Laziness by Need

Stephen Chang

Northeastern University
stchang@ccs.neu.edu

Abstract. Lazy functional programming has many benefits that strict functional languages can simulate via lazy data constructors. In recognition, ML, Scheme, and other strict functional languages have supported lazy stream programming with `delay` and `force` for several decades. Unfortunately, the manual insertion of `delay` and `force` can be tedious and error-prone.

We present a semantics-based refactoring that helps strict programmers manage manual lazy programming. The refactoring uses a static analysis to identify where additional `delay`s and `force`s might be needed to achieve the desired simplification and performance benefits, once the programmer has added the initial lazy data constructors. The paper 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 [12]. For example, the solution to a puzzle may consist of a generator that produces an easily-constructed stream of all *possible* solutions and a filter that extracts the desired *valid* solutions. Due to laziness, only a portion of the possible solutions are explored. Put differently, lazy composition appears to naturally recover the desired degree of efficiency without imposing a contorted programming style.

Unfortunately, programming in a lazy language comes at a cost. Not only are data constructors lazy, but all functions are as well. This pervasiveness of laziness makes it difficult to predict the behavior and time/space performance of lazy programs. As several researchers noticed [2, 6, 15, 16, 23], however, most programs need only a small amount of laziness. In response, people have repeatedly proposed lazy programming in strict functional languages [1, 8, 20, 25, 27]. In fact, Scheme [22] and ML [3] have supported manual stream programming with `delay` and `force` for decades. Using `delay` and macros, a programmer can easily turn an eager, Lisp-style list `cons`tructor into a lazy one [11], while `force` retrieves the value from a delayed computation.

However, merely switching from eager constructors to lazy ones is often not enough to achieve the performance benefits of laziness. The insertion of one `delay` tends to require additional `delay`s elsewhere in the program to achieve the desired lazy behavior. Since these additional `delay` insertions depend on

the value flow of the program, it can be difficult to determine where to insert them, especially in the presence of higher-order functions. In short, manual lazy programming is challenging and error-prone.

In response, we introduce a static analysis-based refactoring that assists programmers with the task of inserting `delay`s and accompanying `force`s. 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 to achieve the desired lazy performance benefit.

The paper is organized as follows. The second section introduces some motivating examples. Section 3 presents the analysis-based program transformation, and section 4 argues its correctness. Section 5 sketches a prototype implementation, and section 6 describes real-world applications. Section 7 compares our approach with other attempts at taming laziness. Finally, section 8 lists some ideas for future work.

2 Motivating Examples

Nearly every modern strict programming language supports laziness, either via `delay` and `force`, or in the form of a streams or other lazy data structure library. 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. The first one, in Racket [10], shows how conventional program reorganizations can eliminate the performance benefits of laziness without warning. The second, in Scala [19], demonstrates how laziness propagates across function calls. The third example illustrates the difficulties of developing an idiomatic lazy n-queens algorithm in a strict language like OCaml [14]. That is, the problems of programming lazily in a strict language are universal across many languages.

2.1 Reorganizations Interfere with Laziness

Using `delay` and `force` occasionally confuses even the most experienced programmers. This subsection retells a recent story involving a senior Racket developer. A game tree is a data structure representing all possible sequences of moves in a game. It is frequently employed in AI algorithms to calculate an optimal next move, and it is also useful for game developers wishing to experiment with the rules of a game. For anything but the simplest games, however, the multitude of available moves at each game state results in an unwieldy or even infinite game tree. Thus, laziness is frequently utilized to manage such trees.

The Racket code to generate a game tree might 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]
(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 `GT-Leaf` node is created. Otherwise, a `GT-Node` is created with the current game state, the current player, and a list of moves from the given game state. The `calc-next-moves` function creates a list of `Move` structures, where each move contains a new game tree starting from the game state resulting from the move.

An upcoming, Racket-based programming book utilizes such a game tree. Initially, only a small game is implemented, so `Move` is defined as a strict constructor. As the book progresses, however, the game tree becomes unwieldy as more features are added to the game. In response, the third argument of the `Move` structure is changed to be lazy, meaning the call to the `Move` constructor implicitly wraps the third argument with a `delay`.[1] With the lazy `Move` constructor, the code above generates only the first node of a game tree.

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

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

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, such a performance bug is easily unnoticed because the program still passes all unit tests.

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

[1] Specifically, `Move` becomes a macro that expands to a private constructor call where the third argument is delayed. This is a common idiom in Lisp-like languages [11].

2.2 Laziness Must Propagate

A 2009 blog post[2] illustrates a related tricky situation in the following Scala [19] example. Scala delays method arguments whose type is marked with `=>`, as in:[3]

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

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

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

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

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

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

2.3 Idiomatic Lazy Programming in a Strict Language

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. Given these two components, a one-line function calculates a solution:

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

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

The approach cleanly separates two distinct concerns. While `all_placements` may ignore the rules of the puzzle, it is the task of `isValid` to enforce them. If the components were large, two different programmers could tackle them in parallel. All they would have to agree on is the representation of queen placements, for which we choose a list of board coordinates (r, c). The rest of the section explains how an OCaml [14] programmer may develop such a lazy algorithm. Here is `all_placements`: :

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

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

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

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

Brackets denote lists, `rng n` is `[1...n]`, `::` is infix `cons`, and `@` is infix append. All possible placements are generated by adding one coordinate at a time. The `process_row` function, given a row r and a list of placements *qss_so_far*, duplicates each placement in *qss_so_far* n times, adding to each copy a new coordinate of r with a different column c, and then appends all these new placements to the final list of all placements. The `process_row` function is called n times, once per row. The result of evaluating `all_placements` 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_placements` with the `nqueens` function from earlier generates all possible placements before testing each one of them for validity. This computation is obviously time consuming and performs far more work than necessary. For instance, here is the timing for $n = 8$ queens:[4]

```
real 0m52.122s   user 0m51.399s   sys 0m0.468s
```

If the programmer switches to lazy lists to represent `all_placements`, then only a portion of the possible placements should be explored. Specifically, all instances of `cons` (`::`) are replaced with its lazy variant, represented with $::_{lz}$ below. In this setting, lazy `cons` is defined using OCaml's `Lazy` module and is `cons` with a delayed rest list. It is also necessary to add `force`s where appropriate.[5] For example, here is append (`@`) and `map` with lazy `cons` (`[]` also represents the empty lazy list):[6]

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

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

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

```
real 1m3.720s   user 1m3.072s   sys 0m0.352s
```

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

[4] Run on an Intel i7-2600k, 16GB memory machine using the Linux `time` command.

[5] "Appropriate" here means we avoid Wadler et al.'s [27] "odd" errors.

[6] OCaml's delaying construct is `lazy` but for clarity and consistency with the rest of the paper we continue to use `delay`. Also, in ML languages, the `delay` is explicit.

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

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

```
real 0m3.103s   user 0m3.068s   sys 0m0.024s
```

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

3 Refactoring For Laziness

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

3.1 Language Syntax

Our starting point is an untyped[7] functional core language. The language is *strict* and uses a standard expression notation:

$$e \in \mathit{Exp} = n \mid b \mid x \mid \lambda(x \ldots).e \mid e\ e \ldots \mid o\ e\ e \mid \texttt{zero?}\ e \mid \texttt{not}\ e \mid \texttt{if}\ e\ e\ e$$
$$\mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{null} \mid \texttt{cons}\ e\ e \mid \texttt{first}\ e \mid \texttt{rest}\ e \mid \texttt{null?}\ e$$
$$n \in \mathbb{Z},\quad b \in \mathit{Bool} = \texttt{true} \mid \texttt{false},\quad x \in \mathit{Var},\quad o \in \mathit{Op} = + \mid - \mid * \mid / \mid < \mid > \mid = \mid \texttt{or} \mid \texttt{and}$$

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

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

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

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

[7] Standard type systems cannot adequately express the flow of laziness and thus cannot solve the `delay`-insertion problems from section 2. A type error can signal a missing `force`, but a type system will not suggest where to add performance-related `delay`s.

3.2 Analysis Step 1: 0-CFA

Our initial analysis is based on 0-CFA [13, 24, 26]. The analysis assumes that each subexpression has a unique label ℓ, also drawn from *Var*, but that the set of labels and the set of variables in a program are disjoint. The analysis computes an abstract environment $\widehat{\rho}$ that maps elements of *Var* to sets of abstract values:

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

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

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

$$\widehat{n} = \texttt{val} \qquad \widehat{b} = \texttt{val} \qquad \widehat{\texttt{null}} = \texttt{val} \qquad \boxed{\widehat{\cdot} : \mathit{Val} \to \widehat{\mathit{Val}}}$$

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

We present our analysis with a standard [18], constraints-based specification, where notation $\widehat{\rho} \models p$ means $\widehat{\rho}$ is an acceptable approximation of program p. Figures 1 and 2 show the analysis for programs and expressions, respectively.

The [*prog*] rule specifies that environment $\widehat{\rho}$ satisfies program $p = d \ldots e$ if it satisfies all definitions $d \ldots$ as well as the expression e in the program. The [*def*] rule says that $\widehat{\rho}$ satisfies a definition if the corresponding abstract λ-value is included for variable f in $\widehat{\rho}$, and if $\widehat{\rho}$ satisfies the function body as well.

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

$$\widehat{\rho} \models d \ldots e \text{ iff} \quad \widehat{\rho} \models_d d \ \wedge\ \ldots\ \wedge\ \widehat{\rho} \models_e e \qquad [\mathit{prog}]$$

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

Fig. 1. 0-CFA analysis on programs

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

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

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

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

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

$$\widehat{\rho} \models_e (\texttt{let}\ x = e_1^{\ell_1}\ \texttt{in}\ e_0^{\ell_0})^{\ell} \text{ iff} \quad [let]$$
$$\widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x) \ \wedge$$
$$\widehat{\rho} \models_e e_0^{\ell_0} \ \wedge \ \widehat{\rho}(\ell_0) \subseteq \widehat{\rho}(\ell)$$

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

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

$$\widehat{\rho} \models_e (\texttt{not}\ e_1^{\ell_1})^{\ell} \text{ iff} \quad [not]$$
$$\widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \texttt{val} \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{if}\ e_1^{\ell_1}\ e_2^{\ell_2}\ e_3^{\ell_3})^{\ell} \text{ iff} \quad [if]$$
$$\widehat{\rho} \models_e e_1^{\ell_1} \ \wedge \ \widehat{\rho} \models_e e_2^{\ell_2} \ \wedge \ \widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell)$$
$$\wedge \ \widehat{\rho} \models_e e_3^{\ell_3} \ \wedge \ \widehat{\rho}(\ell_3) \subseteq \widehat{\rho}(\ell)$$

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

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

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

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

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

Fig. 2. Step 1: 0-CFA analysis on expressions

expression. The [*cons*] rule for an ℓ-labeled, eager `cons` requires that $\widehat{\rho}$ satisfy both arguments and that a corresponding abstract `cons` value be in $\widehat{\rho}(\ell)$. Finally, the [*first*] and [*rest*] rules require satisfiability of their arguments and that the appropriate piece of any `cons` arguments be a result of the entire expression.

3.3 Analysis Step 2: Adding `delay` and `force`

Next we extend our language and analysis with lazy forms:

$$e \in Exp = \ldots \mid \texttt{delay}\ e \mid \texttt{force}\ e \mid \texttt{lcons}\ e\ e$$
$$\text{where } \texttt{lcons}\ e_1\ e_2 \stackrel{df}{=} \texttt{cons}\ e_1\ (\texttt{delay}\ e_2)$$

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

We add promises and lazy lists to the sets of values and abstract values, and $\widehat{\cdot}$ is similarly extended. The abstract representation of a `delay` replaces the

labeled delayed expression with just the label and the abstract `lcons` is similar.

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

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

$$\ldots \quad \widehat{\texttt{delay}\ e^{\ell}} = \texttt{delay}\ \ell \qquad \widehat{\texttt{lcons}\ v_1^{\ell_1}\ e_2^{\ell_2}} = \texttt{lcons}\ \ell_1\ \ell_2 \qquad \boxed{\widehat{\cdot} : \mathit{Val} \to \widehat{\mathit{Val}}}$$

Figure 3 presents the new and extended analysis rules. The [*delay*] rule specifies that for an ℓ-labeled `delay`, the corresponding abstract `delay` must be in $\widehat{\rho}(\ell)$ and $\widehat{\rho}$ must satisfy the delayed subexpression. In addition, the values of the delayed subexpression must also be in $\widehat{\rho}(\ell)$. This means that the analysis approximates evaluation of a promise with both a promise and the result of forcing that promise. We discuss the rationale for this constraint below. The [*force*] rule says that $\widehat{\rho}$ must satisfy the argument and that non-`delay` arguments are propagated to the outer ℓ label. Since the [*delay*] rule already approximates evaluation of the delayed expression, the [*force*] rule does not have any such constraints.

We also add a rule for `lcons` and extend the [*first*] and [*rest*] rules to handle `lcons` values. The [*lcons*] rule requires that $\widehat{\rho}$ satisfy the arguments and requires a corresponding abstract `lcons` at the expressions's ℓ label. The [*first*] rule handles `lcons` values just like `cons` values. For the [*rest*] rule, a `delay` with the `lcons`'s second component is a possible result of the expression. Just like the [*delay*] rule, the [*rest*] rule assumes that the lazy component of the `lcons` is both forced and unforced, and thus there is another constraint that propagates the values of the (undelayed) second component to the outer label.

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

$$\widehat{\rho} \models_e (\texttt{delay}\ e_1^{\ell_1})^{\ell} \text{ iff} \qquad [\mathit{delay}]$$
$$(\texttt{delay}\ \ell_1) \in \widehat{\rho}(\ell)\ \wedge\ \widehat{\rho} \models_e e_1^{\ell_1}\ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{force}\ e_1^{\ell_1})^{\ell} \text{ iff} \qquad [\mathit{force}]$$
$$\widehat{\rho} \models_e e_1^{\ell_1}\ \wedge\ (\forall \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay} : \widehat{v} \in \widehat{\rho}(\ell))$$

$$\widehat{\rho} \models_e (\texttt{lcons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell} \text{ iff} \qquad [\mathit{lcons}]$$
$$\widehat{\rho} \models_e e_1^{\ell_1}\ \wedge\ \widehat{\rho} \models_e e_2^{\ell_2}\ \wedge\ (\texttt{lcons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell)$$

$$\widehat{\rho} \models_e (\texttt{first}\ e_1^{\ell_1})^{\ell} \text{ iff} \quad \ldots\ \wedge\ [\mathit{first}]$$
$$(\forall(\texttt{lcons}\ \ell_2\ \_) \in \widehat{\rho}(\ell_1) :$$
$$\widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$

$$\widehat{\rho} \models_e (\texttt{rest}\ e_1^{\ell_1})^{\ell} \text{ iff} \quad \ldots\ \wedge\ [\mathit{rest}]$$
$$(\forall(\texttt{lcons}\ \_\ \ell_2) \in \widehat{\rho}(\ell_1) :$$
$$(\texttt{delay}\ \ell_2) \in \widehat{\rho}(\ell)\ \wedge$$
$$\widehat{\rho}(\ell_2) \subseteq \widehat{\rho}(\ell))$$

Fig. 3. Step 2: Analysis with lazy forms.

$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (e_f^{\ell_f}\ e_1^{\ell_1} \ldots)^{\ell}$ iff [*app*]
$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_f^{\ell_f}\ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1} \wedge \ldots \wedge$
$(\forall \lambda(x_1 \ldots).\ell_0 \in \widehat{\rho}(\ell_f) :$
$\widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x_1)\ \wedge\ \ldots\ \wedge$
$\boxed{(\texttt{arg}\ \ell_1) \in \widehat{\rho}(x_1)\ \wedge\ \ldots}_1\ \wedge$
$\boxed{(\forall \widehat{v} \in \widehat{\rho}(\ell_0), \widehat{v} \notin \texttt{arg} : \widehat{v} \in \widehat{\rho}(\ell))}_2)$

$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{delay}\ e_1^{\ell_1})^{\ell}$ iff [*delay*]
$(\texttt{delay}\ \ell_1) \in \widehat{\rho}(\ell)\ \wedge$
$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1}\ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(\ell)\ \wedge$
$(\forall x \in fv(e_1) : (\forall(\texttt{arg}\ \ell_2) \in \widehat{\rho}(x) :$
$\boxed{\ell_2 \in \widehat{\mathcal{D}}}_3 \wedge \boxed{(\texttt{darg}\ \ell_2) \in \widehat{\rho}(x)}_4))$

$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{let}\ x = e_1^{\ell_1}\ \texttt{in}\ e_0^{\ell_0})^{\ell}$ iff [*let*]
$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1}\ \wedge\ \widehat{\rho}(\ell_1) \subseteq \widehat{\rho}(x)\ \wedge$
$\boxed{(\texttt{arg}\ \ell_1) \in \widehat{\rho}(x)}_1\ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_0^{\ell_0}\ \wedge$
$\boxed{(\forall \widehat{v} \in \widehat{\rho}(\ell_0), \widehat{v} \notin \texttt{arg} : \widehat{v} \in \widehat{\rho}(\ell))}_2$

$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e (\texttt{lcons}\ e_1^{\ell_1}\ e_2^{\ell_2})^{\ell}$ iff [*lcons*]
$(\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_1^{\ell_1}\ \wedge\ (\widehat{\rho},\widehat{\mathcal{D}}) \models_e e_2^{\ell_2}\ \wedge$
$(\texttt{lcons}\ \ell_1\ \ell_2) \in \widehat{\rho}(\ell)\ \wedge$
$(\forall x \in fv(e_2) : (\forall(\texttt{arg}\ \ell_3) \in \widehat{\rho}(x) :$
$\boxed{\ell_3 \in \widehat{\mathcal{D}}}_3 \wedge \boxed{(\texttt{darg}\ \ell_3) \in \widehat{\rho}(x)}_4))$

Fig. 4. Step 3a: Calculating flow to lazy positions.

3.4 Analysis Step 3: Laziness Analysis

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

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

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

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

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

Figure 4 presents revised analysis rules related to $\widehat{\mathcal{D}}$. To reduce clutter, we express the analysis result as $(\widehat{\rho},\widehat{\mathcal{D}})$, temporarily omitting $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{F}}$. In the new [*app*] and [*let*] rules, additional constraints (box 1) specify that for each labeled argument, an `arg` abstract value with a matching label must be in $\widehat{\rho}$ for the corresponding parameter. We are only interested in the flow of arguments within a function's body, so the result-propagating constraint filters out `arg` values (box 2).

Recall that $\widehat{\mathcal{D}}$ is to contain labels of arguments that reach lazy positions. Specifically, if an $(\texttt{arg}\ \ell)$ value flows to a `delay` or the second position of an

$$(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e (\texttt{force}\ e_1^{\ell_1})^{\ell} \text{ iff } \quad [\textit{force}]$$
$$(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e e_1^{\ell_1} \ \wedge$$
$$(\forall \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay} : \widehat{v} \in \widehat{\rho}(\ell)) \ \wedge$$
$$\boxed{(\forall(\texttt{arg}\ \ell_2) \in \widehat{\rho}(\ell_1) : \ell_2 \in \widehat{\mathcal{S}})}_5$$

$$(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}}) \models_e S[e^{\ell}] \text{ iff } \ \ldots \ \wedge \quad [\textit{strict}]$$
$$\boxed{(\forall(\texttt{arg}\ \ell_1) \in \widehat{\rho}(\ell) : \ell_1 \in \widehat{\mathcal{S}})}_5 \ \wedge$$
$$\boxed{(\exists \texttt{delay} \in \widehat{\rho}(\ell) \Rightarrow \ell \in \widehat{\mathcal{F}})}_6 \ \wedge$$
$$\boxed{(\forall(\texttt{darg}\ \ell_2) \in \widehat{\rho}(\ell) : (\ell,\ell_2) \in \widehat{\mathcal{F}})}_7$$

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

Fig. 5. Step 3b: Calculating flow to strict positions.

`lcons`, then ℓ must be in $\widehat{\mathcal{D}}$ (box 3) ($fv(e)$ calculates free variables in e). If an ℓ-labeled argument reaches a lazy position, the transformation *may* decide to delay that argument, so the analysis must additionally track it for the purposes of inserting `force`s. To this end, we introduce another abstract value ($\texttt{darg}\ \ell$),

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

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

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

3.5 The Refactoring Transformation

Figure 6 specifies our refactoring as a function φ that transforms a program p using analysis result $(\widehat{\rho},\widehat{\mathcal{D}},\widehat{\mathcal{S}},\widehat{\mathcal{F}})$. The φ_e function wraps expression e^{ℓ} with

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

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

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

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

$$\varphi_e[\![e^\ell]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = (\texttt{force}\ (\varphi_e[\![e]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}})^\ell)^{\ell_1}, \quad \text{if } \ell \in \widehat{\mathcal{F}},\ \ell_1 \notin \mathit{dom}(\widehat{\rho}), \qquad (\ddagger)$$

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

$$\ldots$$

Fig. 6. Transformation function φ.

$\texttt{delay}^*$ if ℓ is in $\widehat{\mathcal{D}}$ and not in $\widehat{\mathcal{S}}$. In other words, e is delayed if it flows to a lazy position but not a strict position. With the following correctness section in mind, we extend the set of expressions with $\texttt{delay}^*$, which is exactly like $\texttt{delay}$ and merely distinguishes programmer-inserted $\texttt{delay}$s from those inserted by the our transformation. The new $\texttt{delay}^*$ expression is given a fresh label ℓ_1. In two cases, φ_e inserts a $\texttt{force}$ around an expression e^ℓ. First, if ℓ is in $\widehat{\mathcal{F}}$, it means ℓ is a strict position and a programmer-inserted $\texttt{delay}$ reaches this strict position and must be forced. Second, an expression e^ℓ is also wrapped with $\texttt{force}$ if there is some ℓ_2 such that (ℓ, ℓ_2) is in $\widehat{\mathcal{F}}$ and the analysis says to delay the expression at ℓ_2, i.e., $\ell_2 \in \widehat{\mathcal{D}}$ and $\ell_2 \notin \widehat{\mathcal{S}}$. This ensures that transformation-inserted $\texttt{delay}^*$s are also properly forced. All remaining clauses in the definition of φ_e, represented with ellipses, traverse the structure of e in a homomorphic manner.

4 Correctness

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

4.1 Language Semantics

To establish soundness, we use Flanagan and Felleisen's [9] technique, which relies on a reduction semantics. The semantics is based on evaluation contexts, which are expressions with a hole in place of one subexpression:

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

A reduction step $\longmapsto$ is defined as follows, where $\to$ is specified in figure 7:

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

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

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

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

Exceptions

Our $\rightarrow$ reduction thus far is partial, as is the (elided) δ function. If certain expressions show up in the hole of the evaluation context, e.g., `first null` or division by 0, we consider the evaluation stuck. To handle stuck expressions, we add an exception `exn` to our semantics. We assume that δ returns `exn` for invalid operands of primitives and we extend $\rightarrow$ with the exception-producing reductions in figure 8.

The (apx) rule says that application of non-λs results in an exception. The (fstx) and (rstx) rules state that reducing `first` or `rest` with anything but a non-empty list is an exception as well. The (strictx) and (strictx*) reductions partially override some reductions from figure 7 and specify that an exception occurs when an unforced promise appears in a context where the value of that promise is required. These contexts are exactly the strict contexts S from figure 5. We introduce `dexn` and `dexn`* to indicate when a `delay` or `delay`* causes an exception; otherwise these tokens behave just like `exn`. We also extend $\longmapsto$:

$$E[\texttt{exn}] \longmapsto \texttt{exn}$$

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

$$\begin{array}{rlrl}
(\lambda(x\ldots).e)\ v\ldots \rightarrow e\{x := v, \ldots\} & \text{(ap)} & \texttt{null?}\ \texttt{null} \rightarrow \texttt{true} & \text{(nuln)} \\
o\ v_1\ v_2 \rightarrow \delta\ o\ v_1\ v_2 & \text{(op)} & \texttt{null?}\ v \rightarrow \texttt{false},\ v \neq \texttt{null} & \text{(nul)} \\
\texttt{let}\ x = v\ \texttt{in}\ e \rightarrow e\{x := v\} & \text{(let)} & \texttt{first}\ (\texttt{cons}\ v_1\ v_2) \rightarrow v_1 & \text{(fstc)} \\
\texttt{if false}\ e_1\ e_2 \rightarrow e_2 & \text{(iff)} & \texttt{first}\ (\texttt{lcons}\ v\ e) \rightarrow v & \text{(fstlc)} \\
\texttt{if}\ v\ e_1\ e_2 \rightarrow e_1,\ \ v \neq \texttt{false} & \text{(if)} & \texttt{rest}\ (\texttt{cons}\ v_1\ v_2) \rightarrow v_2 & \text{(rstc)} \\
\texttt{zero?}\ 0 \rightarrow \texttt{true} & \text{(z0)} & \texttt{rest}\ (\texttt{lcons}\ v\ e) \rightarrow \texttt{delay}\ e & \text{(rstlc)} \\
\texttt{zero?}\ v \rightarrow \texttt{false},\ \ v \neq 0 & \text{(z)} & \texttt{force}\ (\texttt{delay}\ e) \rightarrow \texttt{force}\ e & \text{(ford)} \\
\texttt{not false} \rightarrow \texttt{true} & \text{(notf)} & \texttt{force}\ v \rightarrow v,\ v \neq \texttt{delay}\ e & \text{(forv)} \\
\texttt{not}\ v \rightarrow \texttt{false},\ v \neq \texttt{false} & \text{(not)} & &
\end{array}$$

Fig. 7. Call-by-value reduction semantics.

$$v\ v_1 \ldots \to \texttt{exn},\ \text{if } v \neq \lambda(x \ldots).e \quad \text{(apx)} \qquad S[\texttt{delay}\ e] \to \texttt{dexn} \quad \text{(strictx)}$$
$$\texttt{first}\ v \to \texttt{exn},\ \text{if } v \notin \texttt{cons or lcons} \quad \text{(fstx)} \qquad S[\texttt{delay}^*\ e] \to \texttt{dexn}^* \quad \text{(strictx}^*\text{)}$$
$$\texttt{rest}\ v \to \texttt{exn},\ \text{if } v \notin \texttt{cons or lcons} \quad \text{(rstx)}$$

Fig. 8. Exception producing reductions.

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

4.2 Soundness of the Analysis

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

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

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

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

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

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

4.3 Safety of Refactoring

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

$$\xi[\![\widehat{\rho}]\!]_p = \widehat{\rho}',\ \text{where} \qquad \boxed{\xi : \mathit{Env} \times \mathit{Prog} \to \mathit{Env}}$$

$$\forall \ell, x \in \mathit{dom}(\widehat{\rho}) : \widehat{\rho}'(\ell) = \widehat{\rho}(\ell) \cup \{(\texttt{delay}^*\ \ell_1) \mid (\texttt{darg}\ \ell_1) \in \widehat{\rho}(\ell), (\texttt{delay}^*\ e_1^{\ell_1}) \in p\} \quad (1)$$
$$\widehat{\rho}'(x) = \widehat{\rho}(x) \cup \{(\texttt{delay}^*\ \ell_1) \mid (\texttt{darg}\ \ell_1) \in \widehat{\rho}(x), (\texttt{delay}^*\ e_1^{\ell_1}) \in p\}$$

$$\forall (\texttt{delay}^*\ e_1^{\ell_1})^\ell \in p, \ell \notin \mathit{dom}(\widehat{\rho}) : \quad (2)$$
$$\widehat{\rho}'(\ell) = \widehat{\rho}(\ell_1) \cup \{(\texttt{delay}^*\ \ell_1)\} \cup \{(\texttt{delay}^*\ \ell_2) \mid (\texttt{darg}\ \ell_2) \in \widehat{\rho}(\ell_1), (\texttt{delay}^*\ e_2^{\ell_2}) \in p\}$$

$$\forall (\texttt{force}\ e_1^{\ell_1})^\ell \in p, \ell \notin \mathit{dom}(\widehat{\rho}) : \ \widehat{\rho}'(\ell) = \{\widehat{v} \mid \widehat{v} \in \widehat{\rho}(\ell_1), \widehat{v} \notin \texttt{delay}\} \quad (3)$$

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

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

Finally, theorem 3 states the safety property. It says that evaluating a transformed program cannot generate an exception due to `delay`s or `delay`*s.

Theorem 3 (Safety). *For all* p *and* $(\widehat{\rho}, \widehat{\mathcal{D}}, \widehat{\mathcal{S}}, \widehat{\mathcal{F}}) \models p$, *if* $\varphi[\![p]\!]_{\widehat{\rho}\widehat{\mathcal{D}}\widehat{\mathcal{S}}\widehat{\mathcal{F}}} = d \ldots e$, *then* $e \not\mapsto\!\!\!\twoheadrightarrow_{d\ldots}$ ***dexn***, *and* $e \not\mapsto\!\!\!\twoheadrightarrow_{d\ldots}$ ***dexn****.

4.4 Idempotency

Our transformation is not idempotent. Indeed, it may be necessary to refactor a program multiple times to get the "right" amount of laziness. For example:

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

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

5 A Prototype Implementation

We have implemented refactoring for laziness as a tool for Racket [10], in the form of a plugin for the DrRacket IDE. It uses laziness analysis to automatically insert `delay` and `force` expressions as needed, with graphical justification.

5.1 Constraint Solving Algorithm

Computing our laziness analysis requires two stages: (1) generate a set of constraints from a program, and (2) solves for the least solution using a conventional worklist algorithm [18]. The graph nodes are the variables and labels in the program, plus one node each for $\widehat{\mathcal{D}}$, $\widehat{\mathcal{S}}$, and $\widehat{\mathcal{F}}$. Without loss of generality, we use only labels for the nodes and $\widehat{\rho}$ for the analysis result in our description of the algorithm. There exists an edge from node ℓ_1 to ℓ_2 if there is a constraint where $\widehat{\rho}(\ell_2)$ depends on $\widehat{\rho}(\ell_1)$; the edge is labeled with that constraint. Thus one can view a node ℓ as the endpoint for a series of data flow paths. To compute $\widehat{\rho}(\ell)$, it suffices to traverse all paths from the leaves to ℓ, accumulating values according to the constraints along the way.

The analysis result is incrementally computed in a breadth-first fashion by processing constraints according a worklist of nodes. Processing a constraint entails adding values to $\widehat{\rho}$ so the constraint is satisfied. The algorithm starts by processing all constraints where a node depends on a value, e.g., `val` $\in \widehat{\rho}(\ell)$;

the nodes on the right-hand side of these constraints constitute the initial worklist. Nodes are then removed from the worklist, one at a time. When a node is removed, the constraints on the out-edges of that node are processed and a neighbor ℓ of the node is added to the worklist if $\widehat{\rho}(\ell)$ was updated while processing a constraint. A node may appear in the worklist more than once, but only a finite number of times, as shown by the following termination argument.

Termination and Complexity of Constraint Solving

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

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

5.2 Laziness Refactoring Tool

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

We evaluated a number of examples with our tool including the n-queens problem from section 2. Figure 9 (top) shows the program in Racket, including timing information and a graphical depiction of the answer. Despite the use of `lcons`,[8] the program takes as long as an eager version of the same program (not shown) to compute an answer. Figure 9 (bot) shows the program after our tool applies the laziness transformation. When the tool is activated, it: (1) computes an analysis result for the program, (2) uses the result to insert `delay`s and `force`s, highlighting the added `delay`s in yellow and the added `force`s in blue, and (3) adds arryows originating from each inserted `delay`, pointing to the source of the laziness, thus explaining its decision to the programmer in an intuitive manner. Running the transformed program exhibits the desired performance.

[8] Though `lcons` is not available in Racket, to match the syntax of our paper, we simulate it with a macro that wraps a `delay` around the second argument of a `cons`.

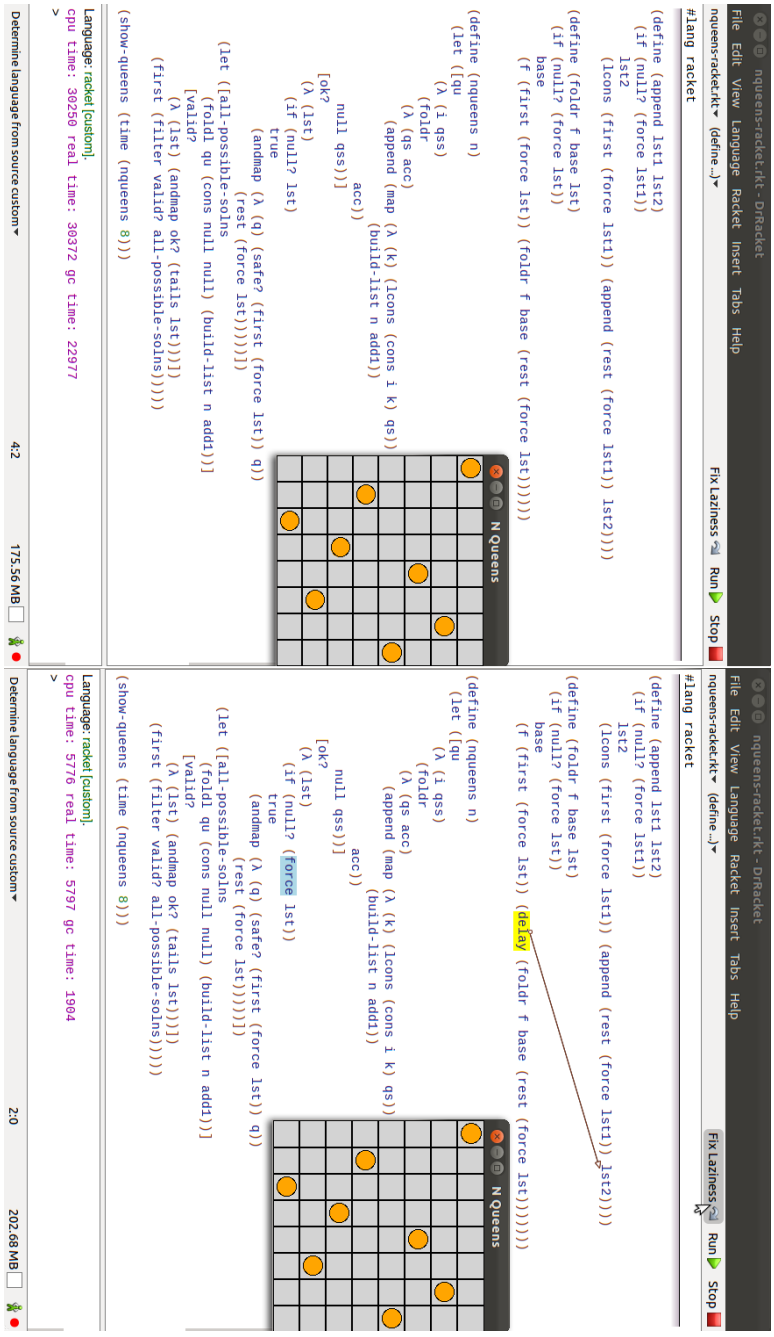

Fig. 9. Evaluating n-queens in Racket: only lazy `cons` (top), after refactoring (bot).

6 Laziness in the Large

To further evaluate our idea and our tool, we examined the Racket code base and some user-contributed packages for manual uses of laziness. We found several erroneous attempts at adding laziness and we verified that our tool would have prevented many such errors.[9] 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 [5] allow students to write contracts for some data structures. These contracts are based on Findler et al.'s lazy contracts [8]. The contracts are primarily implemented via a constructor with a few lazy fields. Additionally, several specialized contract constructors for various data structures call the main constructor. However, since the specialized constructors are implemented with ordinary strict functions, to preserve the intended lazy behavior, the programmer must manually propagate the laziness to the appropriate arguments of these functions, similar to the Scala example from section 2. Thus, a small amount of laziness in the main contract constructor requires several more `delay`s scattered all throughout the program. Adding these `delay`s becomes tedious as the program grows in complexity and unsurprisingly, a few were left out. 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 [21] based on implicit recursive slowdown [20, Chapter 11], where laziness enables fast amortized operations and simplifies the implementation. The library contained several performance bugs, as illustrated by this code snippet from a deque enqueue function:

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

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

7 Related Work

The idea of combining strict and lazy evaluation is old, but most works involve removing laziness from lazy languages. We approach strict-lazy programming

[9] The examples were first translated to work with the syntax in this paper.

from the other, relatively unexplored, end of the spectrum, starting with a strict language and then only adding laziness as needed. This seems worthwhile since empirical studies indicate that most promises in a lazy language are unneeded [6, 15, 16, 23]. Starting with a strict language also alleviates many disadvantages of lazy evaluation such as difficulty reasoning about space/time consumption.

The most well-known related work is strictness analysis [4, 17], which calculates when to eagerly evaluate arguments without introducing non-termination. With our work, calculating divergence properties is not sufficient since even terminating programs may require additional laziness, as seen in examples from this paper. Hence we take a different, flow-analysis-based approach.[10] Researchers have also explored other static [7] and dynamic [2, 6, 15] laziness-removal techniques. However, these efforts all strive to preserve the program's semantics. We focus on the problem of strict programmers trying to use laziness, but doing so *incorrectly*. Thus our transformation necessarily allows the semantics of the program to change (i.e., from non-terminating to terminating), but hopefully in a way that the programmer intended in the first place.

Sheard [25] shares our vision of a strict language that is also practical for programming lazily. While his language does not require explicit `force`s, the programmer must manually insert all required `delay` annotations.

8 Future Work

This paper demonstrates the theoretical and practical feasibility of a novel approach to assist programmers with the introduction of laziness into a strict context. We see several directions for future work. The first is developing a modular analysis. Our transformation requires the whole program and is thus unsatisfactory in the presence of libraries. Also, we intend to develop a typed version of our transformation and tool, so typed strict languages can more easily benefit from laziness as well. We conjecture that expressing strictness information via types may also provide a way to enable a modular laziness-by-need analysis.

Acknowledgements. Partial support provided by NSF grant CRI-0855140. Thanks to Matthias Felleisen, Eli Barzilay, David Van Horn, and J. Ian Johnson for feedback on earlier drafts.

References

1. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. MIT Press (1984)
2. Aditya, S., Arvind, Augustsson, L., Maessen, J.W., Nikhil, R.S.: Semantics of pH: A parellel dialect of Haskell. In: Proc. Haskell Workshop. pp. 34–49 (1995)

[10] Interestingly, we conjecture that our approach would be useful to lazy programmers trying to insert strictness *annotations*, such as Haskell's `seq`, to their programs.

3. Appel, A., Blume, M., Gansner, E., George, L., Huelsbergen, L., MacQueen, D., Reppy, J., Shao, Z.: Standard ML of New Jersey User's Guide (1997)
4. Burn, G.L., Hankin, C.L., Abramsky, S.: Strictness analysis for higher-order functions. Sci. Comput. Program. 7(0), 249–78 (1986)
5. Crestani, M., Sperber, M.: Experience report: growing programming languages for beginning students. In: Proc. 15th ICFP. pp. 229–234 (2010)
6. Ennals, R., Peyton Jones, S.: Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In: Proc. 8th ICFP. pp. 287–298 (2003)
7. Faxén, K.F.: Cheap eagerness: speculative evaluation in a lazy functional language. In: Proc. 5th ICFP. pp. 150–161 (2000)
8. Findler, R.B., Guo, S.y., Rogers, A.: Lazy contract checking for immutable data structures. In: Proc. IFL. pp. 111–128 (2007)
9. Flanagan, C., Felleisen, M.: Modular and polymorphic set-based analysis: Theory and practice. Tech. Rep. TR96-266, Rice Univ. (1996)
10. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2012-1, PLT Inc. (2012), `http://racket-lang.org/tr1/`
11. Friedman, D., Wise, D.: Cons should not evaluate its arguments. In: Proc. 3rd ICALP. pp. 257–281 (1976)
12. Hughes, J.: Why functional programming matters. Comput. J. 32, 98–107 (1989)
13. Jones, N.D.: Flow analysis of lambda expressions. Tech. rep., Aarhus Univ. (1981)
14. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system, release 3.12, Documentation and users manual. INRIA (July 2011)
15. Maessen, J.W.: Eager Haskell: resource-bounded execution yields efficient iteration. In: Proc. Haskell Workshop. pp. 38–50 (2002)
16. Morandat, F., Hill, B., Osvald, L., Vitek, J.: Evaluating the design of the R language. In: Proc. 26th ECOOP. pp. 104–131 (2012)
17. Mycroft, A.: Abstract interpretation and optimising transformations for applicative programs. Ph.D. thesis, Univ. Edinburgh (1981)
18. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer (2005)
19. Odersky, M.: The Scala Language Specification, Version 2.9. EPFL (May 2011)
20. Okasaki, C.: Purely Functional Data Structures. Cambridge Univ. Press (1998)
21. Prashanth K R, H., Tobin-Hochstadt, S.: Functional data structures for Typed Racket. In: Proc. Scheme Workshop (2010)
22. Rees, J., Clinger, W.: Revised^3 Report on the Algorithmic Language Scheme. ACM SIGPLAN Notices 21(12), 37–79 (December 1986)
23. Schauser, K.E., Goldstein, S.C.: How much non-strictness do lenient programs require? In: Proc. 7th FPCA. pp. 216–225 (1995)
24. Sestoft, P.: Replacing function parameters by global variables. Master's thesis, Univ. Copenhagen (1988)
25. Sheard, T.: A pure language with default strict evaluation order and explicit laziness. In: 2003 Haskell Workshop: New Ideas session (2003)
26. Shivers, O.: Control-flow analysis in scheme. In: Proc. PLDI. pp. 164–174 (1988)
27. Wadler, P., Taha, W., MacQueen, D.: How to add laziness to a strict language, without even being odd. In: Proc. Standard ML Workshop (1998)