ell] = 0\}$$

$$\texttt{created}_v\ \ell = \{\ell_{sub} \mid \ell < \ell_{sub} < \textsc{child-vnums}[\ell],\ \ell_{sub} \in \textsc{all-vnums}\}$$

$$\texttt{created}\ \boldsymbol{\ell} = \bigcup_{\ell \in \texttt{unused}\ \boldsymbol{\ell}} \texttt{created}_v\ \ell$$

$$\texttt{LP}\ \boldsymbol{\ell} = \frac{|(\texttt{unused}\ \boldsymbol{\ell}) \cup (\texttt{created}\ \boldsymbol{\ell})|}{|(\texttt{unused}\ \boldsymbol{\ell}) \setminus (\texttt{created}\ \boldsymbol{\ell})|}$$

For program e, the main processing loop then looks like this:

$$\begin{aligned}
&\texttt{while}\ (\bigcup_{\boldsymbol{\ell} \in e} \texttt{unused}\ \boldsymbol{\ell}) \neq \emptyset : \\
&\quad \boldsymbol{\ell} = \operatorname*{arg\,max}_{\boldsymbol{\ell}' \in e, \texttt{unused}\ \boldsymbol{\ell}' \neq \emptyset} (\texttt{LP}\ \boldsymbol{\ell}') \\
&\quad \texttt{for}\ \ell \in (\texttt{unused}\ \boldsymbol{\ell}) : \texttt{erase}\ \ell \\
&\quad \texttt{record-result}\ \boldsymbol{\ell}
\end{aligned}$$

The loop performs another iteration as long as some expression in the program has an unused value. If all values are used, i.e., the program cannot benefit from laziness, then the profiler does not report any suggestions. At the start of each iteration, the analysis selects an expression with the currently greatest laziness potential, identified with label $\boldsymbol{\ell}$ above. The analysis

then simulates delaying this expression by "erasing" its unused values via `erase`:

```
erase ℓ =
  sub-uses-by-vnum ℓ
  ALL-VNUMS \= {ℓ}
  for ℓ_sub ∈ (created_v ℓ) : erase ℓ_sub
sub-uses-by-vnum ℓ =
  for (ℓ_used, n) ∈ USES-BY-VNUM[ℓ] : USES[ℓ_used] −= n
```

The `erase` function erases a value by (1) subtracting its usage of values from the total usage counts; (2) marking the value as erased by removing it from ALL-VNUMS, so it is not considered in subsequent calculations; and (3) recursively erasing all child-values that were created while evaluating the parameter value.

Finally, at the end of each iteration, the analysis records the delayed expression via `record-result`, so it can present a summary to the programmer after the calculations terminate. The main loop always terminates since each iteration of the loop "erases" at least one unused value.

Theorem 6. *The profiler implements the labeled λ-calculus model.*

Proof Sketch. The interesting part is showing that our post-execution analysis implements the functions from section 7.5. Specifically, the ith iteration of the main loop in this section corresponds to the $\mathbf{LP}_i$ function. In an iteration, both the model and implementation first compute the expression with highest laziness potential. The implementation then "erases" information pertaining to this expression and loops, while the math model inserts `delay`s and `force`s and re-runs the program. Thus the correctness of the implementation hinges on a correspondence between the `erase` function and the `delay` and `force` transformation in $\mathbf{LP}_i$.

Delaying an expression and then forcing the needed thunks eliminates: (1) the unused values produced by that expression, (2) the usages incurred while producing those unused values, and (3) all child-values of those unneeded values and their usages. Similarly, an iteration of the main loop in the implementation calls `erase` to eliminate: (1) all unused values of the highest potential expression by removing them from ALL-VNUMS, the set of all values produced during execution, (2) subtracts the usages incurred by those unused values, as recorded in USES-BY-VNUM, from the total usage counts in USES, and (3) recursively calls `erase` for each (remaining) child-value, as recorded in CHILD-VNUMS.

A Note on Effects The theorem does not hold once side-effects as simple as exception handling are added to the language, although our empirical evidence suggests that this is not an issue in practice. □

8.3 IMPLEMENTATION

We have implemented a prototype of the profiler described in this section for most of the Racket language [29]. The profiler supports all the language constructs from section 7.4, as well as several other frequently used Racket forms such as pattern matching (`match`), named records (`struct`), sequence iterations and comprehensions (`for`), additional binding forms (`define`, `let*`, etc.), much of the object system, and a large part of the syntax system. We use Racket's syntax system to modify the compiler so that it implements $\mathcal{I}[\![\cdot]\!]$, $\mathcal{U}[\![\cdot]\!]$, and $\mathcal{A}[\![\cdot]\!]$. Using this compiler API greatly simplifies the implementation of our profiler.

A Note on Performance Preliminary measurements indicate roughly a two to three order-of-magnitude slowdown when profiling certain degenerate programs. This kind of slowdown is not unreasonable for high-level profilers because programmers are expected to test their code with ***small*** *representative inputs*. Programs requiring laziness are *especially* suited for this style of testing because they generate lots of excess unused data, but so long as the ratio of unused to used data remains the same, it does not matter to the profiler whether the absolute amount of data is large or small.

9 PROFILER EVALUATION

To demonstrate the usefulness of the laziness potential metric, we present the results of profiling a range of real-world examples known to benefit from laziness: some of Okasaki's purely functional data structures, monadic parser combinators, and AI game players. We show that in most cases, the profiler reproduces the knowledge of experts. When our results differ from the recommendation of experts, we turn to wall-clock measurements to further evaluate our profiler's output.

9.1 GENERAL PROFILING ISSUES AND EXPERIMENTAL SETUP

In general, profilers produce meaningful results only when given representative programs. For example, it is impossible to demonstrate the effectiveness of a memory profiler using programs without memory leaks. Similarly, we wish to demonstrate our profiler's effectiveness at inserting laziness into strict programs for performance reasons; thus we choose well-known uses of laziness rather than arbitrary programs and use inputs designed to force our test applications to rely on laziness.

Our experiments proceeds as depicted in figure 15.

1. We implemented one of the expert examples;
2. we then removed the laziness from the example to get a strict program;
3. we then profiled this strict version in a representative context;
4. and finally, we re-inserted `delay`s according to the profiler's suggestions. We inserted appropriate `force`s by hand, but one could in principle derive the positions automatically (see part iii).

Each iteration of our experiment yields two lazy versions of a program known to benefit from laziness: a version implemented by experts and a version produced using suggestions from our profiler. If the profiler-produced version resembles the expert version, that suggests that our profiler can duplicate the expertise of experienced lazy programmers.

We conducted all experiments in Racket, an untyped language. We ported all implementations verbatim as we found them, except for a few type-related[1] discrepancies.

1 Okasaki points out [62, page 35] that laziness in a typed language occasionally requires extra annotations "to make the types work out". For example, typed languages must delay an empty

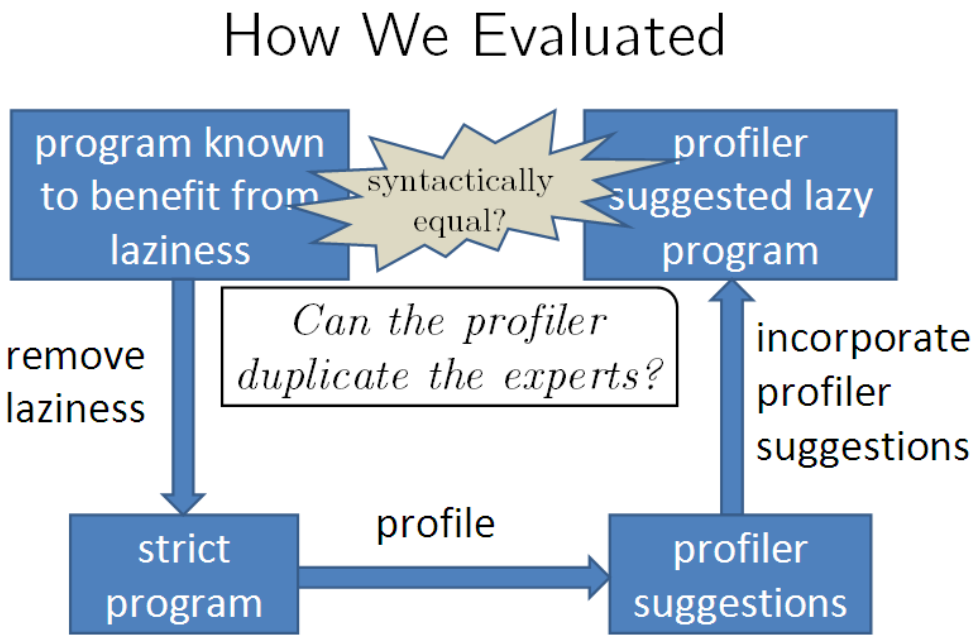

Figure 15: Experimental setup for evaluating our lazy profiler.

9.2 PURELY FUNCTIONAL DATA STRUCTURES

Okasaki's purely functional data structures make up a well-known library of algorithms that benefit from some degree of laziness in the data representation. At the same time, these data structures do not assume that the underlying language is lazy. Hence they are a nearly ideal test bed for checking the usefulness of our profiler.

The appropriateness of our comparisons depends on the design of a particular data structure. We therefore classify Okasaki's lazy data structures into two categories:

1. The first kind of lazy data structure is derived from an existing strict data structure. Okasaki added laziness to improve performance without making any other changes.

2. The second kind of data structure is either designed with laziness in mind or is converted from an existing strict data structure with changes more significant than simply adding laziness.

The first kind of data structure, dubbed *basic*, is ideal for evaluating our profiler because a straightforward comparison of the laziness annotations is appropriate. For the second kind of data structure, which we refer to as *insightful*, the laziness is possibly integral to its design and removing the laziness may not yield a realistic strict data structure. Hence, any results concerning insightful structures may be less general than basic ones, but enlightening nonetheless.

Tables 3 and 4 describe several data structures, basic and insightful, respectively, and presents the results of our comparisons. For most examples, our profiler suggests laziness syntactically identical to Okasaki's version. In some cases, however, our profiler suggests what appears to be a better use of laziness, and we present these latter cases in detail.

9.2.1 *Banker's Queue*

Our profiler-assisted banker's queue and deque implementations improve on those of Okasaki. To help readers understand the differences and how they come about, we present the queue experiment in more detail.

The canonical functional queue is implemented with two eager lists, a "rear" list onto which elements are added to the queue, and a "front" list from

---

stream tail, while in an untyped language, this annotation is unneeded since the empty stream is a value. In general, due to a different data representation of suspensions, laziness in typed languages requires a pairing of `delay`s and `force`s, while the universal object representation in untyped languages enables a more flexible recursive and idempotent `force`.

Data Structure	Description	Result
Basic Data Structures		
banker's queue	canonical two-list functional queue with streams	Ⓟ
banker's deque	like the banker's queue, but allows inserting and deleting from both ends	Ⓟ
binomial heap	stores ordered elements in a series of increasingly sized trees, analogous to the representation of binary numbers; supports constant amortized time insert and log worse-case merge, delete min, and min operations	⊜
pairing heap	stores ordered elements as a tree where each node consists of a heap element and a list of subheaps; supports constant worst-case time insert, merge, and min operations, and log amortized time delete min operation	⊜

Ⓟ = profiler-assisted implementation outperforms Okasaki's
⊜ = profiler suggests the same lazy annotations as Okasaki's
⊛ = mixed results

Table 3: Okasaki vs profiler-generated basic lazy data structures.

Data Structure	Description	Result
Insightful Data Structures		
physicist's queue	canonical two-list functional queue with a stream front list, an eager front list cache, and a delayed rear list	⊛
real-time queue	converted banker's queue with an incrementally reversed rear list, with the goal of achieving worst case, rather than amortized, bounds	⊜
real-time deque	double-ended version of real-time queue	⊜
bootstrapped queue	improves on banker's queue by eliminating redundancy in append operation.	⊜
catenable list	lists that support constant amortized time append	⊜
implicit queue	achieves constant amortized operations via implicit recursive slowdown	ℙ
implicit deque	double-ended version of implicit queue	ℙ
Other Functional Data Structures		
finger tree	Hinze and Paterson's general purpose data structure (related to implicit queues)	ℙ

ℙ = profiler-assisted implementation outperforms Okasaki's

⊜ = profiler suggests the same lazy annotations as Okasaki's

⊛ = mixed results

Table 4: Okasaki vs profiler-generated insightful lazy data structures.

```
def enq x (Q f lenf r lenr) = chk f lenf (x::r) (lenr+1)

def chk f lenf r lenr =
  if lenr ⩽ lenf then Q f lenf r lenr
  else [Q (f ++ rev r) (lenf + lenr) nil 0]rotation

def hd (Q nil _ _ _) = error
  | hd (Q (x::f) lenf r lenr) = x
def tl (Q nil _ _ _) = error
  | tl (Q (x::f) lenf r lenr) = chk f (lenf−1) r lenr

def (++) nil lst = lst
  | (++) (x::xs) lst = x::(xs ++ lst)

def rev lst = rev/acc lst nil
def rev/acc nil acc = acc
  | rev/acc (x::xs) acc = rev/acc xs (x::acc)
```

Figure 16: Functional queue implementation without laziness.

which elements are removed. A record data definition, $(\mathrm{Q}\ \mathit{front}\ \mathit{len}_f\ \mathit{rear}\ \mathit{len}_r)$, represents these queues where Q is the constructor and *front*, len_f, *rear*, and len_r are field names. The front and rear lists are *front* and *rear*, respectively, and len_f and len_r are the number of elements in each list.

Figure 16 presents this strict queue implementation in pseudocode. Function definitions in the figure use a pattern-matching syntax to decompose lists and queues, where _ is the wildcard pattern. The *enq* function adds an element to a queue, *hd* returns the frontmost element without removing it, and *tl* removes the front element and returns the remaining elements as a new queue. The figure also includes infix list append, ++, and list reverse, *rev*.

When elements are added to *rear* or removed from *front*, a queue maintains the invariant that the size of *front* must be equal to or greater than the size of *rear*. When *rear* is larger than *front*, a *rotation* is performed (figure 16 box), i.e., *rear* is reversed and appended to the end of *front*, and *rear* is reset to empty. The *chk* function checks the queue invariant and performs rotations as needed. Though reversing a list takes time proportional to the size of the list, for any given set of inserted elements, the list containing those elements is only reversed once. Thus a functional queue supports constant amortized time *enq*, *hd*, and *tl* operations.

```
def enq x (Q f len_f r len_r) = chk f len_f $(x::r) (len_r+1)

def chk f len_f r len_r =
  if len_r ⩽ len_f then Q f len_f r len_r
  else Q (f ++ rev r) (len_f + len_r) nil 0

def hd (Q $nil _ _ _) = error
  | hd (Q $(x::f) len_f r len_r) = x
def tl (Q $nil _ _ _) = error
  | tl (Q $(x::f) len_f r len_r) = chk f (len_f−1) r len_r

def (++) $nil lst = lst
  | (++) $(x::xs) lst = $(x::(xs ++ lst))

def rev lst = $(rev/acc lst nil)
def rev/acc $nil acc = acc
  | rev/acc $(x::xs) acc = rev/acc xs $(x::acc)
```

Figure 17: Okasaki's lazy banker's queue.

Rotations are expensive but since the rotated elements are added to the end of *front* and are not needed immediately, laziness might improve the strict queue implementation. Also the constant amortized time bounds of the strict queue do not hold when the queue is used *persistently*. To illustrate this problem, consider a queue with enough elements such that calling *tl* triggers a rotation. Calling *tl* repeatedly on this pre-rotated queue triggers a rotation each time. While the cost of the first rotation is amortized over all the elements, the subsequent rotations require more than constant amortized time because the "savings" are already spent by the first one.

To address these issues, Okasaki adds laziness to the strict queue implementation. The new queue, dubbed the banker's queue,[2] replaces the eager lists with lazy streams and is shown in figure 17. We adopt Okasaki's syntax for laziness, where **$** prefixing an expression delays that expression, while **$** in a pattern means to force the argument before trying to match the pattern.

Since the banker's queue merely adds laziness to the strict queue, it falls into the *basic* kind of lazy data structure described earlier. To see what laziness annotations our profiler suggests, we need a representative benchmark that uses the strict queue:

2 Okasaki uses the banker's method to determine the queue's complexity.

```
def build&sum size tosum = sum (build) (size) tosum
def build n = buildq 0 n
def buildq n n = Q nil 0 nil 0
  | buildq m n = enq m (buildq (m+1) n)
def sum q n = sumq q 0 n
def sumq q n n = 0
  | sumq q m n = (hd) (q) + (sumq (tl) (q) (m+1) n)
```

The *build* function builds a queue of the specified size while *sum* adds up the specified number of elements from the front of the given queue. The *build*&*sum* function combines *build* and *sum*.

Our representative benchmark should benefit from laziness if we sum only the first few elements of the queue, leaving most of the queue unused. Hence, profiling this benchmark should reveal the places where it is profitable to insert laziness annotations. Specifically, profiling *build*&*sum* 1024 50 produces the following laziness suggestions:

```
rev r [ln 5]: 8/10 values used
- delaying 2 unused avoids 3848 subvalues, wgt=2564
xs ++ lst [ln 13]: 645/1013 values used
- delaying 368 unused avoids 2204 subvalues, wgt=1870
```

Figure 18 shows a lazy queue that implements these suggestions. Our profiler-assisted lazy queue has fewer and different laziness annotations than Okasaki's. Specifically, the profiler does not recommend inserting laziness in *enq* and *rev* (boxes 1), leaving *rear* as an eager list.[3] This is justified because *rev* is monolithic, i.e., it always traverses its entire argument regardless of whether it is an eager list or a lazy stream. Instead, our profiler suggests a single delay around the call to *rev* in *chk* (box 2). The profiler also differs in its suggestion to delay the tail of the list returned by append (box 3) while Okasaki delays the entire list. Since the first list element is already a value, this difference is trivial.

Since our profiler recommends uses of laziness different from Okasaki, we further compare the queue implementations with empirical experiments. We timed several calls to *build*&*sum*, varying the number of elements summed. For each run, we used a fixed queue of size 2^{20}:

$$\texttt{for i} \in [0, 2^{20}] : \texttt{time } (\textit{build\&sum}\ 2^{20}\ \texttt{i}) \qquad \text{(BUILD\&SUM)}$$

The graph in figure 19 shows that using an eager rear list in the profiler-assisted lazy queue improves performance over Okasaki's queue, which has to build a suspension for each element.

3 Okasaki observes in a footnote that the rear list could be left as an eager list but, because of his preference of theoretical simplicity over practical optimizations, he presents the version in figure 17.

```
def enq x (Q f lenf r lenr) = chk f lenf [(x::r)]1 (lenr+1)

def chk f lenf rlenr =
  if lenr ⩽ lenf then Q f lenf r lenr
  else Q (f ++ [$(rev r)]2) (lenf + lenr) nil 0

def hd (Q $nil _ _ _) = error
  | hd (Q $(x::f) lenf r lenr) = x
def tl (Q $nil _ _ _) = error
  | tl (Q $(x::f) lenf r lenr) = chk f (lenf−1) r lenr

def (++) $nil lst = lst
  | (++) $(x::xs) lst = x::[$(xs ++ lst)]3

def rev lst = [rev/acc lst nil]1
def rev/acc nil acc = acc
  | rev/acc (x::xs) acc = rev/acc xs [(x::acc)]1
```

Figure 18: Profiler-assisted version of lazy banker's queue.

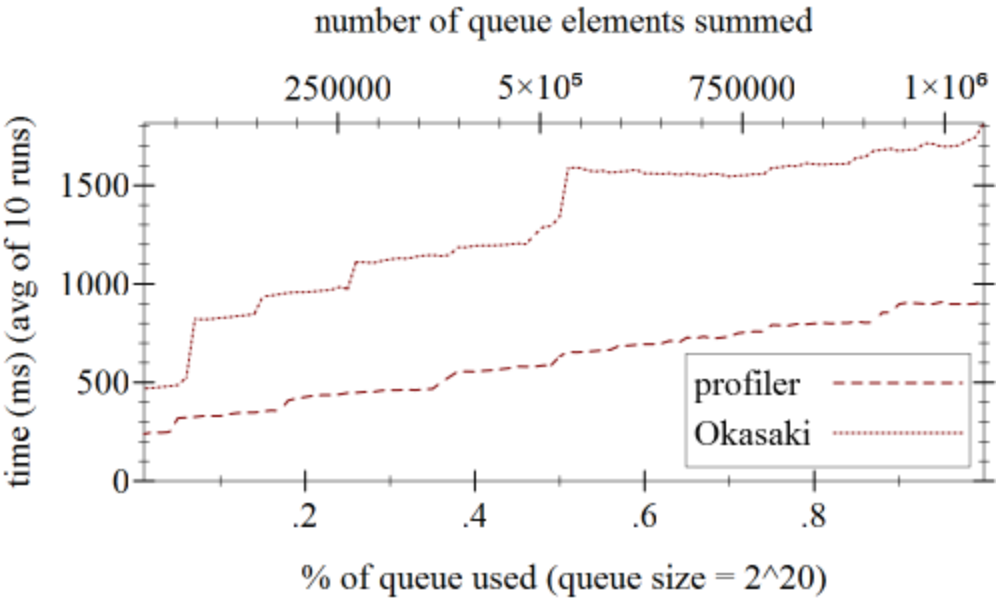

Figure 19: Summing elements of a banker's queue (lower is better).

A problem is that the BUILD&SUM benchmark does not test persistence, because the queue is used in a single-threaded manner. To expose persistence-related issues, a representative benchmark must construct a queue such that a *tl* operation on the queue triggers the rotations suggested by the spikes in the graph of figure 19. For the lazy queues, the rotation operation is delayed, so we must remove a sufficient number of elements before the rotation is forced. Specifically, we build a queue of size $2^n - 1$, right after a delayed rotation is appended to the end of *front*, and then remove $2^{n-1} - 1$ elements from the queue. Removing the next element of the queue forces the rotation. This benchmark uses a *drop* function, which removes some number of elements from a list. We again use $n = 20$ in the persistence benchmark:

```
let q = drop (build (2^20−1)) (2^19−1)                    (PERSIST:LAZY)
in time1 = time (tl q); time2 = time (tl q)
```

Table 5 presents timings for the persistence benchmarks. For reference, we additionally include times for a strict queue, which requires the same time on each access because it repeats the rotation each time. In contrast, the lazy queue times verify that both implementations offer truly constant amortized operations, with the profiler-assisted queue outperforming Okasaki's queue.

implementation	test name	subtest	time (ms)
strict	strict	*time1*	250
strict	strict	*time2*	250
Okasaki	PERSIST:LAZY	*time1*	828
Okasaki	PERSIST:LAZY	*time2*	0
profiler-assisted	PERSIST:LAZY	*time1*	94
profiler-assisted	PERSIST:LAZY	*time2*	0

Table 5: Testing persistence with rotations in the banker's queue.

Though our laziness potential model does not explicitly account for the memoization that enables constant amortized operations for persistently used queues, our profiler is still able to suggest laziness annotations that satisfy the desired behavior, which is a pleasant surprise. This suggests that reasoning with laziness potential possibly suffices to account for both the delaying and the memoization benefits of laziness, which makes intuitive sense because programmers typically do not use laziness for memoization only. This observation warrants further investigation in future work.

9.2.2 *Physicist's Queue*

Okasaki's physicist's queue differs from the banker's queue in three ways. First, it employs a plain list for the rear list instead of a stream. Second, the front list is a delayed plain list instead of a stream. Third, the content of the front list is cached into a separate plain list before a rotation, speeding up *hd* accesses. Thus the physicist's queue requires an extra field in its data definition: $(\mathrm{Q_p}\ front_{cache}\ front\ len_f\ rear\ len_r)$. Figure 20 shows the implementation for Okasaki's physicist's queue.

Deriving the physicist's queue from the strict queue (in figure 16) involves more than just adding lazy annotations, so it belongs in the *insightful* category of data structure. Nevertheless, we follow the steps described at the beginning of the chapter to develop the alternative implementation in figure 21. It turns out that laziness potential cannot replicate Okasaki's caching strategy, and the profiler suggests turning *front* and $front_{cache}$ into duplicate lazy streams. The other profiler suggestions are identical to the profiled *banker's* queue: the rear list remains an eager list and is delayed with a single delay in *chk*.

Figure 22 shows physicist's queue timings for BUILD&SUM, the benchmark used in section 9.2.1. With its caching, Okasaki's physicist's queue is faster than the profiler-assisted implementation, until half the queue is used. After that point, the benefits of caching in the Okasaki's queue are nullified because the entire *front* list must be forced anyways. Also, forcing this last part of *front* takes much longer due to all the extra suspensions created by *tl*, which manifests as a large spike in the graph. Essentially, our profiler-assisted queue forces a little of *front* with each *tl* call, while Okasaki's version delays all the forcings until the second half of the queue is consumed. Deciding which queue implementation is better depends on the expected use cases.

9.2.3 *Implicit Queues*

Implicit queue and deque data structures pose challenging obstacles for both designers and profilers. Hinze and Paterson's finger trees [36] are a related data structure and pose the same problems. This section presents details concerning queues; the cases for deques and finger trees are similar; see table 4.

Implicit queues rely on implicit recursive slowdown [62, Ch. 11], a technique inspired by functional representations of Peano numerals. Thus implicit queues are also an *insightful* kind of data structure. Like explicit queues, implicit queues offer constant amortized time *hd*, *tl*, and *enq* operations. Essentially, the queue maintains some front elements and some rear elements

```
def enq x (Qp f' f len_f r len_r) = chk f' f len_f (x::r) (len_r+1)

def chk f' f len_f r len_r =
  if len_r ⩽ len_f then chkw f' f len_f r len_r
  else let f'' = force f
       in chkw f'' $(f'' ++ rev r) (len_f + len_r) nil 0

def chkw nil f len_f r len_r = (Qp (force f) f len_f r len_r)
  | chkw f' f len_f r len_r = (Qp f' f len_f r len_r)

def hd (Qp nil _ _ _ _) = error
  | hd (Qp (x::f') f len_f r len_r) = x
def tl (Qp nil _ _ _ _) = error
  | tl (Qp (x::f') f len_f r len_r) =
    chkw f' $(rest (force f)) (len_f−1) r len_r

def (++) nil lst = lst
  | (++) (x::xs) lst = x::(xs ++ lst)

def rev lst = rev/acc lst nil
def rev/acc nil acc = acc
  | rev/acc (x::xs) acc = rev/acc xs (x::acc)
```

Figure 20: Okasaki's lazy physicists's queue.

```
def enq x (Qp f' f len_f r len_r) = chk f' f len_f (x::r) (len_r+1)

def chk f' f len_f r len_r =
  if len_r ⩽ len_f then chkw f' f len_f r len_r
  else chkw f (f ++ $(rev r)) (len_f + len_r) nil 0

def chkw nil f len_f r len_r = (Qp f f len_f r len_r)
  | chkw f' f len_f r len_r = (Qp f' f len_f r len_r)

def hd (Qp $nil _ _ _ _) = error
  | hd (Qp $(x::f') f len_f r len_r) = x
def tl (Qp $nil _ _ _ _) = error
  | tl (Qp $(x::f') $(y::f) len_f r len_r) =
    chkw f' f (len_f−1) r len_r

def (++) $nil lst = lst
  | (++) $(x::xs) lst = x::$(xs ++ lst)

def rev lst = rev/acc lst nil
def rev/acc nil acc = acc
  | rev/acc (x::xs) acc = rev/acc xs (x::acc)
```

Figure 21: Profiler-assisted lazy physicists's queue.

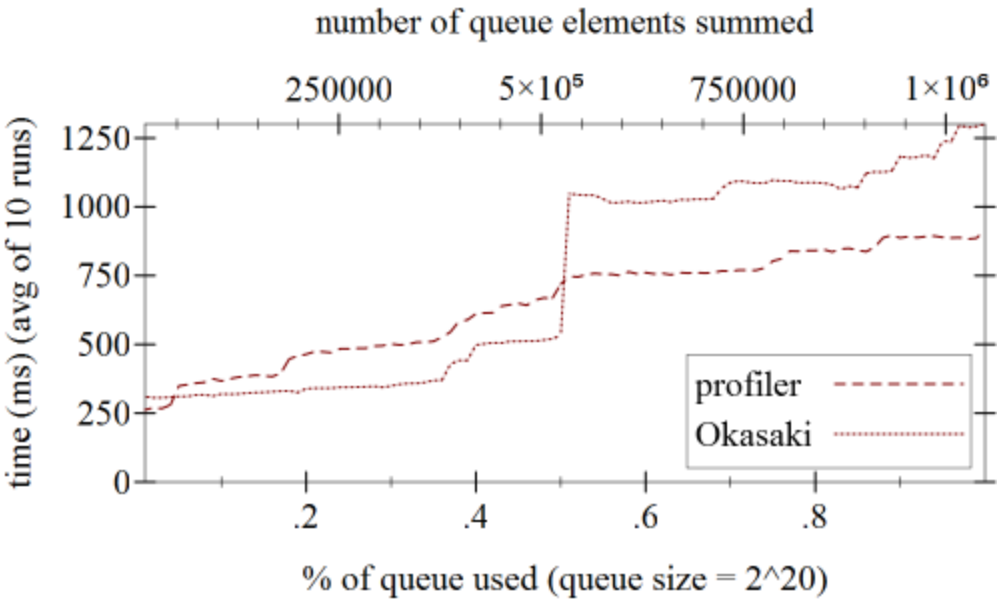

Figure 22: Summing physicist's queue elements (lower is better).

in two "digit" structures, with the remaining elements residing in a delayed middle queue. Here are the record definitions for implicit queues.

A *digit* is a (*Zero*) or (*One* x) or (*Two* x y)
A *queue* is a (*Shallow digit*) or (*Deep* $digit_f$ *midQ* $digit_r$)

A "digit" is either a *Zero*, *One*, or *Two* structure and an implicit queue itself is represented with either a *Shallow* or a *Deep* structure. A *Shallow* queue is comprised of a single digit while a *Deep* queue has two digits plus a delayed middle queue of element *pairs*.

Figure 23 presents *enq*, *tl*, and *hd* for implicit queues. The *enq* function shows the progression of a queue's internal state as more elements are added, with the digits transitioning from *Zero* to *Two*, and the queue itself from *Shallow* to *Deep*. These transitions are reversed in *tl*. Finally, the *hd* function returns the front element by destructuring a queue in a straightforward manner.

While the design of this data structure leverages laziness to support constant amortized operations, it also magnifies the overhead of laziness because every operation must destructure not only the queue, but many fields of the queue as well. These numerous destructuring operations result in a high rate of suspension creation and forcing. In fact, profiling the BUILD&SUM benchmark only finds expressions with low laziness potential, even for queues where most or all of the elements are unused, suggesting that the data struc-

```
def enq (Shallow (Zero)) x = Shallow (One x)
  | enq (Shallow (One x)) y =
      Deep (Two x y) nil (Zero)
  | enq (Deep f m (Zero)) x = Deep f m (One x)
  | enq (Deep f m (One x)) y =
      Deep f $(enq (force m) (x, y)) (Zero)

def hd (Shallow (Zero)) = error
  | hd (Shallow (One x)) = x
  | hd (Deep (One x) m r) = x
  | hd (Deep (Two x y) m r) = x

def tl (Shallow (Zero)) = error
  | tl (Shallow (One x)) = Shallow (Zero)
  | tl (Deep (One x) $(Shallow (Zero)) r) = Shallow r
  | tl (Deep (One x) $m r) =
      let (y, z) = hd m in Deep (Two y z) $(tl m) r
  | tl (Deep (Two x y) m r) = Deep (One y) m r
```

Figure 23: Implicit queue operations.

ture should not use any laziness. This makes intuitive sense since each operation effectively "uses" the entire queue.

To investigate this discrepancy, figure 24 presents BUILD&SUM timing information for a strict queue compared to Okasaki's queue. The chart shows that the high rate of suspension creation and forcing negates any benefits that laziness might provide, no matter how much of the queue is used. Obviously, the laziness overhead is implementation dependent. We implemented suspensions with a lambda and a mutable box. Although direct compiler support for suspensions would likely improve performance of the lazy queue and decrease the gap, the discrepancy is large enough that an implementer should consider omitting laziness.

While BUILD&SUM does not use the queue persistently, the worst-case "expensive" operation for implicit queues is only logarithmic in the size of the queue (as opposed to the linear rotations of the banker's queue). Thus, even when we tested the queues with persistent-usage benchmarks, the strict version still outperformed the lazy one. In conclusion, this experiment further supports omitting laziness from an implicit queue implementation even if it may spoil its theoretical properties.

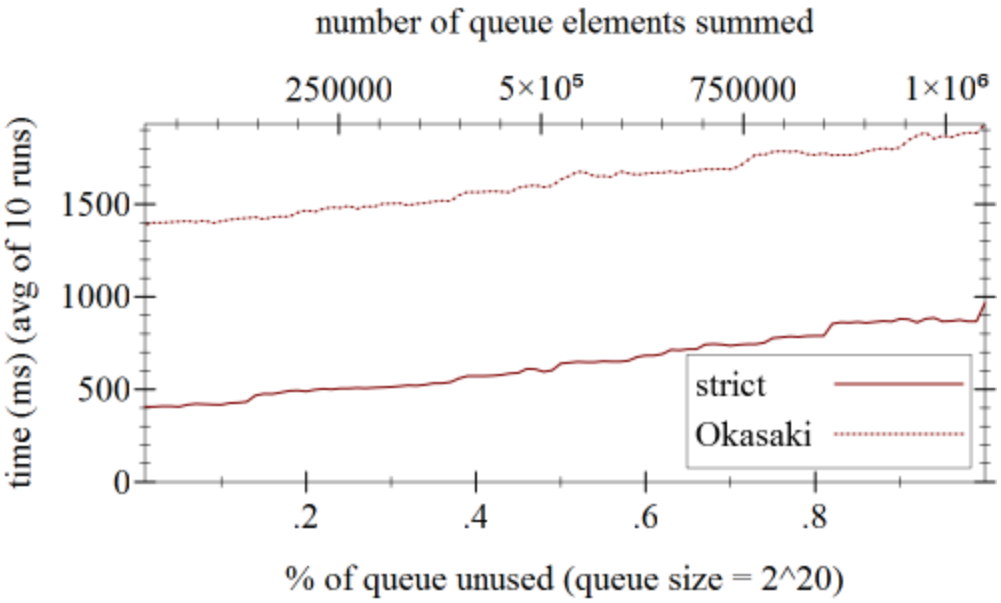

Figure 24: Summing elements of a implicit queue (lower is better).

9.3 MONADIC PARSER COMBINATORS

Our second suite of benchmarks concerns Parsec [48], a monadic parser combinator library. Leijen and Meijer point to one spot in the monadic bind operator where laziness is key, calling it "essential for efficient behavior." Since the library is a higher-order program, we instantiated it for five different scenarios to apply our profiler: a CSV parser, a URL query parser, a JSON parser, and an HTTP request parser.[4] In every case, the profiler suggested adding laziness at the exact spot identified by the Parsec paper.

9.4 MANIPULATING GAME TREES

Our final benchmark is an AI game algorithm from Land of Lisp [8], which introduces programmers to functional Lisp programming via a series of games. One of these games, Dice of Doom, is a turn-based strategy game similar to Risk. After several chapters, a diligent student is rewarded with a full-fledged game, complete with graphical interface and AI players. An AI player generates a game tree of all possible moves to help it determine the next move and utilizes laziness to manage the tree. Without laziness, the AI player analyzes only a small version of the game. Utilizing laziness, however, the AI player scales to a realistic-size game.

4 We followed *Real World Haskell* [63] for guidance.

The game implementation (some 1,500 lines of code) constitutes a broad-spectrum benchmark for our profiler, not only because it requires laziness, but also because it uses many additional language features such as mutable state, list comprehensions, and GUI libraries. Our profiler is able to instrument the game and collects data as the game is played, returning suggestions when the game concludes. It suggests enough laziness so that playing the full game is possible but with fewer annotations than the Land of Lisp version.

Part VI

CONCLUSION AND FUTURE WORK

10 CONCLUSION

Maximizing the benefits of lazy evaluation while minimizing its overhead costs requires specific experience and extensive expertise such that the majority of programmers are ill-equipped to understand how laziness affects their programs. Thus languages that expose linguistic constructs for lazy programming without providing guidance on how to use them put programmers at risk of introducing subtle bugs into their programs.

In support of my thesis, that designing lazy algorithms and data structures in strict languages benefits from tool support, I have developed two techniques, one static and one dynamic. I have also implemented prototype tools that utilize these analyses in order to help the programmer properly leverage laziness in their programs. Finally, I have demonstrated the effectiveness of my tools on a variety of real-world programs.

FUTURE WORK

We see several directions for future work.

- Our work on a static analysis would benefit from developing a modular analysis. Our techniques require the whole program and is thus less straightforward to use in the presence of libraries.
- Using laziness in typed languages raises additional complications since types require additional program constraints that may conflict with laziness annotations. Thus typed languages may require extraneous laziness in certain cases (see Okasaki). Porting our work to a typed setting would be an interesting and non-trivial endeavor. We conjecture that expressing strictness information via types may also provide a way to enable a modular laziness-by-need analysis.
- There may be ways to improve our laziness potential metric from part v. For example, when approximating the cost of delaying an expression, the metric only considers unneeded values but does not account for the cost of suspending and then forcing the needed values. Also, we assume that creating a thunk has uniform cost but each thunk may actually incur varying costs due to differences in the captured environment. It's not clear, however, if any such improvements would affect the overall quality of the profiler's suggestions.

- Finally, though we presented our work for a by-value λ-calculus and implemented our tools for a matching language, the idea should be applicable to any language that supports both strict and lazy evaluation. Specifically, our insights may transfer to a lazy language with strictness annotations, such as Haskell. While the maturity of Haskell's strictness analysis already assists programmers with the task of eliminating unnecessary laziness, the approach is intrinsically limited by its static, approximate nature. Hence, books for real-world programmers [63] suggest the insertion of strictness annotations to help the compiler. We conjecture that a *dynamic* profiler, based on the notion of laziness potential, would assist these programmers with the difficult task of locating appropriate positions for these annotations.

APPENDIX

A

THE CALL-BY-NEED LAMBDA CALCULUS, REVISITED

A.1 A SHORT HISTORY OF THE λ-CALCULUS

Starting in the late 1950s, programming language researchers began to look to Church's λ-calculus [15] for inspiration. Some used it as an analytic tool to understand the syntax and semantics of programming languages, while others exploited it as the basis for new languages. By 1970, however, a disconnect had emerged in the form of call-by-value programming, distinct from the notion of β and normalization in Church's original calculus. Plotkin [65] reconciled the λ-calculus and Landin's SECD machine for the ISWIM language [46] with the introduction of a notion of correspondence and with a proof that two distinct variants of the λ-calculus corresponded to two distinct variants of the ISWIM programming language: one for call-by-value and one for call-by-name.

In the early 1970s, researchers proposed call-by-need [30, 34, 81], a third kind of parameter passing mechanism that could be viewed as yet another variant of the ISWIM language. Call-by-need is supposed to represent the best of both worlds. While call-by-value ISWIM always evaluates the argument of a function, the call-by-name variant evaluates the argument every time it is needed. Hence, if an argument (or some portion) is never needed, call-by-name wins; otherwise call-by-value is superior because it avoids re-evaluation of arguments. Call-by-need initially proceeds like call-by-name, evaluating a function's body before the argument—until the value of the argument is needed; at that point, the argument is evaluated and the resulting value is used from then onward. In short, call-by-need evaluates an argument at most once, and only if needed.

Since then, researchers have explored a number of characterizations of call-by-need [19, 31, 32, 43, 58, 64, 67]. Concerning this appendix, three stand out. Launchbury's semantics [47] specifies the meaning of complete programs with a Kahn-style natural semantics. The call-by-need λ-calculi of Ariola and Felleisen [4, 5, 6], and of Maraist, Odersky, and Wadler [6, 52, 54] are equational logics in the spirit of the λ-calculus.

The appeal of the λ-calculus has several reasons. First, a calculus is sound with respect to the observational (behavioral) equivalence relation [56]. It can therefore serve as the starting point for other, even more powerful logics. Second, its axioms are rich enough to mimic machine evaluation, meaning programmers can reduce programs to values without thinking about implementation details. Finally, the λ-calculus gives rise to a substantial metatheory [7, 18] from which researchers have generated useful and practical results for its cousins.

Unfortunately, neither of the existing by-need calculi model lazy evaluation in a way that matches lazy language implementations. Both calculi suffer from the same two problems. First, unlike the by-name and by-value calculi,

the by-need calculi never discard function calls, even after the call is resolved and the argument is no longer needed. Lazy evaluation does require some accumulation of function calls due to the delayed evaluation of arguments but the existing calculi adopt the extreme solution of retaining every call. Indeed, the creators of the existing calculi acknowledge that a solution to this problem would strengthen their work but they could not figure out a proper solution.

Second, the calculi include re-association axioms even though these axioms have no counterpart in any implementation. The axioms are mere administrative steps, needed to construct β-like redexes. Hence, they should definitely not be considered computationally on par with other axioms.

In this appendix, we provide an alternative equational axiomatization of call-by-need that eliminates both problems. Based on a single axiom, it avoids the retention of function calls and eliminates the extraneous re-association axioms. The single axiom uses a grammar of contexts to describe the exact notion of a *needed computation*. Like its predecessors, our new calculus satisfies consistency and standardization properties and is thus suitable for reasoning about behavioral equivalence. In addition, we establish an intensional correspondence with Launchbury's semantics.

The second section of this appendix recalls the two existing by-need calculi in some detail. The third section presents our new calculus, as well as a way to derive it from Ariola and Felleisen's calculus. Sections A.4 and A.5 show that our calculus satisfies the usual meta-theorems and that it is correct with respect to Launchbury's semantics. Finally, we discuss some possible extensions.[1]

1 The material for this Appendix appeared previously in Chang and Felleisen [2012]: *The Call-by-need Lambda Calculus, Revisited*, In: ESOP '12: Proceedings of the 21st European Symposium on Programming.

A.2 THE ORIGINAL CALL-BY-NEED λ-CALCULI

The original call-by-need λ-calculi are independently due to two different groups—Ariola and Felleisen [4, 5] and Maraist, et al. [52, 54]—and were jointly presented at POPL in 1995 [6]. Both calculi use the standard set of terms as syntax:

$$e = x \mid \lambda x.e \mid e\ e \qquad \text{(Terms)}$$

Our treatment of syntax employs the usual conventions, including Barendregt's standard hygiene condition for variable bindings [7]. Figure 25 specifies the calculus of Maraist et al., λ_{mow}, and λ_{af}, Ariola and Felleisen's variant. Nonterminals in some grammar productions have subscript tags to differentiate them from similar sets elsewhere in the paper. Unsubscripted definitions have the same denotation in all systems.

$$\begin{aligned} v_m &= x \mid \lambda x.e \\ C &= [\,] \mid \lambda x.C \mid C\ e \mid e\ C \end{aligned}$$

$$\begin{aligned} (\lambda x.C[x])\ v_m &= (\lambda x.C[v_m])\ v_m && (\mathcal{V}) \\ (\lambda x.e_1)\ e_2\ e_3 &= (\lambda x.e_1\ e_3)\ e_2 && (\mathcal{C}) \\ (\lambda x.e_1)((\lambda y.e_2)\ e_3) &= (\lambda y.(\lambda x.e_1)\ e_2)\ e_3 && (\mathcal{A}) \\ (\lambda x.e_1)\ e_2 &= e_1, x \notin fv(e_1) && (\mathcal{G}) \end{aligned}$$

$$\begin{aligned} v &= \lambda x.e \\ a_{af} &= v \mid (\lambda x.a_{af})\ e \\ E_{af} &= [\,] \mid E_{af}\ e \mid (\lambda x.E_{af})\ e \mid (\lambda x.E_{af}[x])\ E_{af} \end{aligned}$$

$$\begin{aligned} (\lambda x.E_{af}[x])\ v &= (\lambda x.E_{af}[v])\ v && (\textit{deref}) \\ (\lambda x.a_{af})\ e_1\ e_2 &= (\lambda x.a_{af}\ e_2)\ e_1 && (\textit{lift}) \\ (\lambda x.E_{af}[x])\ ((\lambda y.a_{af})\ e) &= (\lambda y.(\lambda x.E_{af}[x])\ a_{af})\ e && (\textit{assoc}) \end{aligned}$$

Figure 25: Existing call-by-need λ-calculi (top: λ_{mow}, bottom: λ_{af})

In both calculi, the analog to the β axiom—also called a *basic notion of reduction* [7]—replaces variable occurrences, one at a time, with the value of the function's argument. Value substitution means that there is no duplication of work as far as argument evaluation is concerned. The function call is retained because additional variable occurrences in the function body may need the

argument. Since function calls may accumulate, the calculi come with axioms that re-associate bindings to pair up functions with their arguments. For example, re-associating $(\lambda x.(\lambda y.\lambda z.z)\ v_y)\ v_x\ v_z$ in λ_{af} exposes a *deref* redex:

$$\begin{aligned}(\lambda x.(\lambda y.\underline{\lambda z.z})\ v_y)\ v_x\ \underline{v_z} &\xrightarrow{lift} (\lambda x.(\lambda y.\underline{\lambda z.z})\ v_y\ \underline{v_z})\ v_x \\ &\xrightarrow{lift} (\lambda x.(\lambda y.\underline{(\lambda z.z)\ v_z})\ v_y)\ v_x\end{aligned}$$

The two calculi differ from each other in their timing of variable replacements. The λ_{mow} calculus allows the replacement of a variable with its value anywhere in the body of its binding λ. The λ_{af} calculus replaces a variable with its argument only if evaluation of the function body needs it, where "need" is formalized via so-called evaluation contexts (E_{af}). Thus evaluation contexts in λ_{af} serve the double purpose of specifying demand for arguments and the standard reduction strategy. The term $(\lambda x.\lambda y.x)\ v$ illustrates this difference between the two calculi. According to λ_{mow}, the term is a $\mathcal{V}$ redex and reduces to $(\lambda x.\lambda y.v)\ v$, whereas in λ_{af}, the term is irreducible because the x occurs in an inner, unapplied λ, and is thus not "needed."

Also, λ_{mow} is more lenient than λ_{af} when it comes to re-associations. The λ_{af} calculus re-associates the left or right hand side of an application only if it has been completely reduced to an answer, but λ_{mow} permits re-association as soon as one nested function layer is revealed. In short, λ_{mow} proves more equations than λ_{af}, i.e., $\boldsymbol{\lambda_{af}} \subset \boldsymbol{\lambda_{mow}}$.

In λ_{af}, programs reduce to answers:

$$\texttt{eval}_{af}(e) = \texttt{done} \text{ iff there exists an answer } a_{af} \text{ such that } \boldsymbol{\lambda_{af}} \vdash e = a_{af}$$

In contrast, Maraist et al. introduce a "garbage collection" axiom into λ_{mow} to avoid answers and to use values instead. This suggests the following definition:

$$\texttt{eval}_{mow}(e) = \texttt{done} \text{ iff there exists a value } v_m \text{ such that } \boldsymbol{\lambda_{mow}} \vdash e = v_m$$

This turns out to be incorrect, however. Specifically, let $\texttt{eval}_{name}$ be the analogous call-by-name evaluator. Then $\texttt{eval}_{af} = \texttt{eval}_{name}$ but $\texttt{eval}_{mow} \neq \texttt{eval}_{name}$. Examples such as $(\lambda x.\lambda y.x)\ \Omega$ confirm the difference.

In recognition of this serious failure, Maraist et al. use Ariola and Felleisen's axioms and evaluation contexts to create their Curry-Feys-style standard reduction sequences. Doing so reveals the inconsistency of λ_{mow} with respect to Plotkin's *correspondence criteria* [65]. According to Plotkin, a useful calculus *corresponds to* a programming language, meaning its axioms (1) satisfy the Church-Rosser and Curry-Feys Standardization properties, and (2) define a standard reduction function that is equal to the evaluation function

of the programming language. Both the call-by-name and the call-by-value λ-calculi satisfy these criteria with respect to call-by-name and call-by-value SECD machines for ISWIM, respectively. So does λ_{af} with respect to a call-by-need SECD machine, but some of λ_{mow}'s axioms cannot be used as standard reduction relations.

Finally, the inclusion of $\mathcal{G}$ is a brute-force attempt to address the function call retention problem. Because $\mathcal{G}$ may discard arguments even before the function is called, both sets of authors consider it too coarse and acknowledge that a tighter solution to the function call retention issue would "strengthen the calculus and its utility for reasoning about the implementations of lazy languages" [6].

A.3 A NEW CALL-BY-NEED λ-CALCULUS

Our new calculus, λ_{need}, uses a single axiom, β_{need}. The new axiom evaluates the argument when it is first demanded, replaces all variable occurrences with that result, and then discards the argument and thus the function call. In addition, the axiom performs the required administrative scope adjustments as part of the same step, rendering explicit re-association axioms unnecessary. In short, every reduction step in our calculus represents computational progress.

Informally, to perform a reduction, three components must be identified:

1. the next variable in demand (*var*),
2. the function that binds that demanded variable (*fn*),
3. and the argument to that function (*arg*).

In previous by-need calculi the re-association axioms rewrite a term so that the binding function and its argument are adjacent.

Without the re-association axioms, finding the function that binds the demanded variable and its argument requires a different kind of work. The following terms show how the demanded variable, its binding function, and its argument can appear at seemingly arbitrary locations in a program:

- $(\underset{fn}{\lambda x.}\ (\lambda y.\lambda z.\ \underset{var}{x})\ e_y)\ \underset{arg}{e_x}\ e_z$
- $(\lambda x.(\underset{fn}{\lambda y.}\ \lambda z.\ \underset{var}{y})\ \underset{arg}{e_y})\ e_x\ e_z$
- $(\lambda x.(\lambda y.\ \underset{fn}{\lambda z.}\ \underset{var}{z})\ e_y)\ e_x\ \underset{arg}{e_z}$

Our β_{need} axiom employs a grammar of contexts to describe the path from a demanded variable to its binding function and from there to its argument.

The first subsection explains the syntax and the contexts of λ_{need} in a gradual fashion. The second subsection presents the β_{need} axiom and shows how to derive it from Ariola and Felleisen's λ_{af} calculus.

A.3.1 *Contexts*

Like the existing by-need calculi, the syntax of our calculus is that of Church's original calculus. In λ_{need}, calculations evaluate terms e to answers $A[v]$, which generalize answers from Ariola and Felleisen's calculus:

$$e = x \mid \lambda x.e \mid e\ e \qquad \text{(Terms)}$$
$$v = \lambda x.e \qquad \text{(Values)}$$
$$a = A[v] \qquad \text{(Answers)}$$
$$A = [\,] \mid A[\lambda x.A]\ e \qquad \text{(Answer Contexts)}$$

Following Ariola and Felleisen, the basic axiom uses evaluation contexts to specify the notion of demand for variables:

$$E = [\,] \mid E\ e \mid \ldots \qquad \text{(Evaluation Contexts)}$$

The first two kinds, taken from λ_{af}, specify that a variable is in demand, and that a variable in the operator position of an application is in demand, respectively.

Since the calculus is to model program evaluation, we are primarily interested in demanded variables under a λ-abstraction. This kind of evaluation context is defined using an answer context A:

$$E = \ldots \mid A[E] \mid \ldots \qquad \text{(Another Evaluation Context)}$$

Using answer contexts, this third evaluation context dictates that demand exists under a λ if a corresponding argument exists for that λ. Note how *an answer context descends under the same number of* λ*s as arguments for those* λ*s.* In particular, for any term $A[\lambda x.e_1]\ e_2$, e_2 is always the argument of $\lambda x.e_1$. The third evaluation context thus generalizes the function-is-next-to-argument requirement found in both call-by-name and call-by-value. The generalization is needed due to the retention of function calls in λ_{need}.

Here are some example answer contexts:

$$A_0 = \underbrace{(\lambda x.[\,])\ e_x}$$
$$A_1 = \underbrace{(\lambda x.\ \underbrace{(\lambda y.[\,])\ e_y})\ e_x}$$
$$A_2 = \underbrace{(\lambda x.\ \underbrace{(\lambda y.\ \underbrace{(\lambda z.[\,])\ e_z})\ e_y})\ e_x}$$

An underbrace matches each function to its argument. The examples all juxtapose functions and their arguments. In contrast, the next two separate functions from their arguments:

$$A_3 = (\underbrace{\lambda x.\lambda y.\lambda z.[\,]}\ e_x\ e_y\ e_z$$

$$A_4 = (\lambda x.(\underbrace{\lambda y.\lambda z.[\,]}\ e_y)\ e_x\ e_z$$

To summarize thus far, when an in demand variable is discovered under a λ, the surrounding context looks like this:

$$A[E[x]]$$

where both the function binding x and its argument are in A. The decomposition of the surrounding context into A and E assumes that A encompasses as many function-argument pairs as possible; in other words, it is impossible to merge the outer part of E with A to form a larger answer context.

To know which argument corresponds to the demanded variable, we must find the λ that binds x in A. To this end, we split answer contexts so that we can "highlight" a function-argument pair within the context:

$$\hat{A} = [\,] \mid A[\hat{A}]\ e \qquad \text{(Partial Answer Contexts–Outer)}$$

$$\check{A} = [\,] \mid A[\lambda x.\check{A}] \qquad \text{(Partial Answer Contexts–Inner)}$$

Using these additional contexts, any answer context can be decomposed into

$$\hat{A}[A[\lambda x.\check{A}[\ ]]\ e]$$

where e is the argument of $\lambda x.\check{A}[\ ]$. For a fixed function-argument pair in an answer context, this partitioning into $\hat{A}$, A, and $\check{A}$ is unique. The $\hat{A}$ subcontext represents the part of the answer context around the chosen function-argument pair; the $\check{A}$ subcontext represents the part of the answer context in its body; and A here is the subcontext between the function and its argument. Naturally we must demand that $\hat{A}$ composed with $\check{A}$ is an answer context as well so that the overall context remains an answer context. The following table lists the various subcontexts for the example A_4 for various function-argument pairs:

	$A_4 = (\lambda x.(\lambda y.\lambda z.[\,])\ e_y)\ e_x\ e_z$		
var in demand	x	y	z
$\hat{A} =$	$[\,]\ e_z$	$(\lambda x.[\,])\ e_x\ e_z$	$[\,]$
$A =$	$[\,]$	$[\,]$	$(\lambda x.(\lambda y.[\,])\ e_y)\ e_x$
$\check{A} =$	$(\lambda y.\lambda z.[\,])\ e_y$	$\lambda z.[\,]$	$[\,]$
$A_4 =$	$\hat{A}[A[\lambda x.\check{A}]\ e_x]$	$\hat{A}[A[\lambda y.\check{A}]\ e_y]$	$\hat{A}[A[\lambda z.\check{A}]\ e_z]$

Now we can define the fourth kind of evaluation context:

(Final Eval. Context)

$$E = \ldots \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]\ E], \quad \text{where } \hat{A}[\check{A}] \in A$$

This final evaluation context shows how demand shifts to an argument when a function parameter is in demand within the function body.

A.3.2 *The β_{need} Axiom and a Derivation*

Figure 26 summarizes the syntax of λ_{need} as developed in the preceding section.[2] In this section we use these definitions to formulate the β axiom for our calculus.

$$e = x \mid \lambda x.e \mid e\ e \qquad \text{(Terms)}$$
$$v = \lambda x.e \qquad \text{(Values)}$$
$$a = A[v] \qquad \text{(Answers)}$$
$$A = [\,] \mid A[\lambda x.A]\ e \qquad \text{(Answer Contexts)}$$
$$\hat{A} = [\,] \mid A[\hat{A}]\ e \qquad \text{(Partial Answer Contexts–Outer)}$$
$$\check{A} = [\,] \mid A[\lambda x.\check{A}] \qquad \text{(Partial Answer Contexts–Inner)}$$
$$E = [\,] \mid E\ e \mid A[E] \mid \hat{A}[A[\lambda x.\check{A}[E[x]]]\ E], \quad \text{where } \hat{A}[\check{A}] \in A \qquad \text{(Evaluation Contexts)}$$

Figure 26: The syntax and contexts of the our new λ_{need} calculus.

2 We gratefully acknowledge Casey Klein's help with the A production.

Here is the single axiom of λ_{need}:

$$\hat{A}[A_1[\lambda x.\check{A}[E[x]]]\ A_2[v]] = \hat{A}[A_1[A_2[\check{A}[E[x]]\{x ::= v\}]]], \qquad (\beta_{need})$$
$$\text{where } \hat{A}[\check{A}] \in A$$

A β_{need} redex determines which parameter x of some function is in demand and how to locate the corresponding argument $A_2[v]$, which might be an answer not necessarily a value. The contexts from the previous section specify the path from the binding position (λ) to the variable occurrence and the argument. A β_{need} reduction substitutes the value in $A_2[v]$ for all free occurrences of the function parameter—just like in the conventional by-name and by-value λ-calculi. In the process, the function call is discarded. Since the argument has been reduced to a value, there is no duplication of work, meaning our calculus satisfies the requirements of lazy evaluation. Lifting A_2 to the top of the evaluation context ensures that its bindings remain intact and visible for v.

Here is a sample reduction in λ_{need}, where $\longrightarrow$ is the one-step reduction:

$$\begin{aligned}
& ((\lambda x.(\lambda y.\underline{\lambda z.\mathbf{z}\ y\ x})\ \lambda y.y)\ \lambda x.x)\ \underline{\lambda z.z} && (1)\\
\longrightarrow\ & (\lambda x.(\underline{\lambda y.(\lambda z.\underline{z})\ \mathbf{y}\ x})\ \underline{\lambda y.y})\ \lambda x.x && (2)\\
\longrightarrow\ & (\lambda x.((\underline{\lambda z.\mathbf{z}})\ \underline{\lambda y.y})\ x)\ \lambda x.x && (3)\\
\longrightarrow\ & (\underline{\lambda x.(\lambda y.\underline{y})\ \mathbf{x}})\ \underline{\lambda x.x} && (4)
\end{aligned}$$

The variable in demand is in bold; its binding λ and argument are underlined. Line 1 is an example of a reduction that involves a non-adjoined function and argument pair. In line 2, the demand for the value of z (twice underlined) triggers a demand for the value of y; line 4 contains a similar demand chain.

$$\begin{aligned}
v &= \lambda x.e && \text{(Values)}\\
a_{af} &= A_{af}[v] && \text{(Answers)}\\
A_{af} &= [\,] \mid (\lambda x.A_{af})\ e && \text{(Answer Contexts)}\\
E_{af} &= [\,] \mid E_{af}\ e \mid A_{af}[E_{af}] \mid (\lambda x.E_{af}[x])\ E_{af} && \text{(Evaluation Contexts)}
\end{aligned}$$

$$\begin{aligned}
(\lambda x.E_{af}[x])\ v &= E_{af}[x]\{x ::= v\} && (\beta'_{need})\\
(\lambda x.A_{af}[v])\ e_1\ e_2 &= (\lambda x.A_{af}[v\ e_2])\ e_1 && (\mathit{lift}')\\
(\lambda x.E_{af}[x])\ ((\lambda y.A_{af}[v])\ e) &= (\lambda y.A_{af}[(\lambda x.E_{af}[x])\ v])\ e && (\mathit{assoc}')
\end{aligned}$$

Figure 27: A modified calculus, $\lambda_{af\text{-}mod}$.

To furnish additional intuition into β_{need}, we use the rest of the section to derive it from the axioms of λ_{af}. The $\lambda_{af\text{-}mod}$ calculus in figure 27 combines

λ_{af} with two insights. First, Garcia et al. [32] observed that when the answers in λ_{af}'s *lift* and *assoc* redexes are nested deeply, multiple re-associations are performed consecutively. Thus we modify *lift* and *assoc* to perform all these re-associations in one step.[3] The modified calculus defines answers via answer contexts, A_{af}, and the modified *lift'* and *assoc'* axioms utilize these answer contexts to do the multi-step re-associations. Thus programs in this modified calculus reduce to answers $A_{af}[v]$. Also, the A_{af} answer contexts are identical to the third kind of evaluation context in $\lambda_{af\text{-}mod}$ and the new definition of E_{af} reflects this relationship.

Second, Maraist et al. [53] observed that once an argument is reduced to a value, all substitutions can be performed at once. The β'_{need} axiom exploits this idea and performs a full substitution. Obviously β'_{need} occasionally performs more substitutions than *deref*. Nevertheless, any term with an answer in λ_{af} likewise has an answer when reducing with β'_{need}.

Next an inspection of the axioms shows that the contractum of a *assoc'* redex contains a β'_{need} redex. Thus the *assoc'* re-associations and β'_{need} substitutions can be performed with one merged axiom:[4]

$$(\lambda x.E_{af}[x])\ A_{af}[v] = A_{af}[E_{af}[x]\{x ::= v\}] \qquad (\beta''_{need})$$

The final step is to merge *lift'* with β''_{need}, which requires our generalized answer and evaluation contexts. A naïve attempt may look like this:

$$A_1[\lambda x.E[x]]\ A_2[v] = A_1[A_2[E[x]\{x ::= v\}]] \qquad (\beta'''_{need})$$

As the examples in the preceding subsection show, however, the binding occurrence for the "in demand" parameter x may not be the inner-most binding λ once the re-association axioms are eliminated. That is, in comparison with β_{need}, β'''_{need} incorrectly assumes E is always next to the binder. We solve this final problem with the introduction of partial answer contexts.

3 The same modifications cannot be applied to $\mathcal{C}$ and $\mathcal{A}$ in λ_{mow} because they allow earlier re-association and thus not all the re-associations are performed consecutively.

4 Danvy et al. [19] dub a β''_{need} redex a "potential redex" in unrelated work.

A.4 CONSISTENCY, DETERMINISM, AND SOUNDNESS

If a calculus is to model a programming language, it must satisfy some essential properties, most importantly a Church-Rosser theorem and a Curry-Feys standardization theorem [65]. The former guarantees consistency of evaluation; that is, we can define an evaluator *function* with the calculus. The latter implies that the calculus comes with a *deterministic* evaluation strategy. Jointly these properties imply the calculus is *sound* with respect to observational equivalence.

A.4.1 *Consistency: Church-Rosser*

The λ_{need} calculus defines an evaluator for a by-need language:

$$\texttt{eval}_{need}(e) = \texttt{done} \text{ iff there exists an answer } a \text{ such that } \boldsymbol{\lambda_{need}} \vdash e = a$$

To prove that the evaluator is indeed a (partial) function, we prove that the notion of reduction satisfies the Church-Rosser property.

Theorem 7. *$\texttt{eval}_{need}$ is a partial function.*

Proof. The theorem is a direct consequence of lemma 3 (Church-Rosser). □

Our strategy is to define a parallel reduction relation for λ_{need} [7]. Define $\rightarrow$ to be the compatible closure of a β_{need} reduction, and $\twoheadrightarrow$ to be the reflexive, transitive closure of $\rightarrow$. Additionally, define $\Rightarrow$ to be the relation that reduces β_{need} redexes in parallel.

Definition 1 ($\Rightarrow$)**.**

$$\begin{array}{rcl}
e & \Rightarrow & e \\
\hat{A}[A_1[\lambda x.\check{A}[E[x]]]\ A_2[v]] & \Rightarrow & \hat{A}'[A_1'[A_2'[\check{A}'[E'[x]]\{x ::= v'\}]]], \\
 & & \quad \textit{if } \hat{A}[\check{A}] \in A,\ \hat{A}'[\check{A}'] \in A,\ \hat{A} \Rightarrow \hat{A}',\ A_1 \Rightarrow A_1', \\
 & & \quad\quad A_2 \Rightarrow A_2',\ \check{A} \Rightarrow \check{A}',\ E \Rightarrow E',\ v \Rightarrow v' \\
e_1\ e_2 & \Rightarrow & e_1'\ e_2',\ \textit{if } e_1 \Rightarrow e_1',\ e_2 \Rightarrow e_2' \\
\lambda x.e & \Rightarrow & \lambda x.e',\ \textit{if } e \Rightarrow e'
\end{array}$$

The parallel reduction relation $\Rightarrow$ relies on notion of parallel reduction for contexts; for simplicity, we overload the relation symbol to denote both relations.

Definition 2 ($\Rightarrow$ for Contexts)**.**

$$\begin{array}{rcl}
[\,] & \Rightarrow & [\,] \\
A_1[\lambda x.A_2]\ e & \Rightarrow & A_1'[\lambda x.A_2']\ e',\ \textit{if}\ A_1 \Rightarrow A_1',\ A_2 \Rightarrow A_2',\ e \Rightarrow e' \\
A[\hat{A}]\ e & \Rightarrow & A'[\hat{A}']\ e',\ \textit{if}\ A \Rightarrow A',\ \hat{A} \Rightarrow \hat{A}',\ e \Rightarrow e' \\
A[\lambda x.\overset{\vee}{A}] & \Rightarrow & A'[\lambda x.\overset{\vee}{A}'],\ \textit{if}\ A \Rightarrow A',\ \overset{\vee}{A} \Rightarrow \overset{\vee}{A}' \\
E\ e & \Rightarrow & E'\ e',\ \textit{if}\ E \Rightarrow E',\ e \Rightarrow e' \\
A[E] & \Rightarrow & A'[E'],\ \textit{if}\ A \Rightarrow A',\ E \Rightarrow E' \\
\hat{A}[A[\lambda x.\overset{\vee}{A}[E_1[x]]]\ E_2] & \Rightarrow & \hat{A}'[A'[\lambda x.\overset{\vee}{A}'[E_1'[x]]]\ E_2'], \\
 & & \quad \textit{if}\ \hat{A}[\overset{\vee}{A}] \in A,\ \hat{A} \Rightarrow \hat{A}',\ A \Rightarrow A', \\
 & & \quad\ \overset{\vee}{A} \Rightarrow \overset{\vee}{A}',\ E_1 \Rightarrow E_1',\ E_2 \Rightarrow E_2'
\end{array}$$

Lemma 3 (Church-Rosser)**.** *If $e \twoheadrightarrow e_1$ and $e \twoheadrightarrow e_2$, then there exists a term e' such that $e_1 \twoheadrightarrow e'$ and $e_2 \twoheadrightarrow e'$.*

Proof. By lemma 4, $\Rightarrow$ satisfies a diamond property. Since $\Rightarrow$ extends $\rightarrow$, $\twoheadrightarrow$ is also the transitive-reflexive closure of $\Rightarrow$, so $\twoheadrightarrow$ also satisfies a diamond property. □

Lemma 4 (Diamond Property of $\Rightarrow$)**.** *If $e \Rightarrow e_1$ and $e \Rightarrow e_2$, there exists e' such that $e_1 \Rightarrow e'$ and $e_2 \Rightarrow e'$.*

Proof. The proof proceeds by structural induction on the derivation of $e \Rightarrow e_1$. □

A.4.2 *Deterministic Behavior: Standard Reduction*

A language calculus should also come with a deterministic algorithm for applying the reductions to evaluate a program. Here is our *standard reduction*:

$$E[e] \longmapsto E[e'],\ \text{where}\ e\ \ \beta_{need}\ \ e'$$

Our standard reduction strategy picks exactly one redex in a term.

Proposition 1 (Unique Decomposition)**.** *For all closed terms e, e either is an answer or $e = E[e']$ for a unique evaluation context E and β_{need} redex e'.*

Proof. The proof proceeds by structural induction on e. □

Since our calculus satisfies the unique decomposition property, we can use the standard reduction relation to define a (partial) evaluator function:

$$\texttt{eval}^{\text{sr}}_{need}(e) = \texttt{done}\ \text{iff there exists an answer}\ a\ \text{such that}\ e \longmapsto\!\!\!\!\twoheadrightarrow a$$

where $\longmapsto\!\!\!\!\twoheadrightarrow$ is the reflexive, transitive closure of $\longmapsto$. Proposition 1 shows $\texttt{eval}^{\text{sr}}_{need}$ is a function. The following theorem confirms that it equals $\texttt{eval}_{need}$.

Theorem 8. $\mathit{eval}_{need} = \mathit{eval}^{sr}_{need}$

Proof. The theorem follows from lemma 5, which shows how to obtain a standard reduction sequence for any arbitrary reduction sequence. The front-end of the former is a series of standard reduction steps. □

Definition 3 (Standard Reduction Sequences $\mathcal{R}$)**.**

- $x \subset \mathcal{R}$
- $\lambda x.e_1 \diamond \cdots \diamond \lambda x.e_m \in \mathcal{R}$, *if* $e_1 \diamond \cdots \diamond e_m \in \mathcal{R}$
- $e_0 \diamond e_1 \diamond \cdots \diamond e_m \in \mathcal{R}$, *if* $e_0 \longmapsto e_1$ *and* $e_1 \diamond \cdots \diamond e_m \in \mathcal{R}$
- $(e_1\ e'_1) \diamond \cdots \diamond (e_m\ e'_1) \diamond (e_m\ e'_2) \diamond \cdots \diamond (e_m\ e'_n) \in \mathcal{R}$, *if* $e_1 \diamond \cdots \diamond e_m, e'_1 \diamond \cdots \diamond e'_n \in \mathcal{R}$.

Lemma 5 (Curry-Feys Standardization)**.** $e \twoheadrightarrow e'$ *iff there exists* $e_1 \diamond \cdots \diamond e_n \in \mathcal{R}$ *such that* $e = e_1$ *and* $e' = e_n$.

Proof. Replace $\twoheadrightarrow$ with $\Rightarrow$s, and the lemma immediately follows from lemma 6. □

The key to the remaining proofs is a size metric for parallel reductions.

Definition 4 (Size of $\Rightarrow$ Reduction)**.**

$$
\begin{array}{lcl}
\left|e \Rightarrow e\right| & = & 0 \\
\left|(e_1\ e_2) \Rightarrow (e'_1\ e'_2)\right| & = & \left|e_1 \Rightarrow e'_1\right| + \left|e_2 \Rightarrow e'_2\right| \\
\left|\lambda x.e \Rightarrow \lambda x.e'\right| & = & \left|e \Rightarrow e'\right| \\
\left|r\right| & = & 1 + \left|\hat{A} \Rightarrow \hat{A}'\right| + \left|A_1 \Rightarrow A'_1\right| + \left|\check{A}[E[x]] \Rightarrow \check{A}'[E'[x]]\right| + \\
 & & \qquad \left|A_2 \Rightarrow A'_2\right| + \#(x, \check{A}'[E'[x]]) \times \left|v \Rightarrow v'\right|
\end{array}
$$

where

$$r = \hat{A}[A_1[\lambda x.\check{A}[E[x]]\ A_2[v]] \Rightarrow \hat{A}'[A'_1[A'_2[\check{A}'[E'[x]]\{x ::= v'\}]]]$$

$\#(x, e) =$ *the number of free occurrences of* x *in* e

The size of a parallel reduction of a context equals the sum of the sizes of the parallel reductions of the subcontexts and subterms that comprise the context.

Lemma 6. *If* $e_0 \Rightarrow e_1$ *and* $e_1 \diamond \cdots \diamond e_n \in \mathcal{R}$, *there exists* $e_0 \diamond e'_1 \diamond \cdots \diamond e'_p \diamond e_n \in \mathcal{R}$.

Proof. By triple lexicographic induction on (1) length n of the given standard reduction sequence, (2) $\left|e_0 \Rightarrow e_1\right|$, and (3) structure of e_0.[5] □

5 We conjecture that the use of Ralph Loader's technique [50] may simplify our proof.

A.4.3 *Observational Equivalence*

Following Morris [56] two expressions e_1 and e_2 are observationally equivalent, $e_1 \simeq e_2$, if they are indistinguishable in all contexts. Formally, $e_1 \simeq e_2$ if and only if $\texttt{eval}_{need}(C[e_1]) = \texttt{eval}_{need}(C[e_2])$ for all contexts C, where

$$C = [\,] \mid \lambda x.C \mid C\ e \mid e\ C \qquad \text{(Contexts)}$$

It follows from the above theorems that λ_{need} is sound with respect to observational equivalence.

Theorem 9 (Soundness)**.** *If* $\boldsymbol{\lambda_{need}} \vdash e_1 = e_2$*, then* $e_1 \simeq e_2$*.*

Proof. Following Plotkin, a calculus is sound if it satisfies Church-Rosser and Curry-Feys theorems. □

A.5 CORRECTNESS

Ariola and Felleisen [5] prove that λ_{af} defines the same evaluation function as the call-by-name λ-calculus. Nakata and Hasegawa [58] additionally demonstrate extensional correctness of the same calculus with respect to Launchbury's natural semantics [47]. In this section, we show that λ_{need} defines the same evaluation function as Launchbury's semantics. While our theorem statement is extensional, the proof illuminates the tight intensional relationship between the two systems.

A.5.1 *Overview*

The gap between the λ_{need} standard reduction "machine" and Launchbury's natural semantics is huge. While the latter's store-based natural semantics uses the equivalent of assignment statements to implement the "evaluate once, only when needed" policy, the λ_{need} calculus exclusively relies on term substitutions. To close the gap, we systematically construct a series of intermediate systems that makes comparisons easy, all while ensuring correctness at each step. A first step is to convert the natural semantics into a store-based machine [71].

To further bridge the gap we note that a single-use assignment statement is equivalent to a program-wide substitution of shared expressions [24]. A closely related idea is to reduce shared expressions simultaneously. This leads to a parallel program rewriting system, dubbed $\lambda_{||}$. Equipped with $\lambda_{||}$ we get closer to λ_{need} but not all the way there because reductions in λ_{need} and $\lambda_{||}$ are too coarse-grained for direct comparison. Fortunately, it is easy to construct an intermediate transition system that eliminates the remainder of the gap. We convert λ_{need} to an equivalent CK transition system [25], where the program is partitioned into a control string (C) and an explicit context (K) and we show that there is a correspondence between this transition system and $\lambda_{||}$.

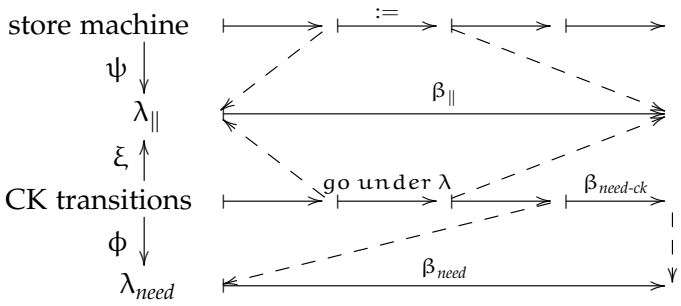

Figure 28: Summary of correctness proof technique.

Figure 28 outlines our proof strategy pictorially. The four horizontal layers correspond to the four rewriting systems. While λ_{need} and $\lambda_{||}$ use large steps to progress from term to term, the machine-like systems take several small steps. The solid vertical arrows between the layers figure indicate how mapping functions relate the rewriting sequences and the dashed arrows show how the smaller machine steps correspond to the larger steps of λ_{need} and $\lambda_{||}$:

- The ψ function maps states from the store-based machine to terms in the $\lambda_{||}$ world. For every step in the natural-semantics machine, the resulting operation in $\lambda_{||}$ is either a no-op or a $\beta_{||}$ reduction, with assignment in the store-machine being equivalent to program-wide substitution in $\lambda_{||}$.
- Similarly, the ξ function maps states of the CK transition system to the $\lambda_{||}$ space and for every CK transition, the resulting $\lambda_{||}$ operation is also either a no-op or a $\beta_{||}$ reduction, with the transition that descends under a λ-abstraction being equivalent to substitution in $\lambda_{||}$.
- Finally, the ϕ function maps states of the CK transition system to λ_{need} terms and is used to show that the CK system and λ_{need} are equivalent. For every CK transition, the equivalent λ_{need} operation is either a no-op or a β_{need} reduction.

Subsections A.5.2 and A.5.3 present the store-based machine and the parallel rewriting semantics, respectively, including a proof of equivalence. Subsection A.5.4 presents the CK system and subsection A.5.5 explains the rest of the proof.

A.5.2 *Adapting Launchbury's Natural Semantics*

Figure 29 describes the syntax and transitions of the store machine.[6] It is dubbed CKH because it resembles a three-register machine [25]: a machine state S_L is comprised of a control string (C), a list of frames (K) that represents the control context in an inside-out manner, and a heap (H). The ... notation means "zero of more of the preceding kind of element." An (`arg` e) frame represents the argument in an application and the (`var x`) frame indicates that a heap expression is the current control string. Parentheses are used to group a list of frames when necessary. The initial machine state for a program e is $\langle e, (), () \rangle$. Computation terminates when the control string is a value and the list of frames is empty.

The *push-arg-ckh* transition moves the argument in an application to a new `arg` frame in the frame list and makes the operator the next control string.

6 To aid comparisons, we slightly alter Launchbury's rules (and the resulting machine) to use pure λ terms. Thus we avoid Launchbury's preprocessing and special syntax.

Syntax

$$
\begin{aligned}
S_L &= \langle e, F_L s, \Gamma \rangle && \text{(States)} \\
F_L s &= F_L, \ldots && \text{(List of Frames)} \\
F_L &= (\texttt{arg}\, e) \mid (\texttt{var}\, x) && \text{(Frames)} \\
\Gamma &= (x \mapsto e, \ldots) && \text{(Heaps)}
\end{aligned}
$$

Transitions

(*push-arg-ckh*)

$$\langle e_1\ e_2, F_L s, \Gamma \rangle \overset{ckh}{\longmapsto} \langle e_1, ((\texttt{arg}\, e_2), F_L s), \Gamma \rangle$$

(*descend-lam-ckh*)

$$\langle \lambda x.e_1, ((\texttt{arg}\, e_2), F_L s), \Gamma \rangle \overset{ckh}{\longmapsto} \langle e_1\{x ::= y\}, F_L s, (\Gamma, y \mapsto e_2) \rangle \text{ , } y \text{ fresh}$$

(*lookup-var-ckh*)

$$\langle x, F_L s, (\Gamma, x \mapsto e) \rangle \overset{ckh}{\longmapsto} \langle e, ((\texttt{var}\, x), F_L s), \Gamma \rangle$$

(*update-heap-ckh*)

$$\langle v, ((\texttt{var}\, x), F_L s), \Gamma \rangle \overset{ckh}{\longmapsto} \langle v, F_L s, (\Gamma, x \mapsto v) \rangle$$

Figure 29: The natural semantics as an abstract machine.

When that operator is a λ-abstraction, the *descend-lam-ckh* transition adds its argument to the heap,[7] mapped to a fresh variable name, and makes the body of the operator the new control string. The *lookup-var-ckh* transition evaluates an argument from the heap when the control string is a variable. The mapping is removed from the heap and a new $(\texttt{var}\ x)$ frame remembers the variable whose corresponding expression is under evaluation. Finally, when the heap expression is reduced to a value, the *update-heap-ckh* transition extends the heap again.

A.5.3 *Parallel Rewriting*

The syntax of the parallel λ-rewriting semantics is as follows:

$$e_{\|} = e \mid {e_{\|}}^{x} \qquad \text{(Terms)}$$
$$v_{\|} = v \mid {v_{\|}}^{x} \qquad \text{(Values)}$$
$$E_{\|} = [\,] \mid E_{\|}\ e_{\|} \mid {E_{\|}}^{x} \qquad \text{(Evaluation Contexts)}$$

This system expresses computation with a selective parallel reduction strategy. When a function application is in demand, the system substitutes the argument for all free occurrences of the bound variable, regardless of the status of the argument. When an instance of a substituted argument is reduced, however, all instances of the argument are reduced in parallel. Here is a sample reduction:

$$(\lambda x.x\ x)\ (I\ I) \xrightarrow{\|} (I\ I)^{x}\ (I\ I)^{x} \xrightarrow{\|} I^{x}\ I^{x} \xrightarrow{\|} I$$

The $\lambda_{\|}$ semantics keeps track of arguments via labeled terms ${e_{\|}}^{x}$, where labels are variables. The set of values includes labeled λ-abstractions. Reducing a labeled term triggers the simultaneous reduction of all other terms with the same label. Otherwise, labels do not affect program evaluation.

We require that all expressions with the same label must be identical.

Definition 5. *A program* $e_{\|}$ *is* consistently labeled (CL) *when for any two subterms* ${e_{\|}}_1^{x_1}$ *and* ${e_{\|}}_2^{x_2}$ *of* $e_{\|}$, $x_1 = x_2$ *implies* $e_{\|1} = e_{\|2}$.

In the reduction of $\lambda_{\|}$ programs, evaluation contexts $E_{\|}$ determine which part of the program to reduce next. The $\lambda_{\|}$ evaluation contexts are the call-by-name evaluation contexts with the addition of the labeled ${E_{\|}}^{x}$ context, which dictates that a redex search goes under labeled terms. Essentially, when searching for a redex, terms tagged with a label are treated as if they were unlabeled.

7 The notation $(\Gamma, x{\mapsto}e)$ is a heap Γ, extended with the variable-term mapping $x{\mapsto}e$.

The parallel semantics can exploit simpler evaluation contexts than λ_{need} because substitution occurs as soon as an application is encountered:

$$\mathsf{E}_{\|}[((\lambda x.e_{\|1})^{\vec{y}})\ e_{\|2}] \xrightarrow{\|} \begin{cases} \mathsf{E}_{\|}[e_{\|}], & \text{if } [\,] \text{ is not under a label in } \mathsf{E}_{\|} \\ \mathsf{E}_{\|}[e_{\|}]\{z \Leftarrow \mathsf{E}_{\|2}[e_{\|}]\}, & \text{if } \mathsf{E}_{\|}[\,] = \mathsf{E}_{\|1}[(\mathsf{E}_{\|2}[\,])^{z}] \\ & \text{and } [\,] \text{ is not under a label in } \mathsf{E}_{\|2} \end{cases} \qquad (\beta_{\|})$$

$$\text{where } e_{\|} = e_{\|1}\{x ::= e_{\|2}^{w}\}, w \text{ fresh}$$

On the left-hand side of $\beta_{\|}$, the program is partitioned into a context and a β-like redex. A term $e^{\vec{y}}$ may have any number of labels and possibly none. On the right-hand side, the redex is contracted to a term $e_{\|1}\{x ::= e_{\|2}^{w}\}$ such that the argument is tagged with an unique label w. Obsolete labels $\vec{y}$ are discarded.

There are two distinct ways to contract a redex: when the redex is not under any labels and when the redex occurs under at least one label. For the former, the redex is the only contracted part of the program. For the latter, all other instances of that labeled term are similarly contracted. In this second case, the evaluation context is further subdivided as $\mathsf{E}_{\|}[\,] = \mathsf{E}_{\|1}[(\mathsf{E}_{\|2}[\,])^{z}]$, where z is the label nearest the redex, i.e., $\mathsf{E}_{\|2}$ contains no additional labels. A whole-program substitution function is used to perform the parallel reduction:

$$\begin{aligned} e_{\|1}^{\,x}\{x \Leftarrow e_{\|}\} &= e_{\|}^{\,x} \\ e_{\|1}^{\,x}\{y \Leftarrow e_{\|}\} &= (e_{\|1}\{y \Leftarrow e_{\|}\})^{x},\ x \neq y \\ (\lambda x.e_{\|1})\{x \Leftarrow e_{\|}\} &= \lambda x.(e_{\|1}\{x \Leftarrow e_{\|}\}) \\ (e_{\|1}\ e_{\|2})\{x \Leftarrow e_{\|}\} &= (e_{\|1}\{x \Leftarrow e_{\|}\}\ e_{\|2}\{x \Leftarrow e_{\|}\}) \\ \text{otherwise, } e_{\|1}\{x \Leftarrow e_{\|}\} &= e_{\|1} \end{aligned}$$

Rewriting terms with $\beta_{\|}$ preserves the consistent labeling property.

Proposition 2. *If $e_{\|}$ is CL and $e_{\|} \xrightarrow{\|} e_{\|}'$, then $e_{\|}'$ is CL.*

The ψ function reconstructs a $\lambda_{\|}$ term from a CKH machine configuration:

$$\psi(\langle e, ((\mathtt{var}\ x), \mathsf{F_L}s), \Gamma\rangle) = \psi(\langle x, \mathsf{F_L}s, (x \mapsto e, \Gamma)\rangle) \qquad \boxed{\psi : \mathsf{S_L} \to e_{\|}}$$

$$\psi(\langle e_1, ((\mathtt{arg}\ e_2), \mathsf{F_L}s), \Gamma\rangle) = \psi(\langle e_1\ e_2, \mathsf{F_L}s, \Gamma\rangle)$$

$$\psi(\langle e, (), \Gamma\rangle) = e\{\Gamma\}$$

The operation $e\{\Gamma\}$, using overloaded notation, replaces all free variables in e with their corresponding terms in Γ and tags them with appropriate labels.

Lemma 7 demonstrates the bulk of the equivalence of the store machine and $\lambda_{\|}$.[8] The rest of the equivalence proof is straightforward [25].

8 The lemma relies on an extension of the typical α-equivalence classes of terms to include variables in labels as well.

Lemma 7. *If* $\langle e, F_L s, \Gamma\rangle \overset{ckh}{\longmapsto} \langle e', F_L s', \Gamma'\rangle$, *then either:*

1. $\psi(\langle e, F_L s, \Gamma\rangle) = \psi(\langle e', F_L s', \Gamma'\rangle)$
2. $\psi(\langle e, F_L s, \Gamma\rangle) \overset{\|}{\longmapsto} \psi(\langle e', F_L s', \Gamma'\rangle)$

A.5.4 *A Transition System for Comparing* λ_{need} *and* $\lambda_{\|}$

The CK layer in figure 28 mediates between $\lambda_{\|}$ and λ_{need}. The corresponding transition system resembles a two-register CK machine [25]. Figure 30 describes the syntax and the transitions of the system.[9]

Syntax

$$S = \langle e, Fs\rangle \qquad \text{(States)}$$
$$Fs = F, \ldots \qquad \text{(List of Frames)}$$
$$F = (\texttt{arg}\, e) \mid (\texttt{lam}\, x) \mid (\texttt{bod}\, x\, Fs\, Fs) \qquad \text{(Frames)}$$

Transitions

(*push-arg-ck*)
$$\langle e_1\ e_2, Fs\rangle \overset{ck}{\longmapsto} \langle e_1, ((\texttt{arg}\, e_2), Fs)\rangle$$

(*descend-lam-ck*)
$$\langle \lambda x.e, Fs\rangle \overset{ck}{\longmapsto} \langle e, ((\texttt{lam}\, x), Fs)\rangle\,, \quad \text{if } \texttt{balance}(Fs) > 0$$

(*lookup-var-ck*)
$$\langle x, (Fs_1, (\texttt{lam}\, x), Fs_2, (\texttt{arg}\, e), Fs)\rangle \overset{ck}{\longmapsto} \langle e, ((\texttt{bod}\, x\, Fs_1\, Fs_2), Fs)\rangle\,, \quad \text{if } \phi_F(Fs_1) \in \overset{\vee}{A}[E], \phi_F(Fs_2) \in A, \phi_F(Fs) \in E[\hat{A}], \hat{A}[\overset{\vee}{A}] \in A$$

($\beta_{need\text{-}ck}$)
$$\langle v, (Fs_3, (\texttt{bod}\, x\, Fs_1\, Fs_2), Fs)\rangle \overset{ck}{\longmapsto} \langle v, (Fs_1\{x ::= v\}, Fs_3, Fs_2, Fs)\rangle\,, \quad \text{if } \phi_F(Fs_3) \in A$$

Figure 30: A transition system for comparing λ_{need} and $\lambda_{\|}$.

States consist of a subterm and a list of frames representing the context. The first kind of frame represents the argument in an application and the second frame represents a λ-abstraction with a hole in the body. The last kind

9 The CK transition system is a proof-technical device. Unlike the original CK machine, ours is ill-suited for an implementation.

of frame has two frame list components, the first representing a context in the body of the λ, and the second representing the context between the λ and its argument. The variable in this last frame is the variable bound by the λ expression under evaluation. The initial state for a program e is $\langle e, () \rangle$, where () is an empty list of frames, and evaluation terminates when the control string is a value and the list of frames is equivalent to an answer context.

The *push-arg-ck* transition makes the operator in an application the new control string and adds a new `arg` frame to the frame list containing the argument. The *descend-lam-ck* transition goes under a λ, making the body the control string, but only if that λ has a corresponding argument in the frame list, as determined by the `balance` function, defined as follows:

$$\boxed{\texttt{balance} : \mathbb{Fs} \to \mathbb{Z}}$$

$$\texttt{balance}(\mathbb{Fs}_3, (\texttt{bod}\, x\, \mathbb{Fs}_1\ \mathbb{Fs}_2), \mathbb{Fs}) = \texttt{balance}(\mathbb{Fs}_3)$$

$$\texttt{balance}(\mathbb{Fs}) = \texttt{\#arg-frames}(\mathbb{Fs}) - \texttt{\#lam-frames}(\mathbb{Fs})$$

$\mathbb{Fs}$ contains no `bod` frames

The `balance` side condition for *descend-lam-ck* dictates that evaluation goes under a λ only if there is a matching argument for it, thus complying with the analogous evaluation context. The `balance` function employs `#arg-frames` and `#lam-frames` to count the number of `arg` or `lam` frames, respectively, in a list of frames. Their definitions are elementary and therefore omitted.

The *lookup-var-ck* transition is invoked if the control string is a variable, somewhere in a λ body, and the rest of the frames have a certain shape consistent with the corresponding parts of a β_{need} redex. With this transition, the argument associated with the variable becomes the next control string and the context around the variable in the λ body and the context between the λ and argument are saved in a new `bod` frame. Finally, when an argument is an answer, indicated by a value control string and a `bod` frame in the frame list—with the equivalent of an answer context in between—the value gets substituted into the body of the λ according to the $\beta_{need\text{-}ck}$ transition. The $\beta_{need\text{-}ck}$ transition uses a substitution function on frame lists, $\mathbb{Fs}\{x ::= e\}$, which overloads the notation for regular term substitution and has the expected definition.

Figure 31 defines metafunctions for the CK transition system. The ϕ function converts a CK state to the equivalent λ_{need} term, and uses $\phi_{\mathbb{F}}$ to convert a list of frames to an evaluation context.

Now we can show that an evaluator defined with $\overset{ck}{\longmapsto}$ is equivalent to $\texttt{eval}^{\text{sr}}_{need}$. The essence of the proof is a lemma that relates the shape of CK transition sequences to the shape of λ_{need} standard reduction sequences. The rest of the equivalence proof is straightforward [25].

$$\boxed{\phi : S \to e}$$

$$\phi(\langle e, \mathsf{Fs}\rangle) = \phi_{\mathsf{F}}(\mathsf{Fs})[e]$$

$$\boxed{\phi_{\mathsf{F}} : \mathsf{Fs} \to E}$$

$$\begin{aligned}
\phi_{\mathsf{F}}(()) &= [\,] \\
\phi_{\mathsf{F}}((\mathtt{lam}\, x), \mathsf{Fs}) &= \phi_{\mathsf{F}}(\mathsf{Fs})[\lambda x.[\,]] \\
\phi_{\mathsf{F}}((\mathtt{arg}\, e), \mathsf{Fs}) &= \phi_{\mathsf{F}}(\mathsf{Fs})[[\,]\ e] \\
\phi_{\mathsf{F}}((\mathtt{bod}\, x\, \mathsf{Fs}_1\ \mathsf{Fs}_2), \mathsf{Fs}) &= \phi_{\mathsf{F}}(\mathsf{Fs})[\phi_{\mathsf{F}}(\mathsf{Fs}_2)[\lambda x.\phi_{\mathsf{F}}(\mathsf{Fs}_1)[x]]\ [\,]]
\end{aligned}$$

$$\boxed{\xi : S \to e_{\|}}$$

$$\xi(\langle e, \mathsf{Fs}\rangle) = \xi_{\mathsf{F}}(\mathsf{Fs}, e)$$

$$\boxed{\xi_{\mathsf{F}} : \mathsf{Fs} \times e_{\|} \to e_{\|}}$$

$$\begin{aligned}
\xi_{\mathsf{F}}((), e_{\|}) &= e_{\|} \\
\xi_{\mathsf{F}}(((\mathtt{arg}\, e_{\|1}), \mathsf{Fs}), e_{\|}) &= \xi_{\mathsf{F}}(\mathsf{Fs}, e_{\|}\ e_{\|1}) \\
\xi_{\mathsf{F}}(((\mathtt{bod}\, x\, \mathsf{Fs}_1\ \mathsf{Fs}_2), \mathsf{Fs}), e_{\|}) &= \xi_{\mathsf{F}}((\mathsf{Fs}_1, (\mathtt{lam}\, x), \mathsf{Fs}_2, (\mathtt{arg}\, e_{\|}), \mathsf{Fs}), x) \\
\xi_{\mathsf{F}}(((\mathtt{lam}\, x), \mathsf{Fs}_1, (\mathtt{arg}\, e_{\|1}), \mathsf{Fs}_2), e_{\|}) &= \xi_{\mathsf{F}}((\mathsf{Fs}_1, \mathsf{Fs}_2), e_{\|}\{x ::= e_{\|1}^{\,y}\}) \\
&\qquad \phi_{\mathsf{F}}(\mathsf{Fs}_1) \in A,\ y \text{ fresh}
\end{aligned}$$

Figure 31: Functions to map CK states to λ_{need} (ϕ) and $\lambda_{\|}$ (ξ).

Lemma 8. *If* $\langle e, Fs\rangle \overset{ck}{\longmapsto} \langle e', Fs'\rangle$*, then either:*

1. $\phi(\langle e, Fs\rangle) = \phi(\langle e', Fs'\rangle)$
2. $\phi(\langle e, Fs\rangle) \longmapsto \phi(\langle e', Fs'\rangle)$

Finally, we show how the CK system corresponds to $\lambda_{\|}$. The ξ function defined in figure 31 constructs a $\lambda_{\|}$ term from a CK configuration.

Lemma 9. *If* $\langle e, Fs\rangle \overset{ck}{\longmapsto} \langle e', Fs'\rangle$*, then either:*

1. $\xi(\langle e, Fs\rangle) = \xi(\langle e', Fs'\rangle)$
2. $\xi(\langle e, Fs\rangle) \overset{\|}{\longmapsto} \xi(\langle e', Fs'\rangle)$

A.5.5 *Relating all Layers*

In the previous subsections, we have demonstrated the correspondence between $\lambda_{\|}$, the natural semantics, and the λ_{need} standard reduction sequences via lemmas 7 through 9. We conclude this section with the statement of an extensional correctness theorem, where $\texttt{eval}_{natural}$ is an evaluator defined with the store machine transitions. The theorem follows from the composition of the equivalences of our specified rewriting systems.

Theorem 10. $\texttt{eval}_{need} = \texttt{eval}_{natural}$

A.6 EXTENSIONS AND VARIANTS

Data Constructors Real-world lazy languages come with data structure construction and extraction operators. Like function arguments, the arguments to a data constructor should not be evaluated until there is demand for their values [30, 34]. The standard λ-calculus encoding of such operators [7] works well:

$$\begin{aligned}\texttt{cons} &= \lambda x.\lambda y.\lambda s.s\,x\,y\\ \texttt{car} &= \lambda p.p\ \lambda x.\lambda y.x\\ \texttt{cdr} &= \lambda p.p\ \lambda x.\lambda y.y\end{aligned}$$

Adding true algebraic systems should also be straightforward.

Recursion Our λ_{need} calculus represents just a core λ-calculus and does not include an explicit `letrec` constructor for cyclic terms. Since cyclic programming is an important idiom in lazy programming languages, others have extensively explored cyclic by-need calculi, e.g., Ariola and Blum [3], and applying their solutions to our calculus should pose no problems.

A.7 CONCLUSION

Following Plotkin's work on call-by-name and call-by-value, we present a call-by-need λ-calculus that expresses computation via a single axiom in the spirit of β. Our calculus is close to implementations of lazy languages because it captures the idea of by-need computation without retaining every function call and without need for re-associating terms. We show that our calculus satisfies Plotkin's criteria, including an intensional correspondence between our calculus and the conventional natural semantics [47]. Our future work will leverage our λ_{need} calculus to derive a new abstract machine for lazy languages.

BIBLIOGRAPHY

[1] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 34–49, 1995.

[2] Andrew Appel, Matthias Blume, Emden Gansner, Lal George, Lorenz Huelsbergen, David MacQueen, John Reppy, and Zhong Shao. *Standard ML of New Jersey User's Guide*, 1997.

[3] Zena Ariola and Stefan Blom. Cyclic lambda calculi. In *TACS '97: Proceedings of the 3rd International Symposium on Theoretical Aspects of Computer Software*, pages 77–106, 1997.

[4] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda-calculus. Technical Report CIS-TR-94-23, University of Oregon, 1994.

[5] Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7:265–301, 1997.

[6] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 233–246, 1995.

[7] Hendrik Pieter Barendregt. *Lambda Calculus, Syntax and Semantics*. North-Holland, 1985.

[8] Conrad Barski. *Land of Lisp*. No Starch Press, 2011.

[9] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.

[10] G L Burn, C L Hankin, and S Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7(0):249–78, 1986.

[11] Stephen Chang. Laziness by need. In *ESOP '13: Proceedings of the 22nd European Symposium on Programming*, pages 81–100, 2013.

[12] Stephen Chang and Matthias Felleisen. The call-by-need lambda calculus, revisited. In *ESOP '12: Proceedings of the 21st European Symposium on Programming*, pages 128–147, 2012.

[13] Stephen Chang and Matthias Felleisen. Profiling for laziness. In *POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 349–360, 2014.

[14] Stephen Chang, David Van Horn, and Matthias Felleisen. Evaluating call by need on the control stack. In *TFP '10: Revised Selected Papers of the 11th International Symposium on Trends in Functional Progrmaming*, pages 1–15, 2010.

[15] Alonzo Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[16] Chris Clack and Simon L. Peyton Jones. Strictness analysis—a practical approach. In *FPCA '85: Proceedings of the 2nd ACM Conference on Functional Programming Languages and Computer Architecture*, pages 35–49, 1985.

[17] Marcus Crestani and Michael Sperber. Experience report: growing programming languages for beginning students. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 229–234, 2010.

[18] Haskell B. Curry and Robert Feys. *Combinatory Logic, Vol. I*. North-Holland, 1958.

[19] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. Defunctionalized interpreters for call-by-need evaluation. In *FLOPS '10: Proceedings of the 10th International Symposium on Functional and Logic Programming*, pages 240–256, 2010.

[20] Robert Ennals and Simon Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *ICFP '03: Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298, 2003.

[21] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In *FPCA '87: Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, volume 274, pages 34–45. Springer Berlin Heidelberg, 1987.

[22] Karl-Filip Faxén. Optimizing lazy functional programs using flow inference. In *SAS '95: Proceedings of the 2nd International Symposium on Static Analysis*, pages 136–153, 1995.

[23] Karl-Filip Faxén. Cheap eagerness: speculative evaluation in a lazy functional language. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 150–161, 2000.

[24] Matthias Felleisen and Daniel P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3):243–287, 1989.

[25] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[26] Matthias Felleisen, Conrad Barski, David Van Horn, Forrest Bice, Rose DeMaio, Spencer Florence, Feng-Yun Mimi Lin, Scott Lindeman, Nicole Nussbaum, Eric Peterson, and Ryan Plessner. *Realm of Racket*. No Starch Press, 2013.

[27] Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In *IFL '07: Revised Selected Papers of the 19th International Workshop on the Implementation and Application of Functional Languages*, pages 111–128, 2007.

[28] Cormac Flanagan and Matthias Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, 1996.

[29] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/.

[30] Daniel P. Friedman and David S. Wise. Cons should not evaluate its arguments. In *ICALP '76: Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 257–281, 1976.

[31] Daniel P. Friedman, Abdulaziz Ghuloum, Jeremy G. Siek, and Onnie Lynn Winebarger. Improving the lazy Krivine machine. *Higher Order Symbolic Computation*, 20:271–293, 2007.

[32] Ronald Garcia, Andrew Lumsdaine, and Amr Sabry. Lazy evaluation and delimited control. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 153–164, 2009.

[33] *The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3*. The GHC Team, April 2013.

[34] Peter Henderson and James H. Morris, Jr. A lazy evaluator. In *POPL '76: Proceedings of the 3rd ACM Symposium on Principles of Programming Languages*, pages 95–103, 1976.

[35] Rich Hickey. *Clojure API Overview - Clojure v1.5 (stable)*.

[36] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. In *Journal of Functional Programming*, pages 197–217, 2006.

[37] Paul Hudak and Jonathan Young. Higher-order strictness analysis in untyped lambda calculus. In *POPL '86: Proceedings of the 13th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[38] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL '07: Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pages 12–1–12–55, 2007.

[39] John Hughes. Strictness detection in non-flat domains. In *Proceedings of the Workshop on Programs as Data Objects*, pages 112–135, 1985.

[40] John Hughes. Why functional programming matters. *Computer Journal*, 32:98–107, 1989.

[41] Neil D. Jones. Flow analysis of lambda expressions. Technical report, Aarhus University, 1981.

[42] Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. In *Functional Programming*, pages 201–220, 1993.

[43] Mark B. Josephs. The semantics of lazy functional languages. *Theoretical Computer Science*, 68(1), 1989.

[44] Jean-Louis Krivine. Un interpréteur du lambda-calcul, 1985.

[45] Peter J. Landin. A correspondence between algol 60 and church's lambda notation: Part i. *Communications of the ACM*, 8(2):89–101, 1965.

[46] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–66, 1966.

[47] John Launchbury. A natural semantics for lazy evaluation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, 1993.

[48] Daan Leijen and Erik Meijer. Parsec: direct style monadic parser combinators for the real world. TR UU-CS-2001-35, Utrecht University, 2001.

[49] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, 2007.

[50] Ralph Loader. Notes on simply typed lambda calculus. Technical Report ECS-LFCS-98-381, Department of Computer Science, University of Edinburgh, 1998.

[51] Jan-Willem Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 38–50, 2002.

[52] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus (unabridged). Technical Report 28/94, Universität Karlsruhe, 1994.

[53] John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. Call-by-name call-by-value, call-by-need, and the linear lambda calculus. In *MFPS '95: Proceedings of the 11th International Conference on Mathematical Foundations of Programming Semantics*, pages 370–392, 1995.

[54] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8:275–317, 1998.

[55] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. Evaluating the design of the R language. In *ECOOP '12: Proceedings of the 26th European Conference on Object-Oriented Programming*, pages 104–131, 2012.

[56] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[57] A Mycroft. *Abstract interpretation and optimising transformations for applicative programs*. PhD thesis, University of Edinburgh, 1981.

[58] Keiko Nakata and Masahito Hasegawa. Small-step and big-step semantics for call-by-need. *Journal of Functional Programming*, 19(6):699–722, 2009.

[59] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[60] Rishiyur S. Nikhil. *Id Language Reference Manual*. MIT, July 1991.

[61] Martin Odersky. *The Scala Language Specification, Version 2.9*. EPFL, May 2011.

[62] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[63] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, 2008.

[64] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *FPCA '89: Proceedings of the 4th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, 1989.

[65] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[66] Hari Prashanth K R and Sam Tobin-Hochstadt. Functional data structures for Typed Racket. In *Proceedings of the Scheme and Functional Programming Workshop*, 2010.

[67] S. Purushothaman and Jill Seaman. An adequate operational semantics for sharing in lazy evaluation. In *ESOP '92: Proceedings of the 4th European Symposium on Programming*, pages 435–450, 1992.

[68] Jonathan Rees and William Clinger. Revised^3 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, December 1986.

[69] Klaus E. Schauser and Seth C. Goldstein. How much non-strictness do lenient programs require? In *FPCA '95: Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 216–225, 1995.

[70] Peter Sestoft. Replacing function parameters by global variables. Master's thesis, University of Copenhagen, 1988.

[71] Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–64, 1997.

[72] Tim Sheard. A pure language with default strict evaluation order and explicit laziness. In *ACM SIGPLAN Workshop on Haskell: New Ideas session*, 2003.

[73] Olin Shivers. Control-flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 1988.

[74] Olin Shivers. Useless-variable elimination. In *WSA '91: Proceedings of the Workshop on Static Analysis*, pages 197–201, 1991.

[75] Guy Lewis Steele, Jr. and Gerald Jay Sussman. Lambda: The ultimate imperative. Technical Report AIM-353, MIT, 1976.

[76] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 395–406, 2008.

[77] Christina von Dorrien. *Stingy Evaluation*. Licentiate thesis, Chalmers University of Technology, 1989.

[78] Philip Wadler. How to replace failure by a list of successes. In *FPCA '85: Proceedings of the 2nd ACM Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, 1985.

[79] Philip Wadler and John Hughes. Projections for strictness analysis. In *FPCA '87: Proceedings of the 3rd ACM Conference on Functional Programming Languages and Computer Architecture*, pages 385–407, 1987.

[80] Philip Wadler, Walid Taha, and David MacQueen. How to add laziness to a strict language, without even being odd. In *Proceedings of the Workshop on Standard ML*, 1998.

[81] Christorpher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University, 1971.

[82] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[83] Haskell Wiki. foldr, foldl, and foldl'. `http://www.haskell.org/haskellwiki/Foldr_Foldl_Foldl{%27}`, 2008. [Online; accessed 2014-March-19].

[84] Stuart C. Wray. *Implementation and programming techniques for functional languages*. PhD thesis, University of Cambridge, 1986.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of May 21, 2014 (`classicthesis` version 4.1).