Profiling for Laziness

Stephen Chang
stchang@ccs.neu.edu

PLT at Northeastern University
Boston, MA 02115

Matthias Felleisen
matthias@ccs.neu.edu

Abstract

While many programmers appreciate the benefits of lazy programming at an abstract level, determining which parts of a concrete program to evaluate lazily poses a significant challenge for most of them. Over the past thirty years, experts have published numerous papers on the problem, but developing this level of expertise requires a significant amount of experience.

We present a profiling-based technique that captures and automates this expertise for the insertion of laziness annotations into strict programs. To make this idea precise, we show how to equip a formal semantics with a metric that measures waste in an evaluation. Then we explain how to implement this metric as a dynamic profiling tool that suggests where to insert laziness into a program. Finally, we present evidence that our profiler's suggestions either match or improve on an expert's use of laziness in a range of real-world applications.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms laziness; profiling; code refactoring

1. Where To Be Lazy

Experts have extolled the benefits of laziness for decades now. As Hughes explained [12], lazy programming languages enable programmers to design code in a modular way; even earlier, Abelson and Sussman [1] argued that programs in strict languages can profit from laziness. Programmers have learned, however, that the key to deploying laziness is moderation, because laziness severely complicates reasoning about a program's resource consumption [11].

Approaches to taming laziness come from two different directions. From one direction, lazy language researchers have devised strategies for determining when to safely *remove* laziness [4, 6, 16]. Because these methods still tend to leave too much laziness behind, lazy programmers frequently annotate their programs with strictness annotations such as Haskell's `seq` [18]. From the other direction, strict programmers *add* laziness via constructs such as `delay` and `force`, for example to manage large data structures [17], delay possibly unneeded tasks [8], or leverage other lazy design patterns [21]. Though the strict approach is appealing since most programs need only a small amount of laziness [5, 14, 15, 20], finding exactly where to add the laziness can be problematic.

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535887

Consider a programmer who wishes to insert laziness annotations into a finite strict program in order to improve its performance. The problem is that laziness is a flow-oriented, global property of programs. Inserting an annotation in one place tends to require additional annotations at other, distant points in the program. Omitting even one of these annotations can result in the complete loss of the desired benefits. In short, the difficulty in placing laziness in strict programs is a "devastating handicap."[1]

Recently, Chang [3] tackled the problem with a static analysis-based tool. This static analysis 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. Finally, Chang reports that he occasionally has to run his tool more than once to find expert-level placement but offers no guidance on this issue.

In this paper, we present a dynamic solution that interprets information from an execution to determine which parts of the program to evaluate lazily. Our solution consists of three concrete contributions. First, we show how to equip a call-by-value λ-calculus semantics with a metric for assessing each expression's *laziness potential*. Roughly, laziness potential represents the amount of "unneeded computation" performed by the evaluation of an expression. It thus predicts the degree to which a program's performance benefits from delaying that expression. Second, we explain how to implement this metric as a dynamic profiler for an untyped scripting language.[2] After profiling a program on a typical input, the tool suggests where to add laziness annotations. Third, we present evidence that our profiler generates advice comparable to that of experts in the literature. For most of the programs we examined, our profiler requests the insertion of `delay`s at the same places as human experts; in a few instances, the profiler's suggestions achieve the same performance benefits with fewer laziness annotations.

The next section presents some motivating examples, and section 3 formally describes the notion of laziness potential. Section 4 presents our prototype profiler implementation, while section 5 demonstrates its effectiveness. Finally, the remaining sections conclude with related and future work.

2. Laziness Potential, the Intuition

In some instances, it is easy for a programmer to identify where laziness is beneficial. Other times, trying to find these spots in a program is challenging. In this section, we explain our method for finding such spots with a series of examples.

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

[2] These languages tend to support laziness annotations natively or with libraries and should benefit most from our work. Our work applies to typed languages as well; see section 5.

2.1 Criteria for Laziness

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 practical 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 1 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 $m - n = 999$ times and two of the values are used, giving it a ratio of $2/999$.

Profiling this example with our tool reports the following:

```
rng (n+1) m [ln 3]: 2/999 values used
```

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] 1 1000                     (5)
in (second lst) + (third lst)                   (6)
```

Here is some data returned by our profiler:

```
f n [ln 3]: 2/999 values used
rng f (n+1) m [ln 3]: 2/999 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 weight 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. Our profiler reports this information, too:

```
rng f (n+1) m [ln 3]: 2/999 values used
- delaying 997 unused avoids 2989 subvalues, wgt=2990
f n [ln 3]: 2/999 values used
- delaying 997 unused avoids 0 subvalues, wgt=1
```

A higher weight indicates a greater benefit from laziness. The exact calculation of laziness potential is discussed in section 3 but the important takeaway here is that the profiler now considers 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 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 nil? 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 1 1000)        (10)
in (second lst) + (third lst)                   (11)
```

Profiling this example reports the following initial data:

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

meaning the profiler proposes delaying only a call to *filter*. Intuitively, the recursive call to rng should be delayed as well, but it does not appear in the results because *filter* "uses" the entire list.

In response, our profiler iteratively:

1. simulates delaying the expression with the highest laziness potential, and then
2. recalculates usages to possibly uncover more unused values.

With this refinement, the profiler reports:

```
~~~~~~~~~~~~~~ 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]: 4/999 values used
  - delaying 995 unused avoids 983 subvalues, wgt=2984
  f n [ln 3]: 5/999 values used
  - delaying 994 unused avoids 0 subvalues, wgt=1
```

After round 0, the analysis simulates 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 repeats until there are no more unused expressions. The profiler then reports that delaying both the calls to *filter* and rng would benefit program performance.

2.2 A Complete Example

This subsection shows how a programmer can use our laziness-potential profiler to solve the n-queens problem. We compare our result to Chang's [3], who describes the problem like this:

> The *n*-queens problem makes an illustrative playground for advertising lazy programming. 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.

Thus, Chang separates the program into two independent components and defines one function to compute a solution:

$$\texttt{def}\ nqueens\ n = \texttt{first}\ (filter\ isValid\ all\_placements)$$

where

$$\begin{array}{l}\texttt{let}\ all\_placements = foldl\ process\_row\ \texttt{[nil]}\ (rng\ n)\\ \texttt{def}\ process\_row\ r\ qss\_so\_far =\\ \quad foldr\ (\lambda(qs\ new\_qss).\\ \qquad\quad (map\ (\lambda c.(r,c)::qs)\ (rng\ n))\ \texttt{@}\ new\_qss)\\ \qquad \texttt{nil}\ qss\_so\_far\end{array}$$

While *all_placements* generates a stream of all possible queen placements, *isValid* checks whether a placement is valid according to the rules of chess. The solution then composes these functions via *filter* and `first`.

As Chang explains, "the approach cleanly separates two distinct concerns: *all_placements* ignore[s] the rules of the puzzle, while *isValid* enforce[s] 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)." Accordingly, running *all_placements* n yields a list of lists of positions:

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

Each line represents one possible placement.

In a strict language, computing *all_placements* generates *all* possible placements. Adding laziness preserves the elegant solution and makes it efficient because it prevents unnecessary evaluation. The question is which parts should be delayed. As Chang [3] reports, naïvely switching all lists to lazy lists does not improve program performance. Chang then presents a static tool to aid programmers with the task of inserting additional laziness. For n-queens, with lazy lists as the initial source of laziness, the static analysis makes the key suggestion of adding laziness to *foldr*, which results in a significant speedup.

Without any such initial hints, our profiler suggests laziness at the same place in *foldr* and also suggests laziness in *filter*:

```
filter p? (rest lst) [ln 23]: 0/2 values used
- delaying 2 unused avoids 5242 subvalues, wgt=5243
foldr f base (rest lst) [ln 40]: 36/85 vals used
- delaying 49 unused avoids 1939 subvalues, wgt=1507
```

Figure 1 summarizes the running times for the 8-queens program implemented in Racket [9] and varying degrees of laziness. The figure reports that the strict version is slow and that adding the suggestions from Chang's static tool produces roughly a four-fold speedup. However, adding laziness annotations as suggested by our profiler produces a program that is roughly *twenty times faster* than the strict version. Chang's oversight is due to the "seed" laziness required by the static analysis. In order for the static tool to compute its suggestions, Chang recommends first converting lists to lazy lists, as strict programmers looking to add laziness commonly do. It turns out, however, that this inserts *too much laziness*. As evident from our profiler's recommendations, *only filter* needs additional laziness; the other lists should be evaluated eagerly.

Implementation Description	Time (ms)
no laziness	20245
static tool-based suggestions from [3]	4874
profiler-based suggestions	1057

Figure 1. Running times for an 8-queens program.

3. Laziness Potential, the Definition

As our informal presentation suggests, we consider *laziness potential* the key to determining where to insert effective laziness annotations in strict programs. This section formally defines the notion in two stages, using a small model.

3.1 Partially-labeled λ-calculus

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

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

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. Define a program to 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 ℓ labels to enable identification of the value's source expression(s). In the example $(\lambda x.x)\ \underline{((\lambda z.z)\ \lambda y.y)}$, $\lambda y.y$ is the result of evaluating both underlined expressions and thus must have two dynamic labels.

Evaluation contexts [7] specify the order of evaluation in our labeled λ-calculus. They are the standard by-value evaluation contexts, plus an additional labeled context:

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

These contexts dictate that reductions take place within a labeled expression only if it comes with a static and a 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:

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

The semantics satisfies a conventional well-definedness theorem.

Theorem 1 (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 [7], 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 [19]. Theorem 2 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 strips labels from programs.

Theorem 2 (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.

3.2 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:

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

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

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

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.

3.3 Usage Contexts

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

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

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."

3.4 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.

We extend our model with typical features. Specifically, we add constants, let, conditionals, primitive arithmetic and boolean operations, lists and list operations, and `delay` and `force` laziness constructs; see figures 2 and 3. The reductions are conventional [7]:

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

Figure 2. Syntax for an extended by-value language.

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

Figure 3. Semantics for an extended by-value language.

- 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.
- 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:

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

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 ellipses traverse all other expressions in a homomorphic manner:

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

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

3.5 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:

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

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

Intuitively, $\mathbb{V}$ inspects all *vstart* steps. In the last example from section 2.1, 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 ${}^{\boldsymbol{\ell}_1}(\mathit{filter}\ p?\ (\texttt{rest}\ lst))$ is a redex and $\mathbb{V}\ (\mathbb{T}\ P)\ \boldsymbol{\ell}_1 = \{\ell_1, \ldots, \ell_{500}\}$.

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

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

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

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:

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

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

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 $\ell_i \in \{\ell_1, \ldots, \ell_{500}\}$, $\mathbb{U}\ (\mathbb{T}\ P)\ \ell_i = 1$ for two values of i and is otherwise 0. Thus, $\mathbf{unused}\ (P, \boldsymbol{\ell}_1) = 2$.

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

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

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

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

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

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:

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

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

In our running example, $\mathbf{created}\ (P, \boldsymbol{\ell}_1)$ 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 $\mathbf{unused}\ (e, \boldsymbol{\ell}) \neq \emptyset$:

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

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

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 $\mathbf{LP}\ (P, \boldsymbol{\ell}_1) \approx 2500$.

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:

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

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 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 Chang's [3] 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 section 2.1, 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.

4. A Laziness Profiler

An implementation of a laziness profiler cannot use the definition of $\mathbf{LP}_i$ as a blueprint because it is highly impractical to re-run a modified program for every 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.

4.1 Usage Information Gathering

Table 1 describes the collected information. It also lists the analogous mathematical function from section 3.5. The $\mathcal{I}$ function instruments a labeled λ-calculus program to collect this information:

$$\mathcal{I}[\![x]\!] = x \qquad \boxed{\mathcal{I} : Exp \to Exp}$$
$$\mathcal{I}[\![\lambda x.e]\!] = \lambda x.\mathcal{I}[\![e]\!]$$
$$\mathcal{I}[\![e_1\ e_2]\!] = \mathcal{U}[\![\mathcal{I}[\![e_1]\!]]\!]\ \mathcal{A}[\![\mathcal{I}[\![e_2]\!]]\!]$$

$\mathcal{I}$ relies on two additional functions, $\mathcal{A}$ and $\mathcal{U}$, to instrument argument and usage positions, respectively. We defined $\mathcal{I}$ only for core λ-calculus expressions so far but in general the application of $\mathcal{A}$ exactly follows the labeling strategy from the previous section. In other words, a subexpression in a program is only instrumented as an "argument" 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:

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 ℓ_{hi} such that the set of values created while evaluating ℓ is the exclusive interval (ℓ, ℓ_{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 ℓ_{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 ℓ_{parent}, then ℓ_{used} is considered used while evaluating ℓ but *not* while evaluating ℓ_{parent}	

Table 1. Information collected by the profiler.

$$\begin{aligned}
\mathcal{A}[\![^{\boldsymbol{\ell}}e]\!] = {}& \texttt{let } \ell_{new} = \texttt{next-vnum } () \texttt{ in} \\
& \texttt{ALL-VNUMS} \cup\!= \{\ell_{new}\} \\
& \texttt{EXPR-VALS}[\boldsymbol{\ell}] \cup\!= \{\ell_{new}\} \\
& \texttt{push-ctxt-vnum } \ell_{new} \\
& \texttt{let } w^{\vec{\ell}} = {}^{\boldsymbol{\ell}}e \texttt{ in} \\
& \texttt{pop-ctxt-vnum } () \\
& \texttt{CHILD-VNUMS}[\ell_{new}] = \texttt{current-vnum } () \\
& \texttt{return } w^{\ell_{new},\vec{\ell}}
\end{aligned}$$

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

In these definitions, `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.

4.2 Post-Execution Analysis

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

$$\texttt{unused}\ \boldsymbol{\ell} =$$
$$\{\ell \mid \ell \in \text{EXPR-VALS}[\boldsymbol{\ell}],\ \ell \in \text{ALL-VNUMS},\ \text{USES}[\ell] = 0\}$$
$$\texttt{created}_v\ \ell =$$
$$\{\ell_{sub} \mid \ell < \ell_{sub} < \text{CHILD-VNUMS}[\ell],\ \ell_{sub} \in \text{ALL-VNUMS}\}$$
$$\texttt{created}\ \boldsymbol{\ell} = \bigcup_{\ell \in \texttt{unused}\ \boldsymbol{\ell}} \texttt{created}_v\ \ell$$
$$\texttt{LP}\ \boldsymbol{\ell} = \frac{|(\texttt{unused}\ \boldsymbol{\ell}) \cup (\texttt{created}\ \boldsymbol{\ell})|}{|(\texttt{unused}\ \boldsymbol{\ell}) \setminus (\texttt{created}\ \boldsymbol{\ell})|}$$

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

$$\begin{array}{l}\texttt{while}\ (\bigcup_{\boldsymbol{\ell} \in e} \texttt{unused}\ \boldsymbol{\ell}) \neq \emptyset :\\ \quad \boldsymbol{\ell} = \underset{\boldsymbol{\ell}' \in e, \texttt{unused}\ \boldsymbol{\ell}' \neq \emptyset}{\arg\max} (\texttt{LP}\ \boldsymbol{\ell}')\\ \quad \texttt{for}\ \ell \in (\texttt{unused}\ \boldsymbol{\ell}) : \texttt{erase}\ \ell\\ \quad \texttt{record-result}\ \boldsymbol{\ell}\end{array}$$

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`:

$$\begin{array}{l}\texttt{erase}\ \ell =\\ \quad \texttt{sub-uses-by-vnum}\ \ell\\ \quad \text{ALL-VNUMS} \setminus= \{\ell\}\\ \quad \texttt{for}\ \ell_{sub} \in (\texttt{created}_v\ \ell) : \texttt{erase}\ \ell_{sub}\\ \texttt{sub-uses-by-vnum}\ \ell =\\ \quad \texttt{for}\ (\ell_{used}, n) \in \text{USES-BY-VNUM}[\ell] : \text{USES}[\ell_{used}] -= n\end{array}$$

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 3. *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 3.5. 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.

Note that 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. □

4.3 Implementation

We have implemented a prototype of the profiler described in this section for most of the Racket language [9]. The profiler supports all the language constructs from section 3.4, 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.

5. 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. Secondarily, when our results differ from the experts, we turn to wall-clock measurements to further evaluate our profiler's output.

General Profiling Issues 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 proceeded as follows:

1. we implemented one of the expert examples as a strict program;
2. we then profiled this strict version in a representative context;
3. 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 [3].

We conducted all experiments in Racket, an untyped language. We ported all implementations verbatim as we found them, except for a few type-related discrepancies. Okasaki points out [17, 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 stream tail, while in an untyped language, this annotation is unneeded since the empty stream is a value. In general, laziness in typed languages requires a pairing of `delay`s and `force`s, while untyped languages are more flexible due to the recursive and idempotent nature of untyped `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	⊜
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	Ⓟ
implicit deque	double-ended version of implicit queue	Ⓟ
Other Functional Data Structures		
finger tree	Hinze and Paterson's general purpose data structure (related to implicit queues)	Ⓟ

Ⓟ = profiler-assisted implementation outperforms Okasaki's implementation

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

⊛ = mixed results

Table 3. Comparison of Okasaki's lazy data structures with analogous profiler-generated data structures.

5.1 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.

Table 3 describes several data structures of each kind and presents the results of our comparisons. For most examples, our profiler suggests laziness 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.

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

```
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]_rotation

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 4. Functional queue implementation without laziness.

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 which elements are removed. A record data definition, $(Q\ front\ len_f\ rear\ len_r)$, 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 4 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

```
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 5. Okasaki's lazy banker's queue.

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 4 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.

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,[3] replaces the eager lists with lazy streams and is shown in figure 5. 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:

```
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)
```

[3] Okasaki uses the banker's method to determine the queue's complexity.

```
def enq x (Q f len_f r len_r) = chk f len_f [(x::r)]_1 (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)]_2) (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)]_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 6. Profiler-assisted version of lazy banker's queue.

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 6 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.[4] 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}:

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

The graph in figure 7 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.

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

[4] 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 5.

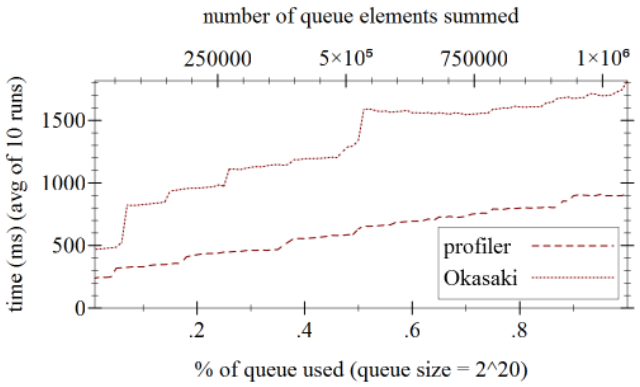

Figure 7. Summing elements of a banker's queue (lower is better).

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 4. Testing persistence with rotations in the banker's queue.

construct a queue such that a tl operation on the queue triggers the rotations suggested by the spikes in the graph of figure 7. 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:

$$\texttt{let}\ q = \mathit{drop}\ (\mathit{build}\ (2^{20}{-}1))\ (2^{19}{-}1) \qquad \text{(PERSIST:LAZY)}$$
$$\texttt{in}\ \mathit{time1} = \texttt{time}\ (\mathit{tl}\ q);\ \mathit{time2} = \texttt{time}\ (\mathit{tl}\ q)$$

Table 4 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.

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.

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: $(Q_p\ \mathit{front}_{cache}\ \mathit{front}\ \mathit{len}_f\ \mathit{rear}\ \mathit{len}_r)$. Figure 8 shows the implementation for Okasaki's physicist's queue.

Deriving the physicist's queue from the strict queue (in figure 4) involves more than just adding lazy annotations, so it belongs in the *insightful* category of data structure. Nevertheless, we follow the previously described steps to develop the alternative implementation in figure 9. 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.

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

def chk f' f lenf r lenr =
  if lenr ≤ lenf then chkw f' f lenf r lenr
  else let f'' = force f
       in chkw f'' $(f'' ++ rev r) (lenf + lenr) nil 0

def chkw nil f lenf r lenr = (Qp (force f) f lenf r lenr)
  | chkw f' f lenf r lenr = (Qp f' f lenf r lenr)

def hd (Qp nil _ _ _ _) = error
  | hd (Qp (x::f') f lenf r lenr) = x
def tl (Qp nil _ _ _ _) = error
  | tl (Qp (x::f') f lenf r lenr) =
    chkw f' $(rest (force 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 8. Okasaki's lazy physicists's queue.

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

def chk f' f lenf r lenr =
  if lenr ≤ lenf then chkw f' f lenf r lenr
  else chkw f (f ++ $(rev r)) (lenf + lenr) nil 0

def chkw nil f lenf r lenr = (Qp f f lenf r lenr)
  | chkw f' f lenf r lenr = (Qp f' f lenf r lenr)

def hd (Qp $nil _ _ _ _) = error
  | hd (Qp $(x::f') f lenf r lenr) = x
def tl (Qp $nil _ _ _ _) = error
  | tl (Qp $(x::f') $(y::f) lenf r lenr) =
    chkw f' 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 9. Profiler-assisted lazy physicists's queue.

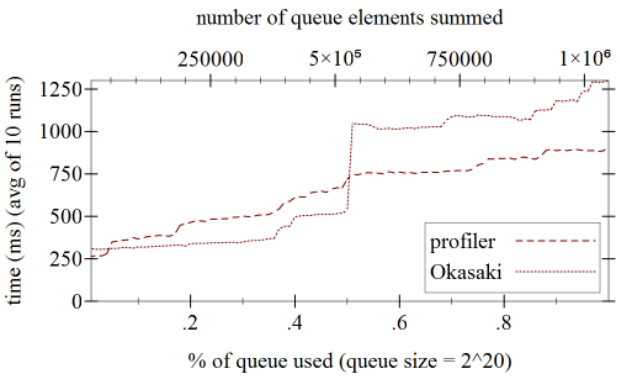

Figure 10. Summing physicist's queue elements (lower is better).

Figure 10 shows physicist's queue timings for the BUILD&SUM benchmark. 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.

Implicit Queues Implicit queue and deque data structures pose challenging obstacles for both designers and profilers. Hinze and Paterson's finger trees [10] 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 3.

Implicit queues rely on implicit recursive slowdown [17, 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 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 11 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

```
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 11. Implicit queue operations.

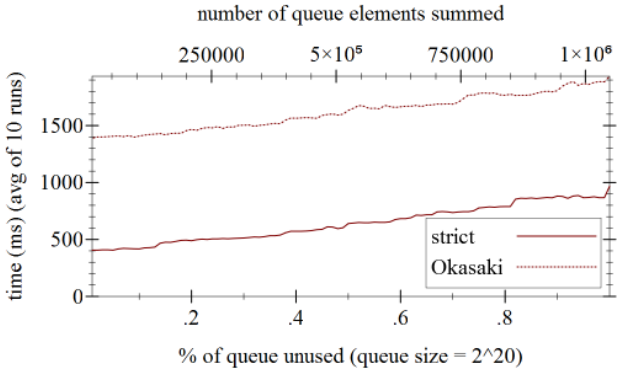

Figure 12. Summing elements of a implicit queue (lower is better).

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

To investigate this discrepancy, figure 12 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.

5.2 Monadic Parser Combinators

Our second suite of benchmarks concerns Parsec [13], 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.[5] In every case, the profiler suggested adding laziness at the exact spot identified by the Parsec paper.

5.3 Manipulating Game Trees

Our final benchmark is an AI game algorithm from Land of Lisp [2], which introduces programmers to functional Lisp 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.

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.

6. Related Work

Strictness Analysis A concept related to "use" is the denotational notion of strictness [4, 16], which is defined via $\bot$: 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 "use" captures.

Low-Utility Data Structures Xu et al. [22] present a tool that uses an object cost-benefit analysis to help programmers find inefficient parts of Java programs. While they analyze imperative, object-oriented programs instead of functional ones, they have goals similar to ours. However, a key difference lies in their somewhat indirect approach. Their goal is to find problematic spots in the program source, but they compute costs at the level of memory accesses and bytecode instructions, thus requiring an extra final step to interpret the results in terms of the source. It is not clear how the paper achieves this last step, which becomes even more complicated for a higher-order language. Our approach computes laziness potential directly in terms of a high-level semantics, from which the problem spots in a program become apparent.

7. Conclusion and Future Work

We introduce the notion of laziness potential, a metric that predicts which program expressions benefit from lazy evaluation. Our Racket profiler implements these calculations and guides programmers with the insertion of laziness annotations. An evaluation of our profiler automatically replicates, and in some cases improves on, the laziness in a range of real-world applications, demonstrating the potential usefulness of our approach.

Though we presented laziness potential for a by-value λ-calculus and implemented our profiler 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 [18] 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.

[5] We followed Real World Haskell [18] for guidance.

Acknowledgments

We thank Greg Morrisett for initial inspiration, Asumu Takikawa and J. Ian Johnson for reading drafts, and the reviewers for their suggestions. This work has been partly supported by NSF Infrastructure grant CI-ADDO-EN 0855140.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[2] C. Barski. *Land of Lisp*. No Starch Press, 2011.

[3] S. Chang. Laziness by need. In *Proc. 22nd ESOP*, pages 81–100, 2013.

[4] C. Clack and S. L. Peyton Jones. Strictness analysis — a practical approach. In *Proc. 2nd FPCA*, pages 35–49, 1985.

[5] R. Ennals and S. Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *Proc. 8th ICFP*, pages 287–298, 2003.

[6] K.-F. Faxén. Cheap eagerness: speculative evaluation in a lazy functional language. In *Proc. 5th ICFP*, pages 150–161, 2000.

[7] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[8] R. B. Findler, S. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Proc. 19th IFL*, pages 111–128, 2007.

[9] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2013-1, PLT Inc., 2013. `http://racket-lang.org/tr1/`.

[10] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. In *J. Funct. Program.*, pages 197–217, 2006.

[11] P. Hudak, J. Hughes, S. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. 3rd HOPL*, pages 12–1–12–55, 2007.

[12] J. Hughes. Why functional programming matters. *Comp. J.*, 32:98–107, 1989.

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

[14] J.-W. Maessen. Eager Haskell: resource-bounded execution yields efficient iteration. In *Proc. Haskell Workshop*, pages 38–50, 2002.

[15] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *Proc. 26th ECOOP*, pages 104–131, 2012.

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

[17] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[18] B. O'Sullivan, D. Stewart, and J. Goerzen. *Real World Haskell*. O'Reilly Media, 2008.

[19] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[20] K. E. Schauser and S. C. Goldstein. How much non-strictness do lenient programs require? In *Proc. 7th FPCA*, pages 216–225, 1995.

[21] P. Wadler. How to replace failure by a list of successes. In *Proc. 2nd FPCA*, pages 113–128, 1985.

[22] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *Proc. 31st PLDI*, pages 174–186, 2010.