Laziness by Need

Stephen Chang

August 3, 2012

Abstract

Lazy functional programming comes with many benefits that strict functional languages can simulate via lazy data constructors. For example, the use of lazy data structures can simplify the composition of components and improve the performance of a program. In recognition, ML and Scheme have supported lazy stream programming with `delay` and `force` for several decades. Unfortunately, the manual insertion of `delay` and `force` is tedious for the most part, challenging in a few instances, and hence truly error-prone.

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

1 Laziness in a Strict World

A lazy functional language naturally supports the construction of reusable components and their composition into reasonably efficient programs [19]. For example, the solution of a puzzle may consist of a generator that produces an easily-constructed stream of all *possible* solutions and a filter that extracts the desired number of *valid* solutions. Due to laziness, the generator does not really produce all possible solutions because the filter component explores only the needed portion of the stream. Put differently, lazy composition appears to naturally recover the desired degree of efficiency without imposing a contorted programming style.

Unfortunately, programming in a lazy language comes at a cost. A lazy language uses not only lazy data constructors but makes all functions lazy.

This pervasiveness of laziness makes it difficult to predict the behavior and time/space performance of lazy programs.

As several researchers noticed [2, 8, 22, 23, 30], most programs need only a small amount of laziness. In response, people have repeatedly proposed lazy programming in strict functional languages [1, 13, 27, 31, 35]. As a matter of fact, Scheme [29] and ML [3] have supported manual stream programming with `delay` and `force` for decades. Using `delay`, a programmer can easily turn an eager, Lisp-style list `cons`tructor into a lazy one [16]. Using `force`, a program retrieves the value from a delayed computation.

The problem is that the manual insertion of `delay` and `force` is tedious labor for the most part, challenging in a few instances, and hence truly error-prone. While it is almost always obvious which basic data constructors to delay, the insertion of one `delay` tends to require the insertion of additional `delay`s elsewhere to achieve the desired benefits. In the same vein, the insertion of `force` must also balance functional behavior and performance. On the one hand, a `force` is needed for every strict point in the language that may receive delayed computations; on the other hand, inserting too many `force`s may erode the intended performance benefits. Last but not least, the presence of higher-order functions hides laziness and can easily mislead programmers.

In this proposal, we present an alternative solution. Specifically, we introduce a transformation that assists programmers with the task of inserting `delay` and `force`. We imagine a programmer who wishes to create a lazy generator and starts using lazy constructs in the obvious places. Our transformation then inserts additional `delay`s and `force`s, based on the result of a static analysis, to achieve the desired lazy performance benefit. We have also implemented the transformation as a tool in Racket [15] and describe preliminary experiences with the tool.

This proposal is organized as follows. The second section introduces some motivating examples in typed and untyped languages. Section 3 describes the analysis-based program transformation and its properties. Section 4 presents a prototype implementation and section 5 describes real-world applications. Section 6 compares our approach with other attempts at taming laziness. Finally, section 7 describes additional practical extensions for the tool as well as a schedule for completing the research.

2 Motivating Examples

Three decades ago, Abelson and Sussman [1] presented lazy programming via `delay` and `force`. Indeed, their textbook implies that first-semester college students can comprehend and deploy these constructs. While `delay` and `force` are certainly easy to understand in isolation, Wadler et al. [35] point out fundamental flaws with Abelson and Sussman's seemingly straightforward approach. Similarly, our own programming and teaching experience suggests that injecting `delay` and `force` requires helpful hints.

A multitude of modern strict languages now admit manual lazy programming. None of these languages offer much help, however, in figuring out the right way to use these forms. To illustrate the problems, this section presents three examples in three distinct languages, typed and untyped:

1. The first one, in Racket [15], shows how conventional program reorganizations can eliminate the performance benefits of laziness without warning.

2. The second example, in Scala [26], demonstrates how laziness must propagate across function calls.

3. The third one, in OCaml [21], illustrates the difficulties of developing an idiomatic lazy n-queens algorithm in a strict language.

The use of different languages for the examples does not mean that each of the described problems is only relevant to a specific language. Rather, the problems of programming lazily in a strict language are universal across a multitude of languages.

2.1 Reorganizations Interfere with Laziness

Programming with `delay` and `force` occasionally confuses even the most experienced programmer and instructor. This subsection retells the story of a recent instance involving the director of my thesis committee.

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 is also useful for game developers as a single point of control, especially if they wish to experiment with the rules of the 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 complete game tree might roughly look like this:

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

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

;; 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]
;; Returns list of possible Moves from given GameState
(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)))
```

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 leaf node is created with the `GT-Leaf` constructor. Otherwise, the `GT-Node` constructor creates a game tree node 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.

Such an example is utilized in an upcoming programming book to illustrate functional programming. Initially, only a small game is implemented, so `Move` is defined as a strict constructor. As the book progresses, however, the game tree gradually becomes unwieldy as more features are added to the game. In response, the third argument of the `Move` structure is changed to be lazy, meaning the call to the `Move` constructor implicitly wraps the third argument with a `delay`. With the lazy `Move` constructor, the code above generates only the first node of a game tree.

In the process of preparing the book for typesetting, the author reorganized the code in a seemingly innocuous fashion to fit it within the margins of the book page:

```
;; calc-next-moves : GameState Player -> [ListOf Move]
;; Returns list of possible Moves from given GameState
(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))
```

Specifically, the underlined code above 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, it is easy for such a performance bug to go unnoticed because the program passes all unit tests. Only properly stress testing the code will catch such a mistake.

In contrast, our laziness transformation recognizes that the `new-gt` variable flows into the lazy position of the `Move` constructor, and in turn, inserts an appropriate `delay` around the construction of the new game tree.

2.2 Laziness Must Propagate

A 2009 blog post[1] illustrates a related tricky situation with the following Scala [26] example. In Scala, an argument to a method can be delayed by marking its type with the `=>` prefix:[2]

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

When the function is called, the second argument is not evaluated until its value is needed inside the function body. However, when we define a function that calls `foo`, like this:

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

we lose the benefit of delaying the second argument to `foo`. To recover it, we must delay the third argument to `bar`:

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

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

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

If there is a call to `bar` elsewhere, then the corresponding function must delay the appropriate argument as well. For programs with complex call graphs, the required delay points may be scattered throughout the program, making programmer errors likely. Our transformation is designed to help with just such situations.

2.3 Idiomatic Lazy Programming in a Strict Language

Puzzles such as the n-queens problem make up 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 another part for deciding whether a particular placement of n queens represents a solution to the puzzle. Given these two components, a solution algorithm is a one-line function definition:

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

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

The approach cleanly separates two distinct concerns. While `all` may thus 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).

The rest of the section explains how an OCaml [21] programmer may develop such a lazy algorithm. Here is `all`:

```
let all =
  foldl
    (fun r qss ->
      foldr
        (fun qs acc ->
          (map (fun c -> (r,c)::qs) (rng n)) @ acc)
        [] qss)
    [[]] (rng n)
```

Lists are represented with brackets, `[]` is the empty list, `rng n` is a list of numbers from 1 to n, `::` is infix notation for `cons`, and `@` is infix for append.

This straightforward expression gradually builds up all the possible placements by adding one coordinate at a time. The inner `foldr`, given a row r and a partially-built placement qs, duplicates the partial placement n times, adds to each copy a new coordinate with the row r and a (different) column c, and then appends all these new partial placements to the final list of all placements. The outer `foldl` repeats this n times, once per row. The result of evaluating `all` looks like this:

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

where each line represents one possible placement.

Since OCaml is strict, however, using `all` 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 instance, here is the timing for $n = 8$ queens using the Linux `time` command:[3]

```
real 3m15.546s   user 3m14.416s   sys 0m0.432s
```

To speed up the performance we add laziness to avoid unnecessary work. Following Abelson and Sussman, the running time of the strict program should improve if we replace all instances of the `cons` (`::`) above with its lazy variant, represented with $::_{lz}$ below, and insert `force`s at the proper places. In this setting, lazy `cons` is defined using OCaml's `Lazy` module and is `cons` with a `delay`ed rest list. For example, here is append (`@`) and `map` with lazy `cons` (we overload `[]` to also represent the lazy empty list):[4]

```
let rec (@) lst1 lst2 =
  match force lst1 with
    | [] -> lst2
    | x::_{lz}xs -> x::_{lz}delay (xs @ lst2)

let rec map f lst =
  match force lst with
    | [] -> []
    | x::_{lz}xs -> f x::_{lz}delay (map f xs)
```

[3]Run on an Intel i7-2600k processor machine with 16GB memory.

[4]OCaml's delaying construct is dubbed `lazy` but for clarity and consistency with the rest of the proposal we continue to use the name `delay`.

Running this program, however, surprises our imaginary lazy-strict programmer:

```
  real 3m27.474s   user 3m26.309s   sys 0m0.416s
```

With lazy `cons` and `force`, the program runs even slower than the strict version.

Part of the slowdown is due to the overhead of the `delay`-generated promises, but clearly, our program is not evaluating lists lazily enough. In other words, using lazy `cons` naïvely does not naturally generate the expected performance gains. Additional `delay`s are required, though it is not immediately obvious where to insert them and whether they need additional `force`s.

This step is precisely where our analysis-based refactoring transformation helps a programmer. In this particular case, our transformation would insert a `delay` in the `foldr` function:

```
let rec foldr f base lst =
  match force lst with
    | [] -> base
    | x::$_{lz}$xs -> f x (delay (foldr f base xs))
```

While this may be surprising, the (underlined) `delay` is needed because the recursive function call eventually flows to a lazy constructor position, specifically the lazy `cons` in the definition of append (`@`). Without this `delay`, the list is evaluated prematurely.

With this refactoring, and the appropriate insertion of `force`s, also via our transformation,[5] we see a dramatic improvement:

```
  real 0m3.379s   user 0m3.324s   sys 0m0.040s
```

Lazy programmers are already familiar with such benefits, but this example demonstrates that our refactoring transformation enables strict programmers to reap the same benefits as well.

3 Refactoring For Laziness

Our refactoring uses a whole-program analysis that infers where lazy data may flow and where additional laziness should be used. Once the analysis delivers results, the transformation uses it to insert `delay`s and `force`s.[6]

[5]The OCaml type checker would also identify the need for `force`s.

[6]We use Scheme's `force`, which consumes any kind of value and thaws any delayed value recursively.

Our analysis starts from a standard, textbook [25] formulation of a 0-CFA analysis. It is then extended with lazy forms and calculations of potential insertion points for `delay` and `force`.

Our analysis makes the assumption that during evaluation, the programmer does not want an unforced `delay` value to appear in a strict position. Thus, if the analysis discovers the potential for an unforced `delay` in the function position of an application, for example, we assume that the programmer forgot a `force` and analyze the potential functional calls anyway. This makes our analysis quite conservative in that `delay`s are approximated as both forced and unforced but it ensures that the computed control flow is not affected by any potential laziness-related errors. On the technical side, implicit forcing also enables the proof of a safety theorem for the transformation.

In addition to approximating evaluation of a program, our analysis also extracts the necessary information to insert additional `delay`s and `force`s. Specifically, the analysis computes three additional sets:

1. the set of function arguments flowing to lazy positions in the program;
2. the set of function arguments that flow to strict positions, a set that is not necessarily disjoint from the first one;
3. and the set of program locations where a `delay`ed value may show up—both those manually inserted by the programmer and those suggested by the analysis.

Our program transformation uses the analysis result to insert additional `delay`s and `force`s. Specifically, a function argument is delayed if it flows to a lazy position but not a strict position. In addition, a `force` is inserted both at strict positions where a programmer inserted `delay` may appear, and strict positions where `delay`s inserted by the analysis may appear. This ensures that `delay`s will be properly forced when its value is required.

Our refactoring for laziness changes the semantics of programs. For example, non-terminating programs may be transformed into terminating ones or exceptions may be delayed indefinitely. Nevertheless, we can still prove interesting properties about our analysis. First, we establish soundness of the analysis using the reduction-based technique of Flanagan and Felleisen [14]. Informally, the soundness theorem says that for any subexpression in the program that reduces to a value, the analysis correctly predicts that value.

Second, we show that our program transformation is safe, meaning that unforced promises—both those inserted by the programmer and by the analysis—cannot cause exceptions. We show safety for the transformation by first establishing the difference between an analysis result satisfying a given program, and an analysis result satisfying that program after transformation. We then use this difference, plus the soundness theorem, to prove the safety theorem.

4 A Prototype Implementation

To evaluate our idea, we have implemented refactoring for laziness as a tool for the Racket language [15]. Our tool is a plugin for the DrRacket IDE [12]. It uses laziness analysis to automatically insert `delay` and `force` expressions as needed, with graphical justification.

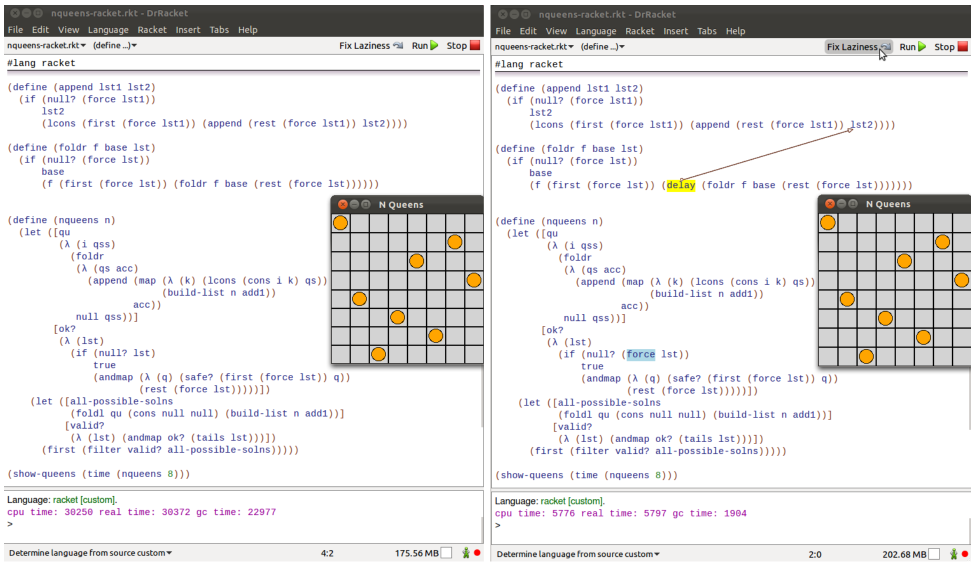
Our laziness analysis implementation consists of two stages. The first stage traverses a program and generates a set of constraints based on the analysis rules. The second stage then finds the least solution to the constraints via a conventional worklist algorithm [25].

Our prototype tool uses the result of the analysis and the transformation function to insert additional `delay`s and `force`s. Using our prototype we evaluated a number of examples. An illustrative example is the n-queens program from section 2. The left-hand side of figure 1 shows the program in Racket. It uses lazy `cons`,[7] which requires `force`s at certain strict positions. In addition, the call to `nqueens` is wrapped with `time`, the Racket timing function, and `show-queens`, a function from the *How to Design Programs* GUI libraries [11], to display a graphical depiction of the answer. Despite the use of lazy `cons`, the program does not behave lazily, as evident from the timing information, which is comparable to an eager version of the same program (not shown).[3]

The right-hand side of figure 1 shows the program after our tool has applied the laziness transformation. When the tool is activated, via the "Fix Laziness" button, it

1. computes an analysis result for the given program,
2. inserts the computed calls to `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.

[7]Though the exact `lcons` form is not available in plain Racket, we simulate it with a macro. The macro wraps a `delay` around the second argument of a regular `cons`.


nqueens-racket.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
nqueens-racket.rkt (define ...)
Fix Laziness Run Stop
N Queens
Determine language from source custom
4:2
175.56 MB
nqueens-racket.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
nqueens-racket.rkt (define ...)
Fix Laziness Run Stop
N Queens
Determine language from source custom
2:0
202.68 MB

```
#lang racket

(define (append lst1 lst2)
  (if (null? (force lst1))
      lst2
      (lcons (first (force lst1)) (append (rest (force lst1)) lst2))))

(define (foldr f base lst)
  (if (null? (force lst))
      base
      (f (first (force lst)) (foldr f base (rest (force lst))))))


(define (nqueens n)
  (let ([qu
         (λ (i qss)
           (foldr
            (λ (qs acc)
              (append (map (λ (k) (lcons (cons i k) qs))
                           (build-list n add1))
                      acc))
            null qss))]
        [ok?
         (λ (lst)
           (if (null? lst)
               true
               (andmap (λ (q) (safe? (first (force lst)) q))
                       (rest (force lst)))))])
    (let ([all-possible-solns
           (foldl qu (cons null null) (build-list n add1))]
          [valid?
           (λ (lst) (andmap ok? (tails lst)))])
      (first (filter valid? all-possible-solns)))))

(show-queens (time (nqueens 8)))
```

```
Language: racket [custom].
cpu time: 30250 real time: 30372 gc time: 22977
>
```

```
#lang racket

(define (append lst1 lst2)
  (if (null? (force lst1))
      lst2
      (lcons (first (force lst1)) (append (rest (force lst1)) lst2))))

(define (foldr f base lst)
  (if (null? (force lst))
      base
      (f (first (force lst)) (delay (foldr f base (rest (force lst)))))))


(define (nqueens n)
  (let ([qu
         (λ (i qss)
           (foldr
            (λ (qs acc)
              (append (map (λ (k) (lcons (cons i k) qs))
                           (build-list n add1))
                      acc))
            null qss))]
        [ok?
         (λ (lst)
           (if (null? (force lst))
               true
               (andmap (λ (q) (safe? (first (force lst)) q))
                       (rest (force lst)))))])
    (let ([all-possible-solns
           (foldl qu (cons null null) (build-list n add1))]
          [valid?
           (λ (lst) (andmap ok? (tails lst)))])
      (first (filter valid? all-possible-solns)))))

(show-queens (time (nqueens 8)))
```

```
Language: racket [custom].
cpu time: 5776 real time: 5797 gc time: 1904
>
```

Figure 1: Evaluating n-queens (l) in Racket with lazy `cons` only, (r) after applying laziness transformations.

Running the transformed program exhibits improved performance, also seen in the right-hand side of figure 1. The figure also shows an inserted `force`. This force is unneeded but the analysis is conservative, so the tool may insert more `force`s than necessary.

5 Laziness in the Large

To further investigate the practicality of our approach, we examined the Racket code base and some user-contributed packages. We found several erroneous attempts—with respect to performance—at adding laziness manually, and we verified that our tool would have prevented many such errors.[8] 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 [7] that ship with Racket allow students to write contracts for some data structures. The implementation of these contracts utilizes laziness and is based on Findler et al.'s lazy contract checking [13]. The contracts are built via a constructor that is lazy in a few positions. Ad-

[8]Our tool prototype currently only works with a restricted core syntax so the examples were first translated to work with this restricted syntax.

ditionally, the language implements several specialized contract constructors for various specific data structures. Since these specialized contract constructors are ordinary strict functions, the programmer must manually delay the appropriate arguments to these functions to preserve the intended lazy behavior. In other words, the programmer must propagate the laziness outwards, similar to the Scala example from section 2. Thus, the addition of a small amount of laziness to the program, in the form of a lazy constructor, requires several more delay points scattered all throughout the program. Adding these `delay`s becomes tedious and error-prone as the program grows in complexity and not surprisingly, a few were left out. Applying our tool identified the missing `delay`s, which the author of the code has confirmed and corrected with commits to the code repository.

A second example concerns queues and deques based on implicit recursive slowdown [27, Chapter 11], implemented in Typed Racket [32]. In this library [28], laziness enables fast amortized operations and simplifies the implementation. The authors of the code inadvertently introduced several performance bugs, however. The essence of the problem is represented in the following 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⟩)
```

In the definition, `elem` is to be enqueued in `dq`, where a `Deque` consists of a lazy part and a strict part. The problem occurs in one execution path, where the lazy part is extracted, forced, and then separated into two additional pieces. 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 also fixes the performance bug. For a reasonably standard benchmark, the fix reduced the running time by an order of magnitude. A similar fix in the queue data structure produced similar improvement. The authors of the code acknowledged the bug and have merged our fix into the code repository.

6 Related Work

6.1 Strictness Analysis

The idea of combining strict and lazy evaluation is old. The most well-known techniques are strictness analysis [5, 6, 17, 18, 20, 24, 34, 36] and related static approaches [4, 9, 10].

Researchers have also explored the idea of combining strict and lazy evaluation via "stingy" strategies [33]. Aditya et al. [2], Maessen [22], and Ennals and Peyton Jones [8] introduced similar strict-by-default mechanisms. In all these systems, programs are evaluated eagerly but various runtime heuristics determine when to suspend the computation to preserve lazy semantics.

While these research efforts and our work share the goal of combining the best of both strict and lazy evaluation, the former aims to completely preserve a lazy semantics. Our work starts with a strict programming language, and our goal is to inject laziness on a by-need basis.

Starting with a strict language eliminates many disadvantages of lazy evaluation. In particular space and time consumption do not pose as much of a problem in strict languages. In addition, since empirical studies have shown that in many cases, most promises in lazy languages are not needed [8, 23, 30], it seems appropriate to start without laziness and to introduce lazy behavior with just a few `delay`s.

6.2 Laziness in a Strict Language

Sheard [31] shares the vision of a strict language that is also practical for programming lazily. However, while his language does not require explicit `force`s, the programmer must manually insert all required `delay` annotations. In addition, the language does not calculate where `force`s are required and instead inserts them everywhere. Naturally his method of `force` insertion produces the correct results, but it inevitably inserts too many `force`s.

6.3 Odd vs Even-style Laziness

Wadler et al. [35] point 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 a definition of `map` from their paper:

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

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

The code, in our model language, uses `lcons` but lacks `force`. In the odd-style, the delayed portion of a list is forced as soon as it is extracted:

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

However, promises should not be forced until needed, at the language's strictness points, so one should not be surprised that the premature forcing of promises in the odd-style 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 how our transformation inserts `force`s, a pattern dubbed the "even" style by Wadler et al. Running the `map` example through our tool suggests a forcing pattern consistent with the even-style and thus our refactored program cannot suffer from any odd-style errors:

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

In the resulting program, the (potential) promise is not forced unless its value is required, indicated by the underlines above.[9]

We note a limitation of our tool in that it will not suggest to switch an odd-style program to the even-style. Adding this functionality to our tool would require a flow-sensitive analysis and is interesting future work.

7 Future Work and Timeline

I have already formulated the analysis rules and the program transformation, and I have established the key theorems mentioned in section 3 [pages 9-10]. Furthermore, I have implemented a prototype of a refactoring tool that

[9]Wadler et al.'s solution differs slightly due to their use of a typed language, but the essence is the same.

applies the transformation to a subset of the Racket language. In addition, I have tested the tool with several examples, including some from the Racket code base. This work represents a first step in the exploration of laziness refactoring. It demonstrates the theoretical and practical feasibility of our novel approach to laziness in a strict context. This first effort also suggests several directions of additional research, mostly to ensure its full practicality for core languages. I intend to pursue two different directions.

First, I plan to extend my system with types to show that our tool is useful in a typed setting as well. Specifically, I will show that the results of our analysis can be expressed with a type system in the spirit of OCaml and Scala. Doing so will require a different `delay` and `force` semantics. I expect to spend one semester on this work, completing it by January 2013.

Second, I also plan to extend the model with features that are ubiquitous in strict languages with support for laziness. Specifically I plan to add mutable state and pattern matching to my model. I expect to spend one semester working on this, completing it by June 2013.

If I fail in any of the above two tasks, I intend to explore alternative paths. First, I will explore the need for improved precision in my analysis, which makes several conservative approximations concerning where to insert `delay`s and `force`s. There exist additional applications of our strategy, however, that require a more accurate analysis. As a second alternative, I will improve on the usability of the tool by devising a way to express the semantic difference between a program, before and after applying the transformation. In addition to a typed model and an effectful language, a validation of practicality also calls for a modular analysis and a library-sensitive refactoring transformation. Unfortunately, this kind of challenge is a historically old one that compiler writers have encountered in different guises over the past few decades. I expect to deal with this challenge after completing my dissertation, though if my types research succeeds faster than expected, I may still deal with the question of a modular analysis.

Finally, I will write my dissertation during the fall semester of 2013 and plan to defend December 2013.

References

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

[2] S. Aditya, Arvind, L. Augustsson, J.-W. Maessen, and R. S. Nikhil.

Semantics of pH: A parellel dialect of Haskell. In *Proc. 1995 Workshop on Haskell*, pages 34–49, 1995.

[3] A. Appel, M. Blume, E. Gansner, L. George, L. Huelsbergen, D. MacQueen, J. Reppy, and Z. Shao. *Standard ML of New Jersey User's Guide*, 1997.

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

[5] G. L. Burn, C. L. Hankin, and S. Abramsky. The theory of strictness analysis for higher order functions. *Science of Computer Programming*, 7:42–62, 1986.

[6] C. Clack and S. L. Peyton Jones. Strictness analysis—a practical approach. In *Proc. 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 35–49, 1985.

[7] M. Crestani and M. Sperber. Experience report: growing programming languages for beginning students. In *Proc. 15th International Conference on Functional Programming*, pages 229–234, 2010.

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

[9] K.-F. Faxén. Optimizing lazy functional programs using flow inference. In *Proc. 2nd International Symposium on Static Analysis*, pages 136–153, 1995.

[10] K.-F. Faxén. Cheap eagerness: speculative evaluation in a lazy functional language. In *Proc. 5th International Conference on Functional Programming*, pages 150–161, 2000.

[11] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs.* MIT Press, 2002.

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

[13] R. B. Findler, S.-y. Guo, and A. Rogers. Lazy contract checking for immutable data structures. In *Proc. Symposium on Implementation and Application of Functional Languages*, pages 111–128, 2007.

[14] C. Flanagan and M. Felleisen. Modular and polymorphic set-based analysis: Theory and practice. Technical Report TR96-266, Rice University, 1996.

[15] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2012-1, PLT Inc., 2012. `http://racket-lang.org/tr1/`.

[16] D. Friedman and D. Wise. Cons should not evaluate its arguments. In *International Conference on Automata, Languages, and Programming*, pages 257–281, 1976.

[17] P. Hudak and J. Young. Higher-order strictness analysis in untyped lambda calculus. In *Proc. 13th Symposium on Principles of Programming Languages*, pages 97–109, 1986.

[18] J. Hughes. Strictness detection in non-flat domains. In *Proc. Workshop Programs as Data Objects*, pages 112–135, 1985.

[19] J. Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[20] S. P. Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Functional Programming*, pages 201–220, 1993.

[21] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system, release 3.12, Documentation and users manual.* INRIA, July 2011.

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

[23] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *Proc. 26th European Conference on Object-Oriented Programming*, 2012.

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

[25] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 2005.

[26] M. Odersky. *The Scala Language Specification, Version 2.9.* EPFL, May 2011.

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

[28] H. Prashanth K R and S. Tobin-Hochstadt. Functional data structures for Typed Racket. In *Proc. Workshop on Scheme and Functional Programming*, 2010.

[29] J. Rees and W. Clinger, editors. *Revised^3 Report on the Algorithmic Language Scheme.* ACM SIGPLAN Notices, December 1986.

[30] K. E. Schauser and S. C. Goldstein. How much non-strictness do lenient programs require? In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1995.

[31] T. Sheard. A pure language with default strict evaluation order and explicit laziness. In *2003 Workshop on Haskell: Discussion of New Ideas session*, 2003.

[32] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proc. 35th Symposium on Principles of Programming Languages*, pages 395–406, 2008.

[33] C. von Dorrien. *Stingy Evaluation.* Licentiate thesis, Chalmers University Of Technology, 1989.

[34] P. Wadler and J. Hughes. Projections for strictness analysis. In *Proc. 1987 Conference on Functional Programming Languages and Computer Architecture*, pages 385–407, 1987.

[35] P. Wadler, W. Taha, and D. MacQueen. How to add laziness to a strict language, without even being odd. In *Proc. 1998 Workshop on Standard ML*, 1998.

[36] S. Wray. *Implementation and programming techniques for functional languages.* PhD thesis, University of Cambridge, 1986.