From Stack Traces to Lazy Rewriting Sequences

Stephen Chang[1], Eli Barzilay[1], John Clements[2], and Matthias Felleisen[1]

[1] Northeastern University, Boston, Massachusetts, USA
[2] California Polytechnic State University, San Luis Obispo, California, USA

Abstract. Reasoning about misbehaving lazy functional programs can be confusing, particularly for novice programmers. Unfortunately, the complicated nature of laziness also renders most debugging tools ineffective at clarifying this confusion. In this paper, we introduce a new lazy debugging tool for novice programmers, an algebraic stepper that presents computation as a sequence of parallel rewriting steps. Parallel program rewriting represents sharing accurately and enables debugging at the level of source syntax, minimizing the presentation of low-level details or the effects of distorting transformations that are typical for other lazy debuggers. Semantically, our rewriting system represents a compromise between Launchbury's store-based semantics and an axiomatic description of lazy computation as sharing-via-parameters. Finally, we prove the correctness of our tool by showing that the stepper's run-time machinery reconstructs the expected lazy rewriting sequence.

Keywords: lazy programming, debugging, lazy lambda calculus

1 How Functional Programming Works

While laziness enables modularization [12], it unfortunately also reduces a programmer's ability to predict the ordering of evaluations. As long as programs work, this cognitive dissonance poses no problems. When a lazy program exhibits erroneous behavior, however, reasoning about the code becomes confusing, especially for novices. A programmer can turn to a debugger for help, but the nature of laziness affects these tools as well, often forcing them to present evaluation in a distorted manner, with some debuggers ignoring or hiding laziness altogether.

In this paper, we present a new debugging tool for students of lazy programming, an algebraic stepper for Lazy Racket that explains computation via a novel rewriting semantics. Our key idea is to use substitution in conjunction with selective parallel reduction to simulate shared reductions. Shared expressions are identified semantically with labels and are reduced simultaneously in the program source. This enables a clean syntactic presentation of lazy evaluation that eliminates many of the drawbacks of previous syntax-based tools. Showing computation as a rewriting of the program source means that we do not need to apply complicated preprocessing transformations nor do we need extraneous low-level details to explain evaluation. In addition, our experience with the DrRacket stepper [6] for call-by-value Racket, as well as studies of other researchers [11, 20], confirm that students find syntax-based tools more intuitive to use than graphical ones. This makes sense because programmers are already used to reasoning about their progams in terms of the source code.

Our rewriting semantics is also the appropriate basis for a correctness proof of the stepper. For the proof, we use a Haskell-like, thunk-based lazy language model, further enriched with continuation marks [6]—which help reconstruct the stepper sequence—and then exploit a proof strategy from Clements [5] to show that the implementation language bisimulates our lazy rewriting semantics.

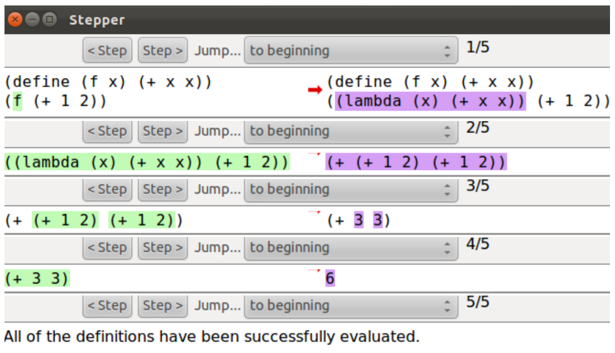
Section 2 briefly introduces Lazy Racket and our stepper with examples. Section 3 presents our novel lazy rewriting system. Section 4 summarizes the implementation of Lazy Racket and presents a model of the lazy stepper, and section 5 presents a correctness proof for the stepper.

2 Lazy Racket and its Stepper

Lazy Racket programs are a series of definitions and expressions that refer to those definitions. Here is a simplistic example:

```
(define (f x) (+ x x))
(f (+ 1 2))
```

A programmer invokes the Lazy Racket stepper from the DrRacket IDE. Running the stepper displays the rewriting sequence for the current program. Figure 1 shows a sequence of screenshots for the rewriting sequence of the above program. A green box highlights redexes on the left-hand side of a step and a purple box highlights contractums on the right-hand side. The programmer can navigate the rewriting sequence in either the forward or backward direction.



```
1/5
(define (f x) (+ x x))          (define (f x) (+ x x))
(f (+ 1 2))                     ((lambda (x) (+ x x)) (+ 1 2))
2/5
((lambda (x) (+ x x)) (+ 1 2))  (+ (+ 1 2) (+ 1 2))
3/5
(+ (+ 1 2) (+ 1 2))             (+ 3 3)
4/5
(+ 3 3)                         6
5/5
All of the definitions have been successfully evaluated.
```

Fig. 1. Lazy Stepper, Example 1

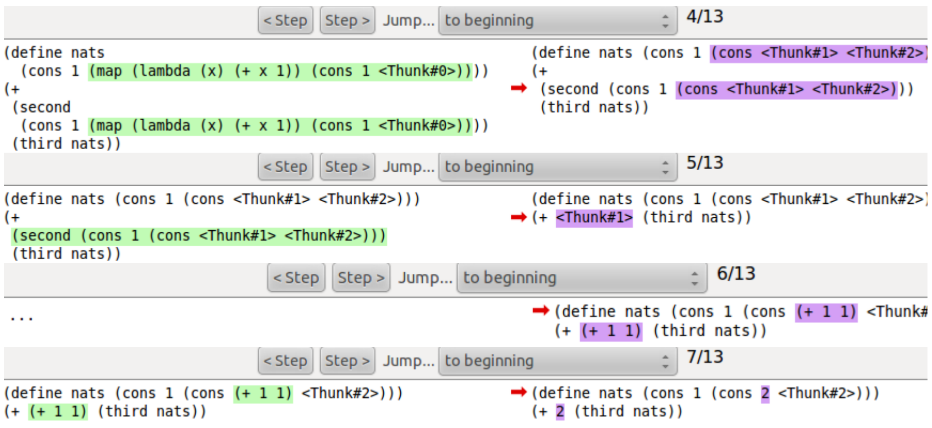
In step 2, evaluation of the argument is delayed so an unevaluated argument replaces each instance of the variable `x` in the function body. In step 3, evaluation of the program at the first `x` requires the value of the argument, so it is forced and all shared instances are reduced simultaneously. That is, *the stepper explains evaluation as an algebraic process using a form of parallel program rewriting.*

Since the second x refers to the same argument as the first x, by the time evaluation of the program requires a value for the second x, a result is already available because the computed value of the first x was saved. In short, no argument evaluation is repeated, satisfying the criteria for lazy evaluation.

A second example involves infinite lists:

```
(define (add-one x) (+ x 1))
(define nats (cons 1 (map add-one nats)))
(+ (second nats) (third nats))
```

The rewriting sequence for this program appears in figure 2. Only the essential steps are shown. In step 4, `<Thunk#0>` is an opaque reference to the rest of `nats`, which is currently being evaluated. The evaluation of `second` forces the `map` expression to produce a `cons` of two more thunks. The thunks are displayed as `<Thunk#1>` and `<Thunk#2>` because `map` is a libary function whose source is unknown. In step 5, `second` extracts `<Thunk#1>` from the list. In step 6, evaluation requires the value of `<Thunk#1>`, so it is forced. The ellipses on the left indicate forcing of an opaque thunk. In steps 6 and 7, the stepper simultaneously updates the `nats` definition with the result. It highlights only shared terms that are part of the current redex. This cleans up the presentation of the rewriting steps and makes lazy evaluation easier to follow in our tool. The remaining steps show the similar evaluation of the other addition operand and are thus omitted.



```
(define nats                                        (define nats (cons 1 (cons <Thunk#1> <Thunk#2>)
  (cons 1 (map (lambda (x) (+ x 1)) (cons 1 <Thunk#0>))))   (+
(+                                                ➡  (second (cons 1 (cons <Thunk#1> <Thunk#2>)))
 (second                                             (third nats))
  (cons 1 (map (lambda (x) (+ x 1)) (cons 1 <Thunk#0>))))
 (third nats))
```

```
(define nats (cons 1 (cons <Thunk#1> <Thunk#2>)))   (define nats (cons 1 (cons <Thunk#1> <Thunk#2>)
(+                                                ➡ (+ <Thunk#1> (third nats))
 (second (cons 1 (cons <Thunk#1> <Thunk#2>)))
 (third nats))
```

```
...                                               ➡ (define nats (cons 1 (cons (+ 1 1) <Thunk#
                                                    (+ (+ 1 1) (third nats))
```

```
(define nats (cons 1 (cons (+ 1 1) <Thunk#2>)))   ➡ (define nats (cons 1 (cons 2 <Thunk#2>)))
(+ (+ 1 1) (third nats))                            (+ 2 (third nats))
```

Fig. 2. Lazy Stepper, Example 2

3 Lazy Racket Semantics

Our key theoretical innovation is the lazy rewriting system, λ_{LR}, which specifies the exact nature of steps for our tool. The syntax of λ_{LR} is identical to the core of most functional programming languages and includes integers, strings, booleans, variables, abstractions, applications, primitives, lists, and a conditional:[3]

$$e = n \mid s \mid b \mid x \mid \lambda x.e \mid (e\ e) \mid (\texttt{cons}\ e\ e) \mid \texttt{null} \mid (p^1\ e) \mid (p^2\ e\ e) \mid (\texttt{if}\ e\ e\ e)$$
$$n \in \mathbb{Z}, \quad s \in \text{Strings}, \quad b = \texttt{true} \mid \texttt{false}, \quad p^1 = \texttt{first} \mid \texttt{rest}, \quad p^2 = + \mid - \mid * \mid /$$

To specify the semantics of λ_{LR}, we first extend e by adding a new expression:

$$e^{\text{LR}} = e \mid e^{\text{LR}^{\ell}} \qquad \ell \in \text{Labels}$$

The "labeled" expression, $e^{\text{LR}^{\ell}}$, consists of a tag ℓ and a subexpression e^{LR}. Labeled expressions are not part of the language syntax but are necessary for evaluation. Rewriting a labeled expression triggers the simultaneous rewriting of all other expressions with the same label. In our language, we require all expressions with the same label ℓ to be identical. We call this consistent labeling:

Definition 1. *A program is consistently labeled if, for all ℓ_1, ℓ_2, e_1^{LR}, e_2^{LR}, if $e_1^{\text{LR}^{\ell_1}}$ and $e_2^{\text{LR}^{\ell_2}}$ are two subexpressions in a program, and $\ell_1 = \ell_2$, then $e_1^{\text{LR}} = e_2^{\text{LR}}$.*

3.1 Rewriting Rules

To further formulate a semantics, we define the notion of values:

$$v = n \mid s \mid b \mid \lambda x.e^{\text{LR}} \mid \texttt{null} \mid (\texttt{cons}\ e^{\text{LR}^{\ell}}\ e^{\text{LR}^{\ell}}) \mid v^{\ell}$$

Numbers, strings, booleans, λs, `null`, and `cons` expressions where each element is labeled, are values. In addition, any value tagged with labels is also a value.

In the rewriting of λ_{LR} programs, evaluation contexts determine which part of the program to rewrite next. Evaluation contexts are expressions where a hole [ ] replaces one subexpression:

$$E = [\,] \mid (E\ e^{\text{LR}}) \mid (p^2\ E\ e^{\text{LR}}) \mid (p^2\ v\ E) \mid (p^1\ E) \mid (\texttt{if}\ E\ e^{\text{LR}}\ e^{\text{LR}}) \mid E^{\ell}$$

The $(E\ e^{\text{LR}})$ context indicates that the operator in an application must be evaluated so that application may proceed. The p^1 and p^2 contexts indicate that primitives p^i are strict in all argument positions and are evaluated left to right. The `if` context dictates strict evaluation of only the test expression. Finally, the E^{ℓ} context dictates that a redex search goes under labeled expressions.

Evaluation of a λ_{LR} program proceeds according to the program rewriting system in figure 3. It has two phases. A rewriting step begins when the progam is partitioned into a redex and an evaluation context and the redex is contracted according to the phase 1 rules. If the redex does not occur under a label, it is the only contracted part of the program. If the redex does occur under a label, then in phase 2, all other instances of that labeled expression are contracted in the same way. In phase 2, the evaluation context is further subdivided as $E[] = E_1[(E_2[\,])^{\ell}]$ where ℓ is the label nearest the redex, E_1 is the context around the ℓ-labeled expression, and E_2 is the context under label ℓ. Thus E_2 contains no additional labels on the path from the root to the hole. An "update" function performs

[3] Cyclic structures are omitted for space but should be straightforward to add, see [9].

$\boxed{\text{Phase 1:}}$ $$E[e^{\text{LR}}] \quad \overset{phase1}{\longmapsto}_{\text{LR}} \quad E[e^{\text{LR}\prime}]$$

where: e^{LR}	$e^{\text{LR}\prime}$	
$((\lambda x.e_1^{\text{LR}})^{\vec{\ell}}\ e_2^{\text{LR}})$	$e_1^{\text{LR}}\{x := e_2^{\text{LR}\,\ell_1}\}$, $\texttt{fresh}\ \ell_1$	β_{LR}
$(p^2\ n_1^{\vec{\ell}}\ n_2^{\vec{\ell}})$	$(\delta\ (p^2\ n_1\ n_2))$	Prim
$(\texttt{cons}\ e_1^{\text{LR}}\ e_2^{\text{LR}})$, e_1^{LR} or e_2^{LR} unlabeled	$(\texttt{cons}\ e_1^{\text{LR}\,\ell_1}\ e_2^{\text{LR}\,\ell_2})$, $\texttt{fresh}\ \ell_1, \ell_2$	Cons
$((\texttt{first} \mid \texttt{rest})\ (\texttt{cons}\ e_1^{\text{LR}}\ e_2^{\text{LR}})^{\vec{\ell}})$	$e_1^{\text{LR}} \mid e_2^{\text{LR}}$	First $\mid$ Rest
$(\texttt{if}\ (\texttt{true} \mid \texttt{false})^{\vec{\ell}}\ e_1^{\text{LR}}\ e_2^{\text{LR}})$	$e_1^{\text{LR}} \mid e_2^{\text{LR}}$	If-T $\mid$ If-F

$\boxed{\text{Phase 2:}}$ If redex occurs under (nearest) label ℓ, where $E[\,] = E_1[(E_2[\,])^{\ell}]$, then:

$$E[e^{\text{LR}\prime}] \quad \overset{phase2}{\longmapsto}_{\text{LR}} \quad E[e^{\text{LR}\prime}]\{\!\{\ell \Leftarrow E_2[e^{\text{LR}\prime}]\}\!\}$$

Fig. 3. The λ_{LR} Rewriting System.

the parallel reduction, where the notation $e_1^{\text{LR}}\{\!\{\ell \Leftarrow e_2^{\text{LR}}\}\!\}$ means "in e_1^{LR}, replace expressions under label ℓ with e_2^{LR}." The function is formally defined as follows:

$$\begin{aligned}
e_1^{\text{LR}\,\ell}\{\!\{\ell \Leftarrow e_2^{\text{LR}}\}\!\} &= e_2^{\text{LR}\,\ell} \\
e_1^{\text{LR}\,\ell_1}\{\!\{\ell_2 \Leftarrow e_2^{\text{LR}}\}\!\} &= (e_1^{\text{LR}}\{\!\{\ell_2 \Leftarrow e_2^{\text{LR}}\}\!\})^{\ell_1},\ \ell_1 \neq \ell_2 \\
(\lambda x.e_1^{\text{LR}})\{\!\{\ell \Leftarrow e_2^{\text{LR}}\}\!\} &= \lambda x.(e_1^{\text{LR}}\{\!\{\ell \Leftarrow e_2^{\text{LR}}\}\!\}) \\
(e_1^{\text{LR}}\ e_2^{\text{LR}})\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\} &= (e_1^{\text{LR}}\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\}\ e_2^{\text{LR}}\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\}) \\
(\_\ \ e_1^{\text{LR}}\ e_2^{\text{LR}} \ldots)\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\} &= (\_\ \ e_1^{\text{LR}}\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\}\ e_2^{\text{LR}}\{\!\{\ell \Leftarrow e_3^{\text{LR}}\}\!\} \ldots)
\end{aligned}$$

In figure 3, the β_{LR} rule specifies that function application occurs before the evaluation of arguments. To remember where expressions originate, the argument receives an unused label ℓ_1 before substitution is performed. The notation $e^{\text{LR}\,\vec{\ell}}$ represents an expression wrapped in one or more labels. During a rewriting step, labels are discarded from values because no further reduction is possible. Binary p^2 primitive applications are strict in their arguments, as seen in the Prim rule. The δ function interprets these primitives and is defined in the standard way (division by 0 results in a stuck state). The Cons rule shows that, if either argument is unlabeled, both arguments are wrapped with new labels. Adding an extra label around an already labeled expression does not affect evaluation because parallel updating only uses the innermost label. The First and Rest rules extract the appropriate component from a `cons` cell, and the If-T and If-F rules choose the first or second branch of the `if` expression.

Program rewriting preserves the consistent labeling property.

Lemma 1. *If $e_1^{\text{LR}} \longmapsto_{\text{LR}} e_2^{\text{LR}}$ and e_1^{LR} is consistently labeled, then e_2^{LR} is as well.*

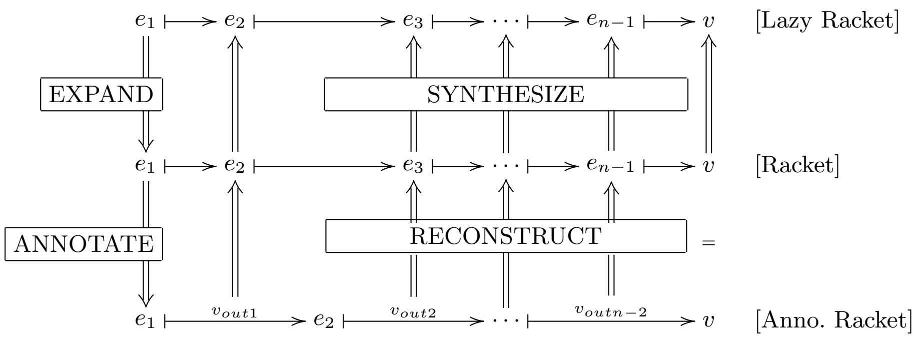
The rewriting rules are deterministic because any expression e^{LR} can be uniquely partitioned into an evaluation context and a redex. Thus if e_1^{LR} rewrites to a expression e_2^{LR}, then e_1^{LR} rewrites to e_2^{LR} in a canonical manner. We can then use $\longmapsto_{\text{LR}}$, the composition of $\xmapsto{phase1}_{\text{LR}}$ and $\xmapsto{phase2}_{\text{LR}}$, to define an evaluator:

$$\texttt{eval}_{\text{LR}}(e) \begin{cases} v, & \text{if } e \longmapsto\!\!\!\!\rightarrow_{\text{LR}} v \\ \bot, & \text{if, for all } e \longmapsto\!\!\!\!\rightarrow_{\text{LR}} e_1^{\text{LR}}, e_1^{\text{LR}} \longmapsto_{\text{LR}} e_2^{\text{LR}} \\ \texttt{error}, & \text{if } e \longmapsto\!\!\!\!\rightarrow_{\text{LR}} e_1^{\text{LR}}, e_1^{\text{LR}} \notin v, \nexists e_2^{\text{LR}} \text{ such that } e_1^{\text{LR}} \longmapsto_{\text{LR}} e_2^{\text{LR}} \end{cases}$$

where $\longmapsto\!\!\!\!\rightarrow_{\text{LR}}$ is the reflexive-transitive closure of $\longmapsto_{\text{LR}}$.

4 Lazy Stepper Implementation

Figure 4 summarizes the software architecture of our stepper. The first row depicts a λ_{LR} Lazy Racket rewriting sequence. To construct this rewriting sequence, the stepper first macro-expands a Lazy Racket program to a plain Racket program with `delay` and `force`. Then, annotations are added to the expanded program such that executing it emits a series of output values representing stack traces. The stepper reconstructs the reduction sequence for the unannotated Racket program from these stack traces. Finally, this plain Racket reduction sequence is synthesized to the desired Lazy Racket rewriting sequence.



EXPAND
SYNTHESIZE
ANNOTATE
RECONSTRUCT
[Lazy Racket]
[Racket]
[Anno. Racket]
e_1, e_2, e_3, e_{n-1}, v
v_{out1}, v_{out2}, v_{outn-2}
=

Fig. 4. Stepper Implementation Architecture

The correctness of the lazy stepper thus depends on two claims:

1. The reduction sequence of a plain Racket program can be reconstructed from the output produced by an annotated version of that program.
2. The rewriting sequence of a Lazy Racket program is equivalent to the reduction sequence of the corresponding Racket program, modulo macros.

The first point corresponds to the work of Clements [6] and is depicted by the bottom half of figure 4. The second point is depicted by the top half of figure 4. The rest of the section formally presents the architecture in enough detail so that (1) our stepper can be implemented for any lazy programming language, and (2) so that we can prove its correctness.

4.1 Racket + `delay`/`force`

When the stepper is invoked on a Lazy Racket program, the source is first macro-expanded to a Racket program with `delay` and `force` strategically inserted. We model this latter language with λ_{DF}:

$$\begin{aligned} e^{\text{RKT}} = {} & n \mid s \mid b \mid x \mid \lambda x.e^{\text{RKT}} \mid (e^{\text{RKT}}\ e^{\text{RKT}}) \mid (\texttt{if}\ e^{\text{RKT}}\ e^{\text{RKT}}\ e^{\text{RKT}}) \mid (\texttt{cons}\ e^{\text{RKT}}\ e^{\text{RKT}}) \mid \texttt{null} \\ & \mid (p^1\ e^{\text{RKT}}) \mid (p^2\ e^{\text{RKT}}\ e^{\text{RKT}}) \mid (\texttt{delay}\ e^{\text{RKT}}) \mid (\texttt{force}\ e^{\text{RKT}}) \end{aligned}$$

$$n \in \mathbb{Z}, \quad s \in \text{Strings}, \quad b = \texttt{true} \mid \texttt{false}, \quad p^1 = \texttt{first} \mid \texttt{rest}, \quad p^2 = + \mid - \mid * \mid /$$

The syntax of λ_{DF} is similar to λ_{LR} except that `delay` and `force` replace labeled expressions. A `delay` expression suspends evaluation of its argument in a thunk; applying `force` to a (nest of) thunk(s) evaluates it and memoizes the result.

The semantics of λ_{DF} combines the usual call-by-value world with store effects. We describe it with a CS machine [8]. The C stands for control string, and the S is a store. In our machine the control string may contain locations, i.e., references to delayed expressions in the store. In contrast to the standard CS machine, our store holds only delayed computations:

$$\begin{array}{lll|lll} e^{\text{DF}} & = e^{\text{RKT}} \mid \ell & \text{(Machine Expressions)} & c^{\text{DF}} & = E^{\text{DF}}[e^{\text{DF}}] & \text{(Control Strings)} \\ \mathcal{S}^{\text{DF}} & = \mathcal{P}_1^{\text{DF}}, \ldots, \mathcal{P}_n^{\text{DF}} & \text{(Transition Sequences)} & \mathcal{P}^{\text{DF}} & = \langle c^{\text{DF}}, \sigma \rangle & \text{(Machine States)} \\ \ell & \in \text{Locations} & & \sigma & = ((\ell, e^{\text{DF}}), \ldots) & \text{(Stores)} \end{array}$$

$$\begin{aligned} & & \text{(Evaluation Contexts)} \\ E^{\text{DF}} = {} & [\,] \mid (E^{\text{DF}}\ e^{\text{DF}}) \mid (v^{\text{DF}}\ e^{\text{DF}}) \mid (\texttt{if}\ E^{\text{DF}}\ e^{\text{DF}}\ e^{\text{DF}}) \mid (p^2\ E^{\text{DF}}\ e^{\text{DF}}) \mid (p^2\ v^{\text{DF}}\ E^{\text{DF}}) & \\ & \mid (\texttt{cons}\ E^{\text{DF}}\ e^{\text{DF}}) \mid (\texttt{cons}\ v^{\text{DF}}\ E^{\text{DF}}) \mid (p^1\ E^{\text{DF}}) \mid (\texttt{force}\ E^{\text{DF}}) \mid (\texttt{force}\ \ell\ E^{\text{DF}}) & \\ v^{\text{DF}} = {} & n \mid s \mid b \mid \lambda x.e^{\text{DF}} \mid (\texttt{cons}\ v^{\text{DF}}\ v^{\text{DF}}) \mid \texttt{null} \mid \ell & \text{(Values)} \end{aligned}$$

The store is represented as a list of pairs; ellipses means "zero or more of the preceding element". The evaluation contexts are the standard by-value contexts, plus two `force` contexts. The first resembles the `force` expression in a program and indicates the forcing of some arbitrary expression. The second `force` context is used when evaluating a delayed computation stored in location ℓ. Evaluation under a (`force` ℓ [ ]) context corresponds to evaluation under a label in λ_{LR}. This special second `force` context is non-syntactic, hence the need for defining separate machine expressions (e^{DF}) and control strings (c^{DF}) above.

The starting machine configuration for a Racket program e^{RKT} is $\langle e^{\text{RKT}}, (\,) \rangle$ where the program is set as the control string and the store is initially empty. Evaluation stops when the control string is a value. Values are numbers, strings, booleans, abstractions, lists, or store locations. The CS machine transitions are specified in figure 5. Every program e^{RKT} has a deterministic transition sequence because the left hand sides of all the transition rules are mutually exclusive and cover all possible control strings in the C register.

The by-value β_v transition is standard, as are the omitted transitions for `if` and the p^i primitive functions. The DELAY transition reduces a `delay` expression to an unused location ℓ and the suspended expression is saved at that location in the store. When the argument to a `force` expression is a location, the suspended expression at that location is retrieved from the store and plugged into a special

$$\begin{array}{rcll}
 & \longmapsto_{\text{CS}} & & \\
\hline
\langle E^{\text{DF}}[((\lambda x.e^{\text{DF}})\ v^{\text{DF}})], \sigma\rangle & \longmapsto_{\text{CS}} & \langle E^{\text{DF}}[e^{\text{DF}}\{x := v^{\text{DF}}\}], \sigma\rangle & \beta_v \\
 & \dots & & \\
\langle E^{\text{DF}}[(\texttt{delay}\ e^{\text{DF}})], \sigma\rangle & \longmapsto_{\text{CS}} & \langle E^{\text{DF}}[\ell], \sigma[\![\ell \leftarrow e^{\text{DF}}]\!]\rangle\ , \ell \notin \texttt{dom}(\sigma) & \textsc{Delay} \\
\langle E^{\text{DF}}[(\texttt{force}\ \ell)], \sigma\rangle & \longmapsto_{\text{CS}} & \langle E^{\text{DF}}[(\texttt{force}\ (\texttt{force}\ \ell\ \sigma[\![\ell]\!]))], \sigma\rangle & \textsc{Force-delay} \\
\langle E^{\text{DF}}[(\texttt{force}\ \ell\ v^{\text{DF}})], \sigma\rangle & \longmapsto_{\text{CS}} & \langle E[v^{\text{DF}}], \sigma[\![\ell \leftarrow v^{\text{DF}}]\!]\rangle & \textsc{Force-update} \\
\langle E^{\text{DF}}[(\texttt{force}\ v^{\text{DF}})], \sigma\rangle\ , v^{\text{DF}} \notin \text{loc} & \longmapsto_{\text{CS}} & \langle E^{\text{DF}}[v^{\text{DF}}], \sigma\rangle & \textsc{Force-val}
\end{array}$$

Fig. 5. CS Machine Transitions

`force` evaluation context. This special context saves the store location of the forced expression so the store can be updated with the resulting value. The outer `force` context is retained in case there are nested `delay`s.

4.2 Continuation Marks

A stepper for a functional language needs access to the control stack of its evaluator to reconstruct the evaluation steps. One implementation technique is to grant complete, privileged access to the control stack. As Clements [6] argued, however, such privileged access is unnecessary and often undesirable.

Continuation marks enable the implementation of stack-accessing tools without granting privileged stack access. The stepper for Lazy Racket utilizes continuation marks to get a progam's stack trace. There are two operations:

1. *store* a value in the current frame of the control stack,
2. *retrieve* all stored continuation marks.

The eager Racket stepper first annotates a source program with *store* and *retrieve* operations at appropriate points. Then, at each *retrieve* point, the stepper reconstructs a reduction step from the information in the continuation marks. The lazy stepper extends this model with `delay` and `force` constructs. The annotation and reconstruction functions are defined in section 5.

4.3 Racket + `delay`/`force` + Continuation Marks

The language λ_{CM} extends λ_{DF} in a stratified manner and models the language for programs annotated with continuation marks:

$$e^{\text{CM}} = e^{\text{RKT}} \mid (\texttt{ccm}) \mid (\texttt{wcm}\ e^{\text{CM}}\ e^{\text{CM}}) \mid (\texttt{output}\ e^{\text{CM}}) \mid (\texttt{loc?}\ e^{\text{CM}})$$

λ_{CM} adds four additional kinds of expressions to λ_{DF}. When a `wcm`, or "with continuation mark", expression is evaluated, its first argument is evaluated and stored in the current stack frame before its second argument is evaluated. A `ccm` expression evaluates to a list of all continuation marks currently stored in the stack. When reducing an `output` expression, its argument is evaluated and sent to an output channel; its result is inconsequential. The `loc?` predicate identifies locations and is needed by annotated programs.

4.4 CSKM Machine

One way to model continuation marks requires an explicit control stack. Hence, we first convert our CS machine to a CSK machine, where the evaluation context is separated from the control string, relocated to a new K register, and converted to a stack of frames. The conversion to a CSK machine is straightfoward [8]. In addition, we pair each context with a continuation mark m, which is stored in a fourth "M" register, giving us a CSKM machine. Here are all the continuations:

$$\begin{aligned} K^{\text{CM}} = {} & \texttt{mt} \mid (\texttt{app1}\ c^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{app2}\ v^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{if}\ c_1^{\text{CM}}\ c_2^{\text{CM}}\ K^{\text{CM}}\ m) \\ & \mid (\texttt{prim2-1}\ p2\ c^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{prim2-2}\ p2\ v^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{prim1}\ p1\ K^{\text{CM}}\ m) \\ & \mid (\texttt{cons1}\ c^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{cons2}\ v^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{loc?}\ K^{\text{CM}}\ m) \\ & \mid (\texttt{force}\ K^{\text{CM}}\ m) \mid (\texttt{force}\ \ell\ K^{\text{CM}}\ m) \mid (\texttt{wcm}\ c^{\text{CM}}\ K^{\text{CM}}\ m) \mid (\texttt{output}\ K^{\text{CM}}\ m) \end{aligned}$$

The configurations of the CSKM machine are:

$\mathcal{P}^{\text{CM}} = \langle c^{\text{CM}}, \sigma, K^{\text{CM}}, m\rangle$	(Machine States)	$\mathcal{S}^{\text{CM}} = \mathcal{P}_1^{\text{CM}}, \ldots, \mathcal{P}_n^{\text{CM}}$	(Transition Seq.)
$c^{\text{CM}} = e^{\text{CM}} \mid \ell$	(Control Strings)	$v^{\text{CM}} = v^{\text{DF}}$	(Values)
$\sigma = ((\ell, c^{\text{CM}}), \ldots)$	(Stores)	$m = \emptyset \mid v^{\text{CM}}$	(Cont. Marks)

Control strings are again extended to include location expressions, values are the same as CS machine values, and stores map locations to control string expressions. The marks m are either empty or values.

The transitions for our CSKM machine are in figure 6. For space reasons, we only include the transitions for the new constructs: `wcm`, `ccm`, `output`, and `loc?`. The other transitions are easily derived from the transitions for the CS machine [8]. To formally model output, we tag each transition, making our machine a labeled transition system [13]. When the machine emits output, the transition is tagged with the outputted value; otherwise, the tag is $\emptyset$.

	$\longmapsto_{\text{CSKM}}$		
$\langle(\texttt{wcm}\ c_1^{\text{CM}}\ c_2^{\text{CM}}), \sigma, K^{\text{CM}}, m\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle c_1^{\text{CM}}, \sigma, (\texttt{wcm}\ c_2^{\text{CM}}\ K^{\text{CM}}\ m), \emptyset\rangle$	WCM:EXP
$\langle v^{\text{CM}}, \sigma, (\texttt{wcm}\ c^{\text{CM}}\ K^{\text{CM}}\ m), m'\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle c^{\text{CM}}, \sigma, K^{\text{CM}}, v^{\text{CM}}\rangle$	WCM:VAL
$\langle(\texttt{ccm}), \sigma, K^{\text{CM}}, m\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle v^{\text{CM}}, \sigma, K^{\text{CM}}, \emptyset\rangle$, $v^{\text{CM}}{=}\pi(K^{\text{CM}}, m)$	CCM
$\langle(\texttt{output}\ c^{\text{CM}}), \sigma, K^{\text{CM}}, m\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle c^{\text{CM}}, \sigma, (\texttt{output}\ K^{\text{CM}}\ m), \emptyset\rangle$	OUT:EXP
$\langle v^{\text{CM}}, \sigma, (\texttt{output}\ K^{\text{CM}}\ m), m'\rangle$	$\xrightarrow{v^{\text{CM}}}_{\text{CSKM}}$	$\langle 42, \sigma, K^{\text{CM}}, m\rangle$	OUT:VAL
$\langle(\texttt{loc?}\ c^{\text{CM}}), \sigma, K^{\text{CM}}, m\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle c^{\text{CM}}, \sigma, (\texttt{loc?}\ K^{\text{CM}}\ m), \emptyset\rangle$	LOC:EXP
$\langle \ell, \sigma, (\texttt{loc?}\ K^{\text{CM}}\ m), m'\rangle$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle \texttt{true}, \sigma, K^{\text{CM}}, m\rangle$	LOC-T:VAL
$\langle v^{\text{CM}}, \sigma, (\texttt{loc?}\ K^{\text{CM}}\ m), m'\rangle$, $v^{\text{CM}}{\notin}\,\ell$	$\xrightarrow{\emptyset}_{\text{CSKM}}$	$\langle \texttt{false}, \sigma, K^{\text{CM}}, m\rangle$	LOC-F:VAL

Fig. 6. CSKM Machine Transitions

The starting state for a program e^{CM} is $\langle e^{\text{CM}}, (\ ), \texttt{mt}, \emptyset\rangle$; evaluation halts when the control string is a value and the control stack is `mt`. The transition sequence for a program is again deterministic.

The WCM:EXP transition sets the first argument as the control string and saves the second argument in a `wcm` continuation. In the resulting machine configuration, the M register is set to $\emptyset$ because a new frame is pushed onto the stack. When evaluation of the first `wcm` argument is complete, the resulting value is set as the new continuation mark, as dictated by the WCM:VAL transition, overwriting any previous mark. The CCM transition uses the π function to retrieve all continuation marks in the stack. The π function is defined as follows:

$$\begin{aligned}(\pi\ \texttt{mt}\ m) &= (\texttt{cons}\ m\ \texttt{null})\\ (\pi\ (\texttt{app1}\ c^{\text{CM}}\ K^{\text{CM}}\ m)\ m') &= (\texttt{cons}\ m'\ (\pi\ K^{\text{CM}}\ m))\\ (\pi\ (\texttt{app2}\ v^{\text{CM}}\ K^{\text{CM}}\ m)\ m') &= (\texttt{cons}\ m'\ (\pi\ K^{\text{CM}}\ m))\\ &\ \ \dots\\ (\pi\ (\texttt{loc?}\ K^{\text{CM}}\ m)\ m') &= (\texttt{cons}\ m'\ (\pi\ K^{\text{CM}}\ m))\end{aligned}$$

Only the first few cases are shown. The rest of the definition is similarly defined. The OUT:EXP transition sets the argument in an `output` expression as the control string and pushes a new `output` continuation frame onto the control stack. Again, the continuation mark register is initialized to $\emptyset$ due to the new stack frame. When the output expression is evaluated, the resulting value is emitted as output, as modeled by the tag on the OUT:VAL transition.

5 Correctness

Our algebraic stepper comes with a concise specification: the λ_{LR} rewriting system. Thus, it is relatively straightforward to state a correctness theorem for the stepper. Let ζ be a label-stripping function.

Theorem 1 (Stepper Correctness). *If for some Lazy Racket program e, the stepper shows the sequence* $e, \zeta[\![e_1^{\text{LR}}]\!], \dots, \zeta[\![e_n^{\text{LR}}]\!]$, *then* $e \mapsto\!\!\!\!\twoheadrightarrow_{\text{LR}} e_1^{\text{LR}} \mapsto\!\!\!\!\twoheadrightarrow_{\text{LR}} \cdots \mapsto\!\!\!\!\twoheadrightarrow_{\text{LR}} e_n^{\text{LR}}$.

The theorem statement involves multistep rewriting because some steps, such as CONS, merely add labels and change nothing else about the term. The proof of the theorem consists of two lemmas. First, we show that a CS machine reduction sequence can be reconstructed from the output of an annotated version of that program running on a CSKM machine. Second, we prove that this reduction sequence is equivalent to the rewriting sequence of the original Lazy Racket program, modulo label assignment. The following subsections spell out the two lemmas and present proof sketches.

5.1 Annotation and Reconstruction Correctness

To state the first correctness lemma, we need three functions. First, $\mathcal{T}$ consumes CSKM steps and produces a trace of output values:

$$\mathcal{T}[\![\cdots \longmapsto_{\text{CSKM}} \mathcal{P}_i^{\text{CM}} \xrightarrow{v^{\text{CM}}}_{\text{CSKM}} \mathcal{P}_{i+1}^{\text{CM}} \longmapsto_{\text{CSKM}} \cdots]\!] = \dots, v^{\text{CM}}, \dots \quad \boxed{\mathcal{T} : \mathcal{S}^{\text{CM}} \to v_1^{\text{CM}}, \dots, v_n^{\text{CM}}}$$

Second, $\mathcal{A} : e^{\text{RKT}} \to e^{\text{CM}}$ annotates a Racket program and $\mathcal{R} : v_1^{\text{CM}}, \dots, v_n^{\text{CM}} \to \mathcal{S}^{\text{DF}}$ reconstructs a CS machine transition sequence from a list of output values.

Lemma 2 (Annotation/Reconstruction Correctness).
For any Racket program e^{RKT}*, if* $\langle e^{\text{RKT}}, (\,)\rangle \longmapsto_{\text{CS}} \cdots \longmapsto_{\text{CS}} \mathcal{P}^{\text{DF}}$*, then:*

$$\mathcal{R}[\![\mathcal{T}[\![\langle \mathcal{A}[\![e^{\text{RKT}}]\!], (\,), \mathit{mt}, \emptyset\rangle \longmapsto_{\text{CSKM}} \cdots \longmapsto_{\text{CSKM}} \mathcal{P}^{\text{CM}}]\!]]\!] = \langle e^{\text{RKT}}, (\,)\rangle \longmapsto_{\text{CS}} \cdots \longmapsto_{\text{CS}} \mathcal{P}^{\text{DF}}$$

Our annotation and reconstruction functions extend that of Clements [6]. Annotation adds `output` and continuation mark `wcm` and `ccm` operations to a program. For example, annotating the program (+ 1 2) results in the following:[4]

```
(let* ([t0 (output (cons Q⟦(+ 1 2)⟧ (ccm)))]
       [v1 (+ 1 2)]
       [t1 (output (cons Q⟦v1⟧ (ccm)))])
  v1)
```

Annotated programs utilize the quoting function, $\mathcal{Q}$, for converting an expression to a value representation. For example $\mathcal{Q}[\![(+\ 1\ 2)]\!]$ = (`list "+"` 1 2). There is also an inverse function, $\mathcal{Q}^{-1}$, for reconstruction. The above annotated program evaluates to 3, outputting the values $\mathcal{Q}[\![(+\ 1\ 2)]\!]$ and $\mathcal{Q}[\![3]\!]$ in the process, from which the reduction sequence (+ 1 2) → 3 can be recovered. The (`ccm`) calls in the example return the empty list since there were no calls to `wcm`. There is no need for `wcm` annotations because the entire program is a redex.

Extending the example to (+ (+ 1 2) 5) yields:

```
(let* ([v0 (wcm (list "prim2-1" "+" 5)
                (let* ([t0 (output (cons Q⟦(+ 1 2)⟧ (ccm)))]
                       [v1 (+ 1 2)]
                       [t1 (output (cons Q⟦v1⟧ (ccm)))])
                  v1))]
       [v2 (+ v0 5)]
       [t2 (output (cons Q⟦v2⟧ (ccm)))])
  v2)
```

This extended example contains the first example as a subexpression and therefore, the annotated version of the program contains the annotated version of the first example. The (+ 1 2) expression now occurs in the context (+ [ ] 5) and the `wcm` expression stores an appropriate continuation mark. The `"prim2-1"` label indicates that the hole is in the left argument position. The first `output` expression now produces the output value (`list` $\mathcal{Q}[\![(+\ 1\ 2)]\!]$ (`list "prim2-1" "+"` 5)), which can be reconstructed to the expression (+ (+ 1 2) 5). Reconstructing all outputs produces (+ (+ 1 2) 5) → (+ 3 5) → 8.

The context information stored in continuation marks is used to reconstruct machine states and the reconstruction and annotation functions defined in figures 7 and 8 demonstrate how this works for `force` and `delay` expressions. If a forced expression e^{RKT} does not evaluate to a location, the annotations are like those for the above examples. If e^{RKT} evaluates to a location, additional continuation marks (figure 7, boxes 1 and 2) are needed to indicate the presence of `force` contexts during the evaluation of the delayed computation. An additional `output` expression (box 3) is also needed so that steps showing the removal of both the (`force` [ ]) and (`force` ℓ [ ]) contexts can be reconstructed. With

[4] For clarity, we use some syntactic sugar here (`let*` and `list`).

(`rest` (`ccm`)) (box 5), we ensure that the (`force` ℓ [ ]) context is not part of the reconstructed expression. The location $v0$ (box 4) is included in the output so the store can be properly reconstructed. The `"val"` tag directs the reconstruction function to use the value $\mathcal{Q}[\![v2]\!]$ from the emitted location-value pair during reconstruction.

```
$\mathcal{A}[\![$(force $e^{\text{RKT}}$)$]\!]$ =                                              $\boxed{\mathcal{A} : e^{\text{RKT}} \to e^{\text{CM}}}$
  (let* ([$v0$ (wcm (list "force") $\mathcal{A}[\![e^{\text{RKT}}]\!]$)]
         [$v1$ (if ($not$ (loc? $v0$))
                 $v0$
                 (wcm $\boxed{\texttt{(list "force")}}_1$
                   (wcm $\boxed{\texttt{(list "force" } v0\texttt{)}}_2$
                     (let* ([$v2$ (force $v0$)] ; v0 is location
                            [$t0$ ($\boxed{\texttt{output}}_3$ (cons (list "val" $\boxed{v0}_4$ $\mathcal{Q}[\![v2]\!]$)
                                                   $\boxed{\texttt{(rest (ccm))}}_5$ ))])
                       $v2$))))]
         [$t1$ (output (cons $\mathcal{Q}[\![v1]\!]$ (ccm)))])
    $v1$)

$\mathcal{A}[\![$(delay $e^{\text{RKT}}$)$]\!]$ =
  (let* ([$t0$ (output (cons $\mathcal{Q}[\![$(delay $e^{\text{RKT}}$)$]\!]$ (ccm)))]
         [$\ell$ (alloc)]
         [$t1$ (output (cons (list "loc" $\ell$ $\boxed{\mathcal{Q}[\![e^{\text{RKT}}]\!]}_6$) (ccm)))])
    (delay $\mathcal{A}[\![e^{\text{RKT}}]\!]$))
```

Fig. 7. Annotation function for `delay` and `force`.

The annotation of a `delay` expression requires predicting the location of the delayed compuation in the store. We therefore assume we have access to an `alloc` function that uses the same location-allocating algorithm as the memory management system of the machine.[5] In addition to the location, the delayed expression itself (box 6) is also included in the output, to enable reconstruction of the store. The `"loc"` tag directs the reconstruction function to use the location from the emitted location-value pair when reconstructing the control string.

The reconstruction function in figure 8 consumes a list of values, where each value is a sublist and reconstructs a CS machine state from each sublist. The first element of every v_i^{CM} sublist represents a (quoted) expression that is plugged into the context represented by the rest of the sublist. The store is reconstructed by retrieving all the location-value pairs in all the sublists up to the current one. The arguments to the store-reconstruction function $\mathcal{R}_S$ may contain duplicate entries for a location, so a location-value pair is only included in the resulting store if it does not occur in any subsequent arguments.

[5] Since labels are displayed as sharing, this unrealistic assumption is acceptable.

$$\mathcal{R}[\![\ldots, v_i^{\text{CM}}, \ldots]\!] = \ldots, \langle \mathcal{R}_E[\![(\texttt{rest}\ v_i^{\text{CM}})]\!][\mathcal{R}_C[\![(\texttt{first}\ v_i^{\text{CM}})]\!]], \quad \boxed{\mathcal{R} : v_1^{\text{CM}}, \ldots, v_n^{\text{CM}} \to \mathcal{S}^{\text{DF}}}$$
$$\mathcal{R}_S[\![v_1^{\text{CM}}, \ldots, v_i^{\text{CM}}]\!]\rangle, \ldots$$

$$\mathcal{R}_E[\![(\texttt{cons}\ (\texttt{list}\ \text{“force”})\ v^{\text{CM}})]\!] = (\texttt{force}\ \mathcal{R}_E[\![v^{\text{CM}}]\!]) \qquad \boxed{\mathcal{R}_E : v^{\text{CM}} \to E^{\text{DF}}}$$
$$\mathcal{R}_E[\![(\texttt{cons}\ (\texttt{list}\ \text{“force”}\ \ell)\ v^{\text{CM}})]\!] = (\texttt{force}\ \ell\ \mathcal{R}_E[\![v^{\text{CM}}]\!])$$

$$\mathcal{R}_C[\![(\texttt{list}\ \text{“val”}\ \ell\ v^{\text{DF}})]\!] = \mathcal{Q}^{-1}[\![v^{\text{DF}}]\!] \qquad \boxed{\mathcal{R}_C : v^{\text{CM}} \to e^{\text{DF}}}$$
$$\mathcal{R}_C[\![(\texttt{list}\ \text{“loc”}\ \ell\ v^{\text{DF}})]\!] = \ell$$
$$\text{otherwise,}\ \mathcal{R}_C[\![v^{\text{DF}}]\!] = \mathcal{Q}^{-1}[\![v^{\text{DF}}]\!]$$

$$\mathcal{R}_S[\![(\texttt{cons}\ (\texttt{list}\ s\ \ell\ v^{\text{DF}\prime})\ \_), v_{rest}^{\text{DF}}, \ldots]\!] \qquad \boxed{\mathcal{R}_S : v_1^{\text{CM}}, \ldots, v_n^{\text{CM}} \to \sigma}$$
$$= \begin{cases} (\texttt{cons}\ (\ell, \mathcal{Q}^{-1}[\![v^{\text{DF}\prime}]\!])\ \mathcal{R}_S[\![v_{rest}^{\text{DF}}, \ldots]\!]), & \text{if } \ell \notin \texttt{dom}(\mathcal{R}_S[\![v_{rest}^{\text{DF}}, \ldots]\!]) \\ \mathcal{R}_S[\![v_{rest}^{\text{DF}}, \ldots]\!] & \text{if } \ell \in \texttt{dom}(\mathcal{R}_S[\![v_{rest}^{\text{DF}}, \ldots]\!]) \end{cases}$$
$$\mathcal{R}_S[\![v^{\text{DF}}, v_{rest}^{\text{DF}}, \ldots]\!] = \mathcal{R}_S[\![v_{rest}^{\text{DF}}, \ldots]\!], \qquad \text{if } (\texttt{first}\ v^{\text{DF}}) \neq (\texttt{list}\ s\ \ell\ v^{\text{DF}\prime})$$

Fig. 8. Reconstruction function for `delay` and `force`.

The proof of lemma 2 extends Clements's proof [5, Section 3.4] with cases for `delay` and `force`. The original cases must cope with the additional store, but this is straightforward.

5.2 Lazy Racket Correctness

The φ function in figure 9 macro-expands a Lazy Racket program. Because source terms don't include labels, φ is undefined for labeled terms. Its partial inverse, φ^{-1}, defined in figure 10, synthesizes an unlabeled Lazy Racket program from a (CS machine representation of a) Racket program.

Lemma 3 states Lazy Racket's correctness in terms of φ, φ^{-1}, the label-stripping function ζ, and the λ_{LR} rewriting system. That is, every CS machine sequence has an equivalent λ_{LR} rewriting sequence, modulo φ^{-1} and ζ.

$$\varphi[\![\lambda x.e]\!] = \lambda x.\varphi[\![e]\!] \qquad \boxed{\varphi : e \to e^{\text{RKT}}}$$
$$\varphi[\![(e_1\ e_2)]\!] = ((\texttt{force}\ \varphi[\![e_1]\!])\ (\texttt{delay}\ \varphi[\![e_2]\!]))$$
$$\varphi[\![(\texttt{cons}\ e_1\ e_2)]\!] = (\texttt{cons}\ (\texttt{delay}\ \varphi[\![e_1]\!])\ (\texttt{delay}\ \varphi[\![e_2]\!]))$$
$$\varphi[\![(\texttt{if}\ e_1\ e_2\ e_3)]\!] = (\texttt{if}\ (\texttt{force}\ \varphi[\![e_1]\!])\ \varphi[\![e_2]\!]\ \varphi[\![e_3]\!])$$
$$\varphi[\![(p^n\ e\ \ldots)]\!] = (p^n\ (\texttt{force}\ \varphi[\![e]\!])\ \ldots)$$
$$\text{otherwise,}\ \varphi[\![e]\!] = e$$

Fig. 9. Macro-expanding Lazy Racket to plain Racket.

$$\varphi^{-1}[\![c^{\mathrm{DF}}]\!]\sigma = e, \text{ where } \langle e, \_\rangle = \bar{\varphi}[\![c^{\mathrm{DF}}]\!]\sigma \qquad \boxed{\varphi^{-1} : c^{\mathrm{DF}} \times \sigma \to e}$$

$$\boxed{\bar{\varphi} : c^{\mathrm{DF}} \times \sigma \to \langle e, \sigma\rangle}$$

$$\begin{aligned}
\bar{\varphi}[\![\lambda x.e^{\mathrm{DF}}]\!]\sigma &= \langle \lambda x.e, \sigma\rangle\,, && \text{where } \langle e, \sigma\rangle = \bar{\varphi}[\![e^{\mathrm{DF}}]\!]\sigma \\
\bar{\varphi}[\![(c_1^{\mathrm{DF}}\ c_2^{\mathrm{DF}})]\!]\sigma &= \big\langle (e_1\ e_2), \sigma''\big\rangle\,, && \text{where } \big\langle e_1, \sigma'\big\rangle = \bar{\varphi}[\![c_1^{\mathrm{DF}}]\!]\sigma,\ \big\langle e_2, \sigma''\big\rangle = \bar{\varphi}[\![c_2^{\mathrm{DF}}]\!]\sigma' \\
\bar{\varphi}[\![(p^2\ c_1^{\mathrm{DF}}\ c_2^{\mathrm{DF}})]\!]\sigma &= \big\langle (p^2\ e_1\ e_2), \sigma''\big\rangle\,, && \text{where } \big\langle e_1, \sigma'\big\rangle = \bar{\varphi}[\![c_1^{\mathrm{DF}}]\!]\sigma,\ \big\langle e_2, \sigma''\big\rangle = \bar{\varphi}[\![c_2^{\mathrm{DF}}]\!]\sigma' \\
\bar{\varphi}[\![(\texttt{cons}\ c_1^{\mathrm{DF}}\ c_2^{\mathrm{DF}})]\!]\sigma &= \big\langle (\texttt{cons}\ e_1\ e_2), \sigma''\big\rangle, && \text{where } \big\langle e_1, \sigma'\big\rangle = \bar{\varphi}[\![c_1^{\mathrm{DF}}]\!]\sigma,\ \big\langle e_2, \sigma''\big\rangle = \bar{\varphi}[\![c_2^{\mathrm{DF}}]\!]\sigma' \\
\bar{\varphi}[\![(p^1\ c^{\mathrm{DF}})]\!]\sigma &= \big\langle (p^1\ e), \sigma'\big\rangle\,, && \text{where } \big\langle e, \sigma'\big\rangle = \bar{\varphi}[\![c^{\mathrm{DF}}]\!]\sigma \\
\bar{\varphi}[\![(\texttt{if}\ c_1^{\mathrm{DF}}\ e_2^{\mathrm{DF}}\ e_3^{\mathrm{DF}})]\!]\sigma &= \big\langle (\texttt{if}\ e_1\ e_2\ e_3), \sigma''\big\rangle, && \text{where } \big\langle e_1, \sigma'\big\rangle = \bar{\varphi}[\![c_1^{\mathrm{DF}}]\!]\sigma,\ \big\langle e_2, \sigma'\big\rangle = \bar{\varphi}[\![e_2^{\mathrm{DF}}]\!]\sigma' \\
& && \qquad\qquad\qquad\qquad\qquad\ \ \big\langle e_3, \sigma'\big\rangle = \bar{\varphi}[\![e_3^{\mathrm{DF}}]\!]\sigma' \\
\bar{\varphi}[\![(\texttt{delay}\ e^{\mathrm{DF}})]\!]\sigma &= \bar{\varphi}[\![e^{\mathrm{DF}}]\!]\sigma \\
\bar{\varphi}[\![\ell]\!]\sigma &= \bar{\varphi}[\![\sigma[\![\ell]\!]]\!]\sigma \\
\bar{\varphi}[\![(\texttt{force}\ c^{\mathrm{DF}})]\!]\sigma &= \bar{\varphi}[\![c^{\mathrm{DF}}]\!]\sigma \\
\bar{\varphi}[\![(\texttt{force}\ \ell\ c^{\mathrm{DF}})]\!]\sigma &= \big\langle e, \sigma'[\![\ell \leftarrow e]\!]\big\rangle\,, && \text{where } \big\langle e, \sigma'\big\rangle = \bar{\varphi}[\![c^{\mathrm{DF}}]\!]\sigma \\
\text{otherwise, }\ \bar{\varphi}[\![e^{\mathrm{DF}}]\!]\sigma &= \big\langle e^{\mathrm{DF}}, \sigma\big\rangle
\end{aligned}$$

Fig. 10. Synthesizing Lazy Racket from plain Racket.

Lemma 3 (LR Correctness). *For all Lazy Racket programs* e *and Racket programs* c^{DF} *such that* $\langle \varphi[\![e]\!], (\,)\rangle \longmapsto\!\!\!\!\rightarrow_{\mathrm{CS}} \langle c^{\mathrm{DF}}, \sigma\rangle$, *there exists a Lazy Racket program* e^{LR} *such that* $e \longmapsto\!\!\!\!\rightarrow_{\mathrm{LR}} e^{\mathrm{LR}}$ *and* $\varphi^{-1}[\![c^{\mathrm{DF}}]\!]\sigma = \zeta[\![e^{\mathrm{LR}}]\!]$.

Proof (Sketch). We prove the lemma by induction on the number of CS machine steps. For the base case, the lemma holds because $\varphi^{-1}[\![\varphi[\![e]\!]]\!](\,) = e$. Otherwise, we proceed by case analysis on the last transition step. For each case, we prove correct synthesis of evaluation contexts and redexes separately. □

6 Related Work

Researchers have developed many debugging tools for lazy languages. However, few of these tools show laziness during evaluation or do so in a intuitive manner. Some tools present declarative traces [18, 17, 23, 25] that show a reduction sequence for each expression in the program but removes all notions of temporal ordering of the reductions, a key element to understanding lazy evaluation.

Declarative tools are popular because researchers struggle to portray laziness operationally. The most basic operational tool records a trace of function calls during evaluation [14]. Unlike declarative debuggers, these tools explain the lazy evaluation of expressions relative to other expressions; however, the complexity of laziness often makes such tools confusing. In reaction, some researchers developed stack tracing tools that approximate a call-by-value language [1, 19].

Operational-style tools similar to ours show the step-by-step evaluation of lazy programs. Unfortunately, many of these hide the laziness during evaluation as well [3, 7, 11]. Snyder [22] developed a tool that resembles ours in that it reconstructs reduction sequences from an annotated program. It uses graph combinators, however, and the tool is only able to reconstruct an approximation of the source code. Lapalme and Latendresse [15] developed a tool that, like ours, inserts “breakpoints” to generate a sequence of reductions but they do not mention how the steps are determined, nor do they show actual examples. The GHCi debugger [16] shows laziness while stepping through a program, but presents it in terms of the low-level implementation. This produces steps that both users and the authors of the tool consider confusing. Foubister [10] and Taylor [24] developed graph reduction stepper tools, but programmers prefer textual tools [11, 20], especially for teaching novices.

Gibbons and Wansbrough created a lazy stepper based on the call-by-need calculus of Ariola and Felleisen [2]. While the calculus is useful for reasoning about equivalences of lazy expressions, it can be confusing to use for reasoning about lazy executions for three reasons. First, the calculus includes administrative transformation steps that do not represent real computation the way substitution does. Second, to express sharing, the calculus never resolves function calls and instead retains all arguments long after they become superfluous. Third, the calculus dereferences variables one at a time, which can be confusing since all instances of a particular variable are supposed to represent just one expression. Watson’s tool [26] manages to eliminate the administrative steps and the persistent arguments at the expense of explicit sharing. Shared terms are held in a implicit store so programmers are forced to memorize those terms. An argument seemingly reappears as each variable is dereferenced, which can potentially be confusing. Penney [21] improves Watson’s visualization using “where” clauses to explicitly show shared terms but tracking such clauses as the number of placeholders accumulate seems like it would be difficult. No tool uses full substitution and parallel rewriting steps to represent laziness.

Finally, few tools come with formal models and correctness proofs for their architecture. Chitil and Luo [4] developed a model for the declarative Hat debugger’s trace generator and show that the evaluation steps can be reconstructed from the information in the traces. Watson also provides formal definitions for his transformations [26], and uses a standard store-based operational semantics for the rewriting steps.

References

1. Allwood, T.O., Peyton Jones, S., Eisenbach, S.: Finding the needle: stack traces for GHC. In: Proc. 2nd Symp. on Haskell. pp. 129–140 (2009)
2. Ariola, Z.M., Felleisen, M., Maraist, J., Odersky, M., Wadler, P.: The call-by-need λ calculus. In: Proc. 22nd POPL. pp. 233–246 (1995)
3. Augustson, M., Reinfelds, J.: A visual miranda machine. In: Proc. Software Education Conference SRIG-ET. pp. 233–246 (1995)

4. Chitil, O., Luo, Y.: Structure and properties of traces for functional programs. In: Proc. 3rd Intl. Works. Term Graph Rewriting. pp. 39–63 (2006)
5. Clements, J.: Portable and High-level Access to the Stack with Continuation Marks. Ph.D. thesis, Northeastern University (2006)
6. Clements, J., Flatt, M., Felleisen, M.: Modeling an algebraic stepper. In: Proc. 10th ESOP. pp. 320–334 (2001)
7. Ennals, R., Peyton Jones, S.: HsDebug: debugging lazy programs by not being lazy. In: Proc. Works. on Haskell. pp. 84–87 (2003)
8. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)
9. Felleisen, M., Friedman, D.P.: A syntactic theory of sequential state. Theor. Comput. Sci. 69(3), 243–287 (1989)
10. Foubister, S.P.: Graphical Application and Visualisation of Lazy Functional Computation. Ph.D. thesis, University of York (1995)
11. Goldson, D.: A symbolic calculator for non-strict functional languages. Comput. J. 37(3), 177–187 (1994)
12. Hughes, J.: Why functional programming matters. Comput. J. 32(2), 98–107 (1989)
13. Keller, R.: Formal verification of parallel programs. Commun. ACM 19(7) (1976)
14. Kishon, A.: Theory and Art of Semantics-Directed Program Execution Monitoring. Ph.D. thesis, Yale University (1992)
15. Lapalme, G., Latendresse, M.: A debugging environment for lazy functional languages. LISP and Symbolic Computation 5(3), 271–287 (1992)
16. Marlow, S., Iborra, J., Pope, B., Gill, A.: A lightweight interactive debugger for Haskell. In: Proc. Works. on Haskell. pp. 13–24 (2007)
17. Naish, L.: Declarative debugging of lazy functional programs. In: Proc. 16th ACSC (1993)
18. Nilsson, H., Fritzson, P.: Algorithmic debugging for lazy functional languages. In: Proc. 4th PLIPL. pp. 385–399 (1992)
19. O'Donnell, J.T., Hall, C.V.: Debugging in applicative languages. LISP and Symbolic Computation 1(2), 113–145 (1988)
20. Patel, M.J., du Boulay, B., Taylor, C.: Effect of format on information and problem solving. In: Proc. 13th Conf. of the Cognitive Science Society. pp. 852–856 (1991)
21. Penney, A.: Augmenting Trace-based Functional Debugging. Ph.D. thesis, University of Bristol, Australia (1999)
22. Snyder, R.M.: Lazy debugging of lazy functional programs. New Generation Computing 8, 139–161 (1990)
23. Sparud, J., Runciman, C.: Tracing lazy functional computations using redex trails. In: Proc. 9th PLILP. pp. 291–308 (1997)
24. Taylor, J.P.: Presenting the Lazy Evaluation of Functions. Ph.D. thesis, Queen Mary and Westfield College (1996)
25. Wallace, M., Chitil, O., Brehm, T., Runciman, C.: Multiple-view tracing for Haskell: a new hat. In: Proc. Works. on Haskell. pp. 151–170 (2001)
26. Watson, R.D.: Tracing Lazy Evaluation by Program Transformation. Ph.D. thesis, Southern Cross University, Australia (1997)