The Memorist Tale: Every Thunk Every Cost All At Once

Xing Li[1], Yao Li[2], Peter Schachte[1], and Christine Rizkallah[1]

[1] University of Melbourne
[2] Portland State University

Abstract. Lazy evaluation offers great flexibility by computing only what is necessary. However, analysing the cost of lazy programs is notoriously challenging, as computation occurs out of order and depends on future demands. Recent work has proposed alternative semantics for modelling lazy evaluation cost that avoid reasoning about program states. However, existing approaches either rely on nondeterminism or require complex bidirectional semantics. We present the *Memorist Semantics*, a novel semantics for analysing the cost of lazy programs by explicitly tracking the cost and dependencies of every subterm. Our semantics annotates components of a term with fine-grained cost and usage information, yielding a deterministic semantics that can be expressed through a simple monadic interface. We formalize the semantics in Rocq and verify its soundness with respect to the existing Clairvoyance Semantics. Similar to prior formalized semantics, our semantics is defined for a total, typed language with built-in structural recursion and without support for first-class functions. We outline ideas for possible extensions.

Keywords: computation cost · lazy evaluation · operational semantics

1 Introduction

It is challenging to analyse computation cost of functional programs. Functional programs typically abstract away *how* programs evaluate. However, the computation cost of a program depends partly on the evaluation strategy adopted. The call-by-need strategy, used by lazy languages such as Haskell, enables expressive and flexible programming by avoiding unnecessary computation [20]. Unlike call-by-value which is used by eager languages such as Standard ML, call-by-need ensures that components of a term are evaluated only when necessary. This allows for elegant program composition and potential efficiency improvements, but at a cost: analysing the cost of lazy programs is notoriously difficult.

To explain the difference, we consider the following example. Let `truePrefix` be a function that takes a list of Booleans as input and returns the prefix of the list that only contains `true`. The function `truePrefixAppend` takes two lists of Booleans, concatenates them using `append`, and applies `truePrefix` on the resulting list. Figure 1 presents an implementation in Rocq Prover (formerly Coq) [27]. Now consider the program: `truePrefixAppend [true;false] [true]`.

```
Fixpoint append {A} (xs ys : list A)
  : list A :=
  match xs with
  | [] => ys
  | x::xs' =>
      let zs := append xs' ys
      in x::zs
  end.
```

```
Fixpoint truePrefix (xs : list bool)
  : list bool :=
  match xs with
  | [] => []
  | x::xs' =>
      let zs := truePrefix xs' in
      if x then x::zs else []
  end.
```

```
Definition truePrefixAppend (xs ys : list bool) : list bool :=
  let zs := append xs ys in truePrefix zs.
```

Fig. 1: Rocq definitions of the functions placed in A-normal form.

$$
\begin{aligned}
&\texttt{truePrefixAppend}\ (\texttt{t} :: \texttt{f} :: []) \ (\texttt{t} :: []) \\
&= \checkmark\,\texttt{let}\ zs := \texttt{append}\ (\texttt{t} :: \texttt{f} :: []) \ (\texttt{t} :: []) \ \texttt{in}\ \texttt{truePrefix}\ zs \\
&= \texttt{let}\ zs := \checkmark(\texttt{let}\ z_1 := \texttt{append}\ (\texttt{f} :: []) \ (\texttt{t} :: []) \ \texttt{in}\ \texttt{t} :: z_1)\ \texttt{in}\ \texttt{truePrefix}\ zs \\
&= \texttt{let}\ z_1 := \texttt{append}\ (\texttt{f} :: []) \ (\texttt{t} :: []) \ \texttt{in}\ \texttt{truePrefix}\ (\texttt{t} :: z_1) \\
&= \texttt{let}\ z_1 := \ldots\ \texttt{in}\ \checkmark\texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{t} \looparrowright \texttt{t} :: w_1 \\
&= \texttt{let}\ z_1 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{t} :: w_1 \qquad (*) \\
&= \texttt{let}\ z_1 := \checkmark(\texttt{let}\ z_2 := \texttt{append}\ [] \ (\texttt{t} :: []) \ \texttt{in}\ \texttt{f} :: z_2)\ \texttt{in} \\
&\qquad\ \texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{t} :: w_1 \\
&= \texttt{let}\ z_2 := \texttt{append}\ [] \ (\texttt{t} :: []) \ \texttt{in}\ \texttt{let}\ w_1 := \texttt{truePrefix}\ (\texttt{f} :: z_2)\ \texttt{in}\ \texttt{t} :: w_1 \\
&= \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := \checkmark(\texttt{let}\ w_2 := \texttt{truePrefix}\ z_2\ \texttt{in}\ \texttt{f} \looparrowright \texttt{f} :: w_2)\ \texttt{in}\ \texttt{t} :: w_1 \\
&= \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := (\texttt{let}\ w_2 := \ldots\ \texttt{in}\ [])\ \texttt{in}\ \texttt{t} :: w_1 \\
&= \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{t} :: []
\end{aligned}
$$

Fig. 2: Lazily evaluating the program `truePrefixAppend`. We place a tick ($\checkmark$) immediately after unfolding a step of function application to indicate the incurring of a cost. The notation $e \looparrowright xs$ denotes `if` e `then` xs `else []`. We use `t` and `f` as shorthands for `true` and `false`.

We are interested in the time cost of evaluating this program, which we model by the number of function applications. While Rocq does not prescribe a specific evaluation strategy, we can view this code as a *shallow embedding* of a program in another language. For instance, tools such as `hs-to-coq` [38, 3] can automatically translate Haskell programs into this form.

Under eager evaluation, `truePrefixAppend` first fully computes `append` of the two input lists, evoking 3 calls to `append`. The result is then passed to `truePrefix`. Every element is processed, thus evoking 4 calls to `truePrefix`. Overall, the program evaluates to `[true]` and incurs a total cost of $3 + 4 + 1 = 8$, with one additional call to the top-level `truePrefixAppend`.

In comparison, lazy evaluation is demand-driven: computation is performed based on the demand on the output. Figure 2 illustrates how the evaluation proceeds stepwise. If the result is demanded to its weak-head normal form (WHNF), we only need it to compute to `true::_` without computing its tail, and can stop at the step marked ($*$). Only one step of `truePrefix` and `append` each is needed. Together with one call to `truePrefixAppend`, the total computation cost is 3.

With *sharing* in lazy evaluation, if more, *e.g.,* the full list, is demanded from a later computation, we can resume from what was already computed, *i.e.,* `true::_`. Now we need to compute the tail and fully reduce the result to `true::[]`. This requires one more step of `truePrefix` and of `append` each, making the cost 5 in total. Note that this cost is still smaller than that incurred by eager evaluation, as we never perform computations not necessary for producing the final result, such as computing the recursive call `append [] [true]`.

Challenges. The example shows several challenges in analysing lazy evaluation cost. Firstly, *the cost of lazy evaluation is not local.* Unlike eager evaluation, cost incurred by a function may not happen at its call site but later inside another function's application. The cost also depends on future demand. For these reasons, it is challenging to analyse individual functions in isolation. Secondly, *evaluation steps of different functions are interleaved.* The evaluation steps of `truePrefix` and `append` in Fig. 2 are an example. Furthermore, *the evaluation is stateful.* To model sharing, an evaluation needs to remember what has been previously computed.

Existing approaches. One can treat lazy programs as stateful programs, reasoning about states using some program logics. One example [35] utilizes the Iris$^{\$}$ framework [29] and the Iris separation logic [39]. Another approach [8, 15] employs *equational reasoning*, tracking computation cost by, *e.g.,* using a graded monad. Instead of directly dealing with the stateful natural semantics of laziness [24], we can use an *alternative semantics* equivalent in terms of computation cost. Since it only matters *whether*, but not *when*, a computation happens for time cost analysis, one can *localize* lazy cost if the future demand is known. In this vein, the Clairvoyance Semantics [14], encodable in a simple monadic interface [26], simulates future demands via *nondeterminism.* However, nondeterminism makes formal reasoning and testing challenging. In Li *et al.* [26], an additional logic similar to incorrectness logic [33] is proposed for this reason. To avoid nondeterminism, the Demand Semantics [42] employs bidirectional evaluations: a forward one that computes output values from inputs as normal, and a backward one that calculates *the minimal input demand* and computation cost from pure input and *an output demand.* However, this requires a function to be translated into two different versions, which duplicates code, is error-prone and poses new challenges to formal reasoning. We defer a detailed discussion contrasting these approaches to Section 7.

Our key idea. In this paper, we propose the Memorist Semantics, a novel cost semantics for lazy evaluation that tracks both usage and cost of evaluation of

terms. The key idea is to run an eager evaluation, while giving every piece of data a unique name and 'memorizing' information used for computing the data. The Memorist Semantics enables analysing computation cost *locally* without considering states, similar to the Clairvoyance and Demand Semantics. Meanwhile, the Memorist Semantics improves on the former by being *deterministic*, and improves on the latter by only employing one semantics and requiring no code duplication, thus combining the best of two worlds.

To achieve this, we address two key challenges. The first is *precise cost attribution* for different components of a term or value. Since lazy evaluation may only require part of a term, we must exclude unnecessary computations from cost calculations. For this, we track the cost of computing each component separately and annotate it with its cost. The second challenge is *accounting for shared computations*. Because lazy evaluation evaluates each component of a term at most once, our semantics must prevent duplicating cost accounting. We achieve this by tracking *usage sets* and using set union for bookkeeping.

Contributions. We make the following contributions:

- We introduce the Memorist Semantics, a deterministic cost semantics for call-by-need that tracks both cost and usage of subexpressions (Section 3).
- We prove that our semantics correctly models the execution cost of a program under call-by-need by relating it to the Clairvoyance Semantics (Section 4).
- We show that our semantics can be encoded using a simple monadic interface via a proof of concept implementation in Rocq (Section 5).
- We formalize our results in Rocq to ensure rigorous proofs.

We describe the intuition behind our semantics in Section 2. We discuss limitations and potential ways to address them in Section 6, including lack of handling of general recursion and formalizing first-class functions. We discuss related work in Section 7 and conclude the paper with future work in Section 8.

2 The Memorist Approach

Imagine a person with a remarkably retentive memory, whom we call a *Memorist*. When evaluating a program, the Memorist does all computations eagerly while memorizing for each computation what other computations ("thunks") it requires and its own computation cost. After evaluating the entire program, the Memorist learns all the thunks used by computing each part of the value and their individual computation cost. If some program demands parts of the value, the Memorist can tell all the thunks that are used in the evaluation up to these parts. A corresponding lazy evaluation would also have to evaluate exactly these thunks. One can thus infer the lazy evaluation cost based on this information.

Thunks and annotations. To realize this approach, we wrap every piece of data inside a *thunk*, and track its usage and cost during evaluations. Thunks here are intended to simulate thunks in lazy evaluation, but they are not encoded as suspended computations. Each Memorist thunk has a unique name to distinguish

it from others. We annotate a thunk with a pair of cost and a set of thunk names. The former tracks the cost incurred by evaluating the thunk to its WHNF, *without* the cost incurred by evaluating other thunks. The latter, called a *usage set*, records all thunks whose results are used in that evaluation. We also annotate an output value in the same fashion.

Consider the program `truePrefixAppend [true;false] [true]` from a Memorist's perspective. Since lazy evaluations do not necessarily evaluate every function or constructor argument, we accordingly wrap the input lists and all arguments to the list constructor `::` in thunks. For illustration, we denote a thunked expression x with a thunk name i and an annotation a by $x_{i@a}$. We can then represent the first input list, after being thunked, as

$$\{\!|\mathtt{true}_{i_4@a_{i_4}} :: \{\!|\mathtt{false}_{i_3@a_{i_3}} :: \mathtt{[]}_{i_1@a_{i_1}}|\!\}_{i_2@a_{i_2}}|\!\}_{i_7@a_{i_7}}$$

and the second as $\{\!|\mathtt{true}_{i_6@a_{i_6}} :: \mathtt{[]}_{i_5@a_{i_5}}|\!\}_{i_8@a_{i_8}}$, for some unique names $i_1, \ldots, i_8$ and annotations $a_{i_1}, \ldots, a_{i_8}$. We delay the formal definitions to Section 3.

The Memorist Semantics in action. We evaluate the program eagerly while tracking thunk usage and cost, with detailed steps in Fig. 3. We first compute `append` (Fig. 3a). At each step, we "unthunk" the first argument to access the list inside; *e.g.,* at the first step, we unthunk thunk i_7 wrapping the first argument, and say we *used* i_7. We then recursively apply `append` to the tail of the first input list until we reach the empty list, at which point we unthunk both thunks i_1 and i_8, and return the second input list.

We wrap this result from computing `append` $\mathtt{[]}_{i_1}$ $\{\!|\mathtt{true}_{i_6} :: \mathtt{[]}_{i_5}|\!\}_{i_8}$ (bound to z_2) in a new thunk freshly named j_1. The computation incurs one call to `append`; hence, the cost annotation is 1. It uses the thunks i_1 and i_8. Note that the thunks in the usage sets annotated to i_1 and i_8 must all be used if i_1 and i_8 are used. So we include all of them in j_1's usage set annotation. Denoting union of $\{i_1\}$ and i_1's usage set by $s(i_1)$, j_1's usage set annotation is then $s(i_1) \cup s(i_8)$. The rest of the computation proceeds similarly. The final output is also annotated.

Lazy computation cost. With the information in the annotations, we can analyse the cost of `append` with respect to *any demand* on the output. We do so by collecting all thunks used by a demand and aggregating their individual cost. For example, if we demand the output list to its WHNF, we need only to consider the thunk usage in the output annotation, which is $s(i_7)$. Moreover, we are interested in the cost of `append` itself, not in previous computations of its input. We account for this by removing all the thunks existing prior to the evaluation of `append [true;false] [true]` from $s(i_7)$. This gives us the empty set, suggesting no thunked cost to consider. Therefore, we only need to count the cost annotation to the final output. The inferred lazy cost is thus 1, as expected.

If we demand the output to the first two elements instead, we must additionally take into account the thunk j_2 wrapping the first cons cell, and the thunks i_4 and i_3 wrapping the first and the second element in the output list. That is, we consider all thunks in the set $s(i_7) \cup s(j_2) \cup s(i_4) \cup s(i_3)$. Only j_2 is created

$$
\begin{aligned}
&\texttt{append}\ \{\!|\mathtt{t}_{i_4}@a_{i_4} :: \{\!|\mathtt{f}_{i_3}@a_{i_3} :: []_{i_1}@a_{i_1}|\!\}_{i_2}@a_{i_2}|\!\}_{i_7}@a_{i_7}\ \{\!|\mathtt{t}_{i_6}@a_{i_6} :: []_{i_5}@a_{i_5}|\!\}_{i_8}@a_{i_8} \\
=\ &\checkmark\texttt{let}\ z_1 := \texttt{append}\ \{\!|\mathtt{f}_{i_3} :: []_{i_1}|\!\}_{i_2}\ \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{i_8}\ \texttt{in}\ \mathtt{t}_{i_4} :: \underline{z_1} && \rightsquigarrow i_7 \\
=\ &\texttt{let}\ z_1 := \checkmark(\texttt{let}\ z_2 := \texttt{append}\ []_{i_1}\ \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{i_8}\ \texttt{in}\ \mathtt{f}_{i_3} :: \underline{z_2})\ \texttt{in}\ \mathtt{t}_{i_4} :: \underline{z_1} && \rightsquigarrow i_2 \\
=\ &\texttt{let}\ z_1 := (\texttt{let}\ z_2 := \checkmark\mathtt{t}_{i_6} :: []_{i_5}\ \texttt{in}\ \mathtt{f}_{i_3} :: \underline{z_2})\ \texttt{in}\ \mathtt{t}_{i_4} :: \underline{z_1} && \rightsquigarrow i_1, i_8 \\
=\ &\texttt{let}\ z_1 := \mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1@(1,s(i_1)\cup s(i_8))}\ \texttt{in}\ \mathtt{t}_{i_4} :: \underline{z_1} \\
=\ &\mathtt{t}_{i_4} :: \{\!|\mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1}|\!\}_{j_2@(1,s(i_2))} \qquad @(1, s(i_7))
\end{aligned}
$$

(a) Evaluation steps of append

$$
\begin{aligned}
&\texttt{truePrefix}\ \{\!|\mathtt{t}_{i_4} :: \{\!|\mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1}|\!\}_{j_2}|\!\}_{j_3} \\
=\ &\checkmark\texttt{let}\ w_1 := \texttt{truePrefix}\ \{\!|\mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1}|\!\}_{j_2}\ \texttt{in}\ \mathtt{t}_{i_4} \curvearrowleft \mathtt{t}_{i_4} :: \underline{w_1} && \rightsquigarrow j_3 \\
=\ &\texttt{let}\ w_1 := \checkmark(\texttt{let}\ w_2 := \texttt{truePrefix}\ \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1}\ \texttt{in}\ \mathtt{f}_{i_3} \curvearrowleft \mathtt{f}_{i_3} :: \underline{w_2})\ \texttt{in}\ \ldots && \rightsquigarrow j_2 \\
=\ &\texttt{let}\ w_1 := (\texttt{let}\ w_2 := \checkmark(\texttt{let}\ w_3 := \texttt{truePrefix}\ []_{i_5}\ \texttt{in}\ \mathtt{t}_{i_6} \curvearrowleft \mathtt{t}_{i_6} :: \underline{w_3})\ \texttt{in}\ \ldots)\ \texttt{in}\ \ldots \\
& && \rightsquigarrow j_1 \\
=\ &\texttt{let}\ w_1 := (\texttt{let}\ w_2 := (\texttt{let}\ w_3 := []\ \texttt{in}\ \mathtt{t}_{i_6} \curvearrowleft \mathtt{t}_{i_6} :: \underline{w_3})\ \texttt{in}\ \ldots)\ \texttt{in}\ \ldots && \rightsquigarrow i_5 \\
=\ &\texttt{let}\ w_1 := (\texttt{let}\ w_2 := (\mathtt{t}_{i_6} \curvearrowleft \mathtt{t}_{i_6} :: []_{k_1@(1,s(i_5))})\ \texttt{in}\ \mathtt{f}_{i_3} \curvearrowleft \mathtt{f}_{i_3} :: \underline{w_2})\ \texttt{in}\ \ldots \\
=\ &\texttt{let}\ w_1 := (\texttt{let}\ w_2 := \mathtt{t}_{i_6} :: []_{k_1}\ \texttt{in}\ \mathtt{f}_{i_3} \curvearrowleft \mathtt{f}_{i_3} :: \underline{w_2})\ \texttt{in}\ \mathtt{t}_{i_4} \curvearrowleft \mathtt{t}_{i_4} :: \underline{w_1} && \rightsquigarrow i_6 \\
=\ &\texttt{let}\ w_1 := \mathtt{f}_{i_3} \curvearrowleft \mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{k_1}|\!\}_{k_2@a_{k_2}}\ \texttt{in}\ \mathtt{t}_{i_4} \curvearrowleft \mathtt{t}_{i_4} :: \underline{w_1} \\
& \quad \text{where } a_{k_2} = (1, s(j_1) \cup s(i_6)) = (1, \{j_1\} \cup s(i_1) \cup s(i_8) \cup s(i_6)) \\
=\ &\texttt{let}\ w_1 := []\ \texttt{in}\ \mathtt{t}_{i_4} \curvearrowleft \mathtt{t}_{i_4} :: \underline{w_1} && \rightsquigarrow i_3 \\
=\ &\mathtt{t}_{i_4} \curvearrowleft \mathtt{t}_{i_4} :: []_{k_3@(1,s(j_2)\cup s(i_3))} \\
=\ &\mathtt{t}_{i_4} :: []_{k_3} \qquad @(1, s(j_3) \cup s(i_4)) && \rightsquigarrow j_3, i_4
\end{aligned}
$$

(b) Evaluation steps of truePrefix

$$
\begin{aligned}
&\texttt{truePrefixAppend}\ \{\!|\mathtt{t}_{i_4} :: \{\!|\mathtt{f}_{i_3} :: []_{i_1}|\!\}_{i_2}|\!\}_{i_7}\ \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{i_8} \\
=\ &\checkmark\texttt{let}\ zs := \texttt{append}\ \{\!|\mathtt{t}_{i_4} :: \{\!|\mathtt{f}_{i_3} :: []_{i_1}|\!\}_{i_2}|\!\}_{i_7}\ \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{i_8}\ \texttt{in}\ \texttt{truePrefix}\ \underline{zs} \\
=\ &\texttt{truePrefix}\ \{\!|\mathtt{t}_{i_4} :: \{\!|\mathtt{f}_{i_3} :: \{\!|\mathtt{t}_{i_6} :: []_{i_5}|\!\}_{j_1}|\!\}_{j_2}|\!\}_{j_3@(1,s(i_7))} \\
=\ &\mathtt{t}_{i_4} :: []_{k_3} \qquad @(1 + 1, s(j_3) \cup s(i_4)) = (2, s(j_3) \cup s(i_4))
\end{aligned}
$$

(c) Evaluation steps of truePrefixAppend

Fig. 3: The Memorist evaluation for the program truePrefixAppend. Annotations are shown once and omitted subsequently. The final output values have their annotations shown next to themselves. We use the same notations and shorthands from Fig. 2. We also denote by $\underline{w_1}$ the thunking of w_1, by $s(i)$ the union of $\{i\}$ and the usage set annotated to i, and by $\rightsquigarrow i, j, \ldots$ that thunks named $i, j, \ldots$ are used in the last step proceeding to the expression on the current line.

during the computation of `append`. Thus, we ignore all the other thunks, and add only the cost annotation to j_2 to the output cost annotation. This gives us $1 + 1 = 2$ as the inferred lazy cost. Note that, while we have reasoned about different demand, we need not re-evaluate the entire expression, but only extract and aggregate information based on the demand from the same evaluated result.

Composing lazy cost analysis. A key feature of the Memorist Semantics is that we can analyse cost locally and compose cost analyses. Consider the evaluation of `truePrefix` and `truePrefixAppend`. Figure 3b illustrates the evaluation of `truePrefix` applied to the result of `append` (wrapped in a thunk named j_3). Its annotation comes from the output annotation of previous computations. The evaluation of `truePrefix` proceeds similarly as that of `append`, except it must also examine the list elements, and therefore, use the thunks wrapping the elements. For instance, computing $\texttt{true}_{i_6} \rightsquigarrow \texttt{true}_{i_6} :: []_{k_1}$ must access the value inside the thunk i_6 to determine which if-branch to take. Hence, the thunk k_2 wrapping this result must include $s(i_6)$ (among others) in its annotation. The similar thing happens with k_3 and the final output. Notice that, if a computation is never necessary, its cost and thunk usage will not be included anywhere in the final result. Indeed, the thunks k_1 (wrapping the empty list) and k_2 (wrapping the third element of the input list) are "thrown away" after encountering `false` and never needed to compute the final result in a lazy evaluation regardless of demand.

Now consider the top-level program `truePrefixAppend` in Fig. 3c. The Memorist evaluation of it essentially computes `append` on the two input lists, thunks the resulting list, and then computes `truePrefix`. We can collect thunks based on demand as before. If the output list is demanded only to its WHNF, we take only the thunks in the output annotation, which is $s(j_3) \cup s(i_4)$. But all except j_3 already exist in the input environment. The cost is thus the sum of the cost annotations given to the final output and to j_3, which is $2 + 1 = 3$. If the entire output list is demanded, we take also $s(i_4)$ and $s(k_3)$ into account, where the thunk i_4 thunks the first and only element and k_3 wraps the empty tail. With k_3, j_2 and j_3 being the thunks not existing in the input and each having a cost annotation of 1, we infer the total lazy cost to be $2 + 1 + 1 + 1 = 5$.

3 The Memorist Semantics

We consider the following typed total language $\mathcal{L}$ with Booleans, lists, explicit thunks, ticks, and structural recursion on lists. [3]

$$
\begin{array}{lll}
\text{Types} & A, B & ::= \mathsf{bool} \mid \mathsf{list}\, A \mid \mathsf{T}\, A \\
\text{Variables} & x, y & \in \text{Var} \\
\text{Expressions} & M, N & ::= x \mid \text{let}\, x = M\, \text{in}\, N \mid \text{true} \mid \text{false} \mid \text{if}\, M_1\, M_2\, M_3 \mid \text{tick}\, M \\
 & & \quad \mid \text{nil} \mid \text{cons}\, M\, N \mid \text{foldr}\, (\lambda xy.\, M_1)\, M_2\, M_3 \mid \text{lazy}\, M \mid \text{force}\, M
\end{array}
$$

[3] We have also handled pairs in our Rocq formalization; see the accompanying artefact for details. We omit them here for brevity.

Well-typed terms are given by judgements of the form $\gamma \vdash M : A$. For the typing rules, we refer the readers to [42] where the same language is considered. Thunks are represented with the T type and manipulated explicitly, allowing us to study laziness directly. The lazy construct thunks a computation, and force forces the evaluation of a thunk. The tick construct simulates a computation that incurs a unit cost. The construct foldr is included as a primitive to facilitate structural recursions on lists. We can view the language as an intermediate representation into which an ordinary functional language can be translated.

Memorist Semantics. We define the Memorist Semantics for the language $\mathcal{L}$ formally below. Terms are evaluated into values defined by

$$v ::= \text{true} \mid \text{false} \mid \text{nil} \mid \text{cons}\, v_1\, v_2 \mid \text{th}_i\, v$$

A value $\text{th}_i\, v$ of type $\mathsf{T}\, A$, where A is the type of v, represents a thunk and is assigned a unique name $i \in \mathbb{N}$. Following the idea in Section 2, the semantics associates the output value and each thunk with a pair (c, s) of a cost c and a usage set s of names of thunks. Components of an annotation can be extracted via the usual first and second projections on pairs, denoted $\pi_1(\cdot)$ and $\pi_2(\cdot)$ respectively.

An evaluation occurs in an evaluation environment defined as $\Gamma ::= \emptyset \mid \Gamma, x \mapsto v$ that maps variables to values. We also define an *annotation context*, $\mathcal{A} ::= \emptyset \mid \mathcal{A}, i \mapsto \alpha_i$, that maps a thunk name i to its annotation α_i, to track annotations. Keeping track of two separate contexts allows more flexibility in manipulating thunks and analysing their usage and cost.

Formally, the Memorist semantics for well-typed terms in $\mathcal{L}$ is defined by the evaluation judgement $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha \rangle$. It states that the well-typed term M evaluates, under the environment Γ and the annotation context $\mathcal{A}$, to the value v annotated by α with the extended annotation context $\mathcal{A}'$. The judgement is defined by the operational semantic rules in Fig. 4. Evaluation is eager by default, which is easier to reason about than its lazy counterpart. We claim (and prove in Section 4) that the recorded thunk usage captures exactly the thunks needed to be evaluated in a corresponding lazy evaluation, from which we can derive the lazy evaluation cost.

The EMBasic rule introduces the base cases. The EMVar rule looks up the variable in the environment for the value it binds to. The rules EMCons and EMLet proceed by evaluating the subterms sequentially. The annotations to values of the two sub-evaluations are combined by adding the costs and taking the union of the usage sets. With set union, the semantics can account for potential sharing of thunks. The rules EMIfTrue and EMIfFalse evaluates the if-condition and proceeds to evaluate the then- or else-branches accordingly. The rule EMTick increments the cost count by one.

In EMLazy, the term M is evaluated to a value v before being wrapped inside a th constructor. The computation cost and thunk usage associated with this evaluation is annotated to the thunked value v. Accordingly, the cost and usage associated with the final result $\text{th}_i\, v$ is empty, as a thunked computation incurs no cost and uses no additional thunks per se.

$$\textsc{EMVar}\ \frac{\Gamma(x) = v}{\Gamma; \mathcal{A} \vdash x \Downarrow \mathcal{A}; \langle v, \odot \rangle} \qquad \textsc{EMLet}\ \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}_1; \langle v_1, \alpha_1 \rangle \quad \Gamma, (x \mapsto v_1); \mathcal{A}_1 \vdash N \Downarrow \mathcal{A}_2; \langle v_2, \alpha_2 \rangle}{\Gamma; \mathcal{A} \vdash \text{let } x = M \text{ in } N \Downarrow \mathcal{A}_2; \langle v_2, \alpha_1 \oplus \alpha_2 \rangle}$$

$$\textsc{EMBasic}\ \frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{\Gamma; \mathcal{A} \vdash t \Downarrow \mathcal{A}; \langle t, \odot \rangle} \qquad \textsc{EMCons}\ \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}_1; \langle v_1, \alpha_1 \rangle \quad \Gamma; \mathcal{A}_1 \vdash N \Downarrow \mathcal{A}_2; \langle v_2, \alpha_2 \rangle}{\Gamma; \mathcal{A} \vdash \text{cons } M\ N \Downarrow \mathcal{A}_2; \langle \text{cons } v_1\ v_2, \alpha_1 \oplus \alpha_2 \rangle}$$

$$\textsc{EMIfTrue}\ \frac{\Gamma; \mathcal{A} \vdash M_1 \Downarrow \mathcal{A}_1; \langle \text{true}, \alpha_1 \rangle \quad \Gamma; \mathcal{A}_1 \vdash M_2 \Downarrow \mathcal{A}_2; \langle v, \alpha_2 \rangle}{\Gamma; \mathcal{A} \vdash \text{if } M_1\ M_2\ M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_1 \oplus \alpha_2 \rangle} \qquad \textsc{EMIfFalse}\ \frac{\Gamma; \mathcal{A} \vdash M_1 \Downarrow \mathcal{A}_1; \langle \text{false}, \alpha_1 \rangle \quad \Gamma; \mathcal{A}_1 \vdash M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_2 \rangle}{\Gamma; \mathcal{A} \vdash \text{if } M_1\ M_2\ M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_1 \oplus \alpha_2 \rangle}$$

$$\textsc{EMLazy}\ \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha \rangle \quad i \notin \text{dom}(\mathcal{A}')}{\Gamma; \mathcal{A} \vdash \text{lazy } M \Downarrow \mathcal{A}', (i \mapsto \alpha); \langle \text{th}_i\ v, \odot \rangle} \qquad \textsc{EMForce}\ \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle \text{th}_i\ v, \alpha \rangle \quad s = \alpha \oplus \{i\} \oplus \pi_2(\mathcal{A}'(i))}{\Gamma; \mathcal{A} \vdash \text{force } M \Downarrow \mathcal{A}'; \langle v, s \rangle}$$

$$\textsc{EMFoldrNil}\ \frac{\Gamma; \mathcal{A} \vdash M_3 \Downarrow \mathcal{A}_1; \langle \text{nil}, \alpha_1 \rangle \quad \Gamma; \mathcal{A}_1 \vdash M_2 \Downarrow \mathcal{A}_2; \langle v, \alpha_2 \rangle}{\Gamma; \mathcal{A} \vdash \text{foldr } (\lambda xy.M_1)\ M_2\ M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_1 \oplus \alpha_2 \rangle} \qquad \textsc{EMTick}\ \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha \rangle}{\Gamma; \mathcal{A} \vdash \text{tick } M \Downarrow \mathcal{A}'; \langle v, \alpha \oplus 1 \rangle}$$

$$\textsc{EMFoldrCons}\ \frac{\begin{array}{c}\Gamma; \mathcal{A} \vdash M_3 \Downarrow \mathcal{A}_1; \langle \text{cons } v\ (\text{th}_i\ vs), \alpha_1 \rangle \\ \Gamma; \mathcal{A}_1 \vdash \text{foldr } (\lambda xy.M_1)\ M_2\ vs \Downarrow \mathcal{A}_2; \langle v_2, \alpha_2 \rangle \quad j \notin \text{dom}(\mathcal{A}_2) \\ x' \neq y' \quad x', y' \notin \text{dom}(\Gamma) \cup \text{FV}(M_1) \cup \text{FV}(M_2) \cup \text{FV}(M_3) \quad s = \alpha_2 \oplus \{i\} \oplus \pi_2(\mathcal{A}_2(i)) \\ \Gamma, (x' \mapsto v), (y' \mapsto \text{th}_j\ v_2); \mathcal{A}_2, (j \mapsto s) \vdash M_1[x', y'/x, y] \Downarrow \mathcal{A}_3; \langle v_3, \alpha_3 \rangle\end{array}}{\Gamma; \mathcal{A} \vdash \text{foldr } (\lambda xy.M_1)\ M_2\ M_3 \Downarrow \mathcal{A}_3; \langle v_3, \alpha_1 \oplus \alpha_3 \rangle}$$

Fig. 4: The Memorist Semantics. The notation $\text{FV}(M)$ denotes the set of all free variables appearing in M. The empty annotation $(0, \emptyset)$ is denoted $\odot$. The operation $\oplus$ on annotations is defined as $\alpha_1 \oplus \alpha_2 := (\pi_1(\alpha_1) + \pi_1(\alpha_2),\ \pi_2(\alpha_1) \cup \pi_2(\alpha_2))$. We use the shorthand $\alpha \oplus n$ where n is a number to mean $(\pi_1(\alpha) + n,\ \pi_2(\alpha))$ and $\alpha \oplus s$ where s is a set to mean $(\pi_1(\alpha),\ \pi_2(\alpha) \cup s)$.

Thunked cost and usage are taken into effect only when the thunk is needed, forcing the computation to actually take place. Such forcing is explicitly done via the force construct, evaluated according to the EMForce rule. While thunks in the usage set annotated to the thunked value (i.e., $\pi_2(\mathcal{A}'(i))$ in the rule) are directly merged into the usage set annotated to the final value, the thunked cost is not added in yet. This avoids duplicating the cost when the same thunk is needed more than once. The total evaluation cost is to be derived afterwards.

The term foldr $(\lambda xy.\, M_1)\ M_2\ M_3$ is evaluated based on whether the list M_3 computes to is empty. If empty, M_2 is evaluated per the rule EMFoldrNil. Otherwise, the evaluation follows EMFoldrCons which recursively evaluates foldr on the tail of the list and then applies $\lambda xy.\, M_1$ to its head and the thunked

result of the recursive call.

Basic properties of annotation contexts. We are only interested in annotation contexts that are *valid* in the following sense. In the following theorem, we denote by $\mathrm{dom}(\cdot)$ and $\mathrm{im}(\cdot)$ the domain and the image of a function, respectively. By "thunks inside v", we mean thunks wrapping the arguments to the outermost and nested constructors of v: *e.g.,* all thunks appearing in the list $\mathrm{cons}\,(\mathrm{th}_i\,\mathrm{true})\,(\mathrm{th}_j\,(\mathrm{th}_k\,\mathrm{nil}))$.

Definition 1 (Valid annotation context). *An annotation context $\mathcal{A}$ is* valid *if for every annotation $\alpha \in \mathrm{im}(\mathcal{A})$, $\pi_2(\alpha) \subseteq \mathrm{dom}(\mathcal{A})$. We say such an $\mathcal{A}$ is* valid *for a value v if for any name i of a thunk inside v, we also have $i \in \mathrm{dom}(\mathcal{A})$. Moreover, $\mathcal{A}$ is* valid *for an environment Γ if $\forall x \in \mathrm{dom}(\Gamma)$, $\mathcal{A}$ is valid for $\Gamma(x)$.*

A Memorist evaluation may extend the annotation context but never modifies the existing annotations. It also preserves the validity of annotation contexts.

Lemma 1 (Evaluation only extends annotation context). *If $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha\rangle$ then $\forall i \in \mathrm{dom}(\mathcal{A}), \mathcal{A}'(i) = \mathcal{A}(i)$.*

Lemma 2 (Evaluation preserves annotation context validity). *If $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha\rangle$ and $\mathcal{A}$ is valid for Γ, then $\mathcal{A}'$ is valid for Γ and v, and $\pi_2(\alpha) \subseteq \mathrm{dom}(\mathcal{A}')$.*

Lazy cost and usage analysis. To derive the complete usage set and cost with respect to a demand, we must consider every thunk inferred to be needed. We have illustrated the basic idea in Section 2, which we express formally here. Consider an evaluation $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$. All thunks recorded in the usage set s annotated to v are needed to lazily evaluate M to v in WHNF. The total lazy cost is then the sum of all annotated costs given to every thunk in s, plus c. To perform such calculations, we define the following operation

$$\mathrm{sumcost}_M(\mathcal{A}, s) := \sum_{i \in s \cap \mathrm{dom}(\mathcal{A})} \pi_1(\mathcal{A}(i))$$

for an annotation context $\mathcal{A}$ and a usage set s. The total lazy cost of the above evaluation can be then expressed as $c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s)$. The set difference $\mathcal{A}' \setminus \mathcal{A}$ (where $\mathcal{A}$ and $\mathcal{A}'$ are treated as functions, i.e, sets of mappings) excludes thunks already evaluated prior to this evaluation. By so doing, we avoid including the cost incurred by previous computations.

Suppose the above evaluation yields $\mathrm{cons}\,(\mathrm{th}_i\,\mathrm{true})\,(\mathrm{th}_j\,\mathrm{nil})$, annotated with $(c, \{k, m\})$, and subsequent computations demand also the head of the list. We now need to collect also the thunk i and its usage set, *i.e.,* $\{k, m\} \cup \{i\} \cup \pi_2(\mathcal{A}'(i))$. Call this set s'. The cost is then $c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s')$. By collecting names in a set first, we ensure that no thunk can contribute to the cost more than once.

4 Correctness of the Memorist Semantics

We prove the Memorist Semantics is correct by relating it to the Clairvoyance Semantics [14, 26], which is equivalent to the standard call-by-need semantics [24]. The Clairvoyance Semantics nondeterministically chooses to evaluate or skip an expression when first encountered instead of delaying the evaluation until needed.

Reasoning directly about the relationship between the Memorist and the Clairvoyance Semantics is challenging. Instead, we define, as a stepping stone, a variant of the Clairvoyance Semantics where thunks are named and annotated with cost. We prove that this variant is equivalent to the original Clairvoyance Semantics and corresponds to the Memorist Semantics, therefore establishing a correspondence between the original Clairvoyance Semantics and the Memorist Semantics. The proofs presented here have been formalized in Rocq; see the accompanying artefact.

4.1 The Clairvoyance Semantics

Here we present a Clairvoyance Semantics for the language $\mathcal{L}$, adapted directly from the formalization by [42] which is itself based on a monadic variant of the Clairvoyance Semantics due to [26]. Types are interpreted as

$$\begin{aligned}
[\![A]\!] &: \mathrm{Set} \\
[\![\mathsf{bool}]\!] &:= \{\mathrm{true}, \mathrm{false}\} \\
[\![\mathsf{list}\ A]\!] &:= \{\mathrm{nil}\} \cup \{\mathrm{cons}\ \hat{v}_1\ \hat{v}_2 \mid \hat{v}_1 \in [\![\mathsf{T}\ A]\!], \hat{v}_2 \in [\![\mathsf{T}\ (\mathsf{list}\ A)]\!]\} \\
[\![\mathsf{T}\ A]\!] &:= \{\bot\} \cup \{\mathrm{th}\ \hat{v} \mid \hat{v} \in [\![A]\!]\}
\end{aligned}$$

The thunk type T is interpreted as a set whose elements are either of the form $\mathrm{th}\,\hat{v}$ representing a thunk evaluated to a value v, or $\bot$ representing skipped computation. The interpretation extends to the type context.

The semantics of evaluating a well-typed term $\gamma \vdash M : A$ is denoted by $[\![M]\!] : [\![\gamma]\!] \to \mathcal{P}([\![A]\!] \times \mathbb{N})$ that takes an interpreted environment $\hat{\Gamma} \in [\![\gamma]\!]$ and produces a set of pairs of values $\hat{v} \in [\![A]\!]$ and the cost $c \in \mathbb{N}$ incurred by the evaluation. The detailed definition of the semantics can be found in [26, 42]. When evaluating lazy M and foldr which involves creating thunks, the semantics nondeterministically chooses to evaluate a subterm to a value and wrap it in a th constructor, or to skip the evaluation and produce $\bot$. Forcing a thunk is simply accessing the evaluated value in the thunk. The evaluation of force N fails if the computation of N is skipped. The evaluation cost tracked and output by the semantics varies with the nondeterministic choices made during the evaluation. For a given term and environment, if the output is nonempty, the minimal cost for the same value corresponds to the call-by-need cost.

$$\textsc{EAVar}\ \frac{\tilde{\Gamma}(x) = \tilde{v}}{\tilde{\Gamma}; \mathcal{C} \vdash x \Downarrow^A \mathcal{C}; \langle \tilde{v}, 0 \rangle} \qquad \textsc{EALet}\ \frac{\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}_1; \langle \tilde{v}_1, c_1 \rangle \quad (\tilde{\Gamma}, x \mapsto \tilde{v}_1); \mathcal{C}_1 \vdash N \Downarrow^A \mathcal{C}_2; \langle \tilde{v}_2, c_2 \rangle}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{let}\, x = M \,\mathrm{in}\, N \Downarrow^A \mathcal{C}_2; \langle \tilde{v}_2, c_1 + c_2 \rangle}$$

$$\textsc{EALazy}\ \frac{\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle \quad i \notin \mathrm{dom}(\mathcal{C}')}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{lazy}\, M \Downarrow^A \mathcal{C}', (i \mapsto c); \langle \mathrm{th}_i\, \tilde{v}, 0 \rangle} \qquad \textsc{EALazySkip}\ \frac{}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{lazy}\, M \Downarrow^A \mathcal{C}; \langle \bot, 0 \rangle}$$

$$\textsc{EAForce}\ \frac{\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \mathrm{th}_i\, \tilde{v}, c \rangle}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{force}\, M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle} \qquad \textsc{EAFoldrNil}\ \frac{\tilde{\Gamma}; \mathcal{C} \vdash M_3 \Downarrow^A \mathcal{C}_1; \langle \mathrm{nil}, c_1 \rangle \quad \tilde{\Gamma}; \mathcal{C}_1 \vdash M_2 \Downarrow^A \mathcal{C}_2; \langle \tilde{v}, c_2 \rangle}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{foldr}\, (\lambda xy.\, M_1)\, M_2\, M_3 \Downarrow^A \mathcal{C}_2; \langle \tilde{v}, c_1 + c_2 \rangle}$$

$$\textsc{EAFoldrConsSkip}\ \frac{\begin{array}{c}\tilde{\Gamma}; \mathcal{C} \vdash M_3 \Downarrow^A \mathcal{C}_1; \langle \mathrm{cons}\, \tilde{v}\, \tilde{vs}, c_1 \rangle \\ x' \neq y' \quad x', y' \notin \mathrm{dom}(\tilde{\Gamma}) \cup \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \cup \mathrm{FV}(M_3) \\ \tilde{\Gamma}, (x' \mapsto \tilde{v}_1), (y' \mapsto \bot); \mathcal{C}_2 \vdash M_1[x', y'/x, y] \Downarrow^A \mathcal{C}_3; \langle \tilde{v}_2, c_3 \rangle\end{array}}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{foldr}\, (\lambda xy.\, M_1)\, M_2\, M_3 \Downarrow^A \mathcal{C}_3; \langle \tilde{v}_2, c_1 + c_3 \rangle}$$

$$\textsc{EAFoldrCons}\ \frac{\begin{array}{c}\tilde{\Gamma}; \mathcal{C} \vdash M_3 \Downarrow^A \mathcal{C}_1; \langle \mathrm{cons}\, \tilde{v}\, (\mathrm{th}_i\, \tilde{vs}), c_1 \rangle \quad \tilde{\Gamma}; \mathcal{C}_1 \vdash \mathrm{foldr}\, (\lambda xy.\, M_1)\, M_2\, \tilde{vs} \Downarrow^A \mathcal{C}_2; \langle \tilde{v}_2, c_2 \rangle \\ j \notin \mathrm{dom}(\mathcal{C}_2) \cup \{i\} \quad x' \neq y' \quad x', y' \notin \mathrm{dom}(\tilde{\Gamma}) \cup \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \cup \mathrm{FV}(M_3) \\ \tilde{\Gamma}, (x' \mapsto \tilde{v}), (y' \mapsto \mathrm{th}_i\, \tilde{v}_2); \mathcal{C}_2, (j \mapsto c_2) \vdash M_1[x', y'/x, y] \Downarrow^A \mathcal{C}_3; \langle \tilde{v}_3, c_3 \rangle\end{array}}{\tilde{\Gamma}; \mathcal{C} \vdash \mathrm{foldr}\, (\lambda xy.\, M_1)\, M_2\, M_3 \Downarrow^A \mathcal{C}_3; \langle \tilde{v}_3, c_1 + c_3 \rangle}$$

Fig. 5: Selected rules of the Annotated Clairvoyance Semantics. $\mathrm{FV}(M)$ denotes the set of free variables in M.

4.2 A Cost Annotated Variant of the Clairvoyance Semantics

We define a cost-annotated variant of the Clairvoyance Semantics, similar to the Memorist Semantics, except that there are no usage sets. We define values as

$$\tilde{v} ::= \mathrm{true} \mid \mathrm{false} \mid \mathrm{nil} \mid \mathrm{cons}\, \tilde{v}_1\, \tilde{v}_2 \mid \mathrm{th}_i\, \tilde{v} \mid \bot$$

Values of the thunk type T now take two forms: an evaluated thunk, $\mathrm{th}_i\, v$, with a unique name $i \in \mathbb{N}$, or a skipped computation $\bot$.

Evaluations occur in an environment $\tilde{\Gamma} ::= \emptyset \mid \tilde{\Gamma}, x \mapsto \tilde{v}$ and a cost annotation context $\mathcal{C} ::= \emptyset \mid \mathcal{C}, i \mapsto c_i$ that maps thunk names to their annotations. The semantics is defined by evaluation judgements of the form $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle$. It states that a well-typed term M is evaluated in an environment $\tilde{\Gamma}$ and a cost annotation context $\mathcal{C}$ to the value $\tilde{v}$ with a cost annotation c and a cost annotation context $\mathcal{C}'$. Most semantic rules are similar to those of the Memorist Semantics sans the tracking of thunk usage with sets. We show some of the rules in Fig. 5. The main difference resides in the evaluation of lazy and foldr, where thunks are created. Under Clairvoyance Semantics, the evaluation nondeterministically evaluates the term (EALazy and EAFoldrCons) or skips

it (EALazySkip and EAFoldrConsSkip). Forcing a term by force M only succeeds if M evaluates to an evaluated thunk, and fails otherwise.

Basic properties. An Annotated Clairvoyance evaluation may extend the cost annotation context but never modifies the existing annotations.

Lemma 3 (Evaluation only extends cost annotation context). *If* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle$*, then* $\forall i \in \mathrm{dom}(\mathcal{C}), \mathcal{C}'(i) = \mathcal{C}(i)$.

We analogously define validity of cost annotation contexts and show it is preserved by evaluation. We consider only such valid contexts in subsequent proofs.

Definition 2 (Valid cost annotation context). *A cost annotation context* $\mathcal{C}$ *is* valid *for a value* $\tilde{v}$*, if any name* i *assigned to a thunk nested inside* $\tilde{v}$ *is in* $\mathrm{dom}(\mathcal{C})$*.* $\mathcal{C}$ *is* valid *for an environment* $\tilde{\Gamma}$ *if* $\forall x \in \mathrm{dom}(\tilde{\Gamma})$*,* $\mathcal{C}$ *is valid for* $\tilde{\Gamma}(x)$*.*

Lemma 4 (Evaluation preserves cost annotation context validity). *If* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle$ *and* $\mathcal{C}$ *is valid for* $\tilde{\Gamma}$*, then* $\mathcal{C}'$ *is valid for* $\tilde{\Gamma}$ *and for* $\tilde{v}$*.*

Clairvoyance evaluation cost. The total evaluation cost can be recovered by adding up the cost annotations given to the thunks created during this evaluation, plus the cost annotation given to the output value. We define the following operation to sum over cost annotations from a cost annotation context $\mathcal{C}$:

$$\mathrm{sumcost}_A(\mathcal{C}) := \sum_{i \in \mathrm{dom}(\mathcal{C})} \mathcal{C}(i)$$

For an evaluation $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle$, the cost incurred by the evaluation can be computed by $c + \mathrm{sumcost}_A(\mathcal{C}' \setminus \mathcal{C})$.

Correspondence between the two Clairvoyance Semantics. We define a correspondence between values and between environments of the two semantics.

Definition 3 (Corresponding Clairvoyance values and environments). *Let* $\tilde{v}$ *and* $\hat{v}$ *be values of the Annotated and of the monadic Clairvoyance Semantics respectively. They are* corresponding values *modulo names, denoted* $\tilde{v} \sim \hat{v}$*, if the rules below apply. The relation extends to environments naturally.*

$$\frac{t \in \{\mathrm{true}, \mathrm{false}, \mathrm{nil}, \bot\}}{t \sim t} \qquad \frac{\tilde{v}_1 \sim \hat{v}_1 \quad \tilde{v}_2 \sim \hat{v}_2}{\mathrm{cons}\, \tilde{v}_1\, \tilde{v}_2 \sim \mathrm{cons}\, \hat{v}_1\, \hat{v}_2} \qquad \frac{\tilde{v} \sim \hat{v}}{\mathrm{th}_i\, \tilde{v} \sim \mathrm{th}\, \hat{v}}$$

If $\tilde{v} \sim \hat{v}$, then $\hat{v}$ can be regarded as abstracting away the name from $\tilde{v}$. Furthermore, a value $\hat{v}$ in the monadic Clairvoyance Semantics corresponds to an infinite set of values $\tilde{v}$ in the Annotated Semantics that are all structurally the same but with different names to thunks, if the values have thunks nested inside.

The theorems below establish the soundness and completeness together with cost equality. We consider the evaluation of a well-typed term $\gamma : M \vdash A$.

Theorem 1 (Soundness of the Annotated Clairvoyance Semantics wrt the monadic Clairvoyance Semantics). *Let* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c \rangle$*, with the cost annotation context* $\mathcal{C}$ *valid for* $\tilde{\Gamma}$*. Then for all* $\hat{\Gamma} \in [\![\gamma]\!]$ *with* $\tilde{\Gamma} \sim \hat{\Gamma}$*, there exists* $(\hat{v}, c') \in [\![M]\!](\hat{\Gamma})$*, such that* $\tilde{v} \sim \hat{v}$ *and* $c + \mathrm{sumcost}_A(\mathcal{C}' \setminus \mathcal{C}) = c'$*.*

Theorem 2 (Completeness of the Annotated Clairvoyance Semantics wrt the monadic Clairvoyance Semantics). *Let* $\hat{\Gamma} \in [\![\gamma]\!]$ *with* $\tilde{\Gamma} \sim \hat{\Gamma}$ *and let* $\mathcal{C}$ *be a cost annotation context valid for* $\tilde{\Gamma}$*. Then for all* $(\hat{v}, c) \in [\![M]\!](\hat{\Gamma})$*, there is an evaluation* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c' \rangle$ *with* $\tilde{v} \sim \hat{v}$ *and* $c' + \mathrm{sumcost}_A(\mathcal{C}' \setminus \mathcal{C}) = c$*.*

4.3 The Memorist and the Annotated Clairvoyance Semantics

Clairvoyance evaluation can make nondeterministic choices that are more eager than necessary. Thus in principle, to show that the Memorist Semantics corresponds to the lazy semantics, we need to *choose* the right nondeterministic branch of Clairvoyance evaluation for a given demand, which is difficult to do directly. Instead, we take an approach inspired by [42]. We prove that the cost inferred from the Memorist Semantics is no larger than any nondeterministic branch of a corresponding Clairvoyance evaluation, and that there is always a Clairvoyance branch producing the same cost.

However, there is still a problem. The Memorist Semantics provides a summary of all individual pieces of information at the end of evaluation. When a thunk is created, it is not known whether it would be eventually used in the Memorist evaluation, yet it is already evaluated or skipped in a corresponding Clairvoyance evaluation. Therefore, we cannot simply compare the evaluation cost step by step. To address this, we prove first that the thunk usage tracked by the Memorist Semantics correctly captures the lazy behaviour. We show that the inferred thunk usage is *minimal* in that it never gives more thunks than actually evaluated in a corresponding Clairvoyance evaluation, and it is *sufficient* in that there is some corresponding Clairvoyance evaluation evaluating exactly those thunks as inferred. From these we can derive the cost correctness.

Corresponding thunks in Memorist and Annotated Clairvoyance Semantics may have different names. Thus, we define renaming functions of type $\mathcal{N}_A \to \mathcal{N}_M$ mapping Annotated Clairvoyance thunk names $\mathcal{N}_A \subset \mathbb{N}$ to Memorist thunk names $\mathcal{N}_M \subset \mathbb{N}$. Since the Memorist evaluation is always eager, it never evaluates less than the corresponding Clairvoyance evaluation. Hence, thunk renaming functions are always total. We define a notion of validity for renaming functions to ensure that the names in the domains and images are indeed names assigned in the respective evaluation. We also require renaming functions to be injective so that two names, if related under such a function, are uniquely related.

Definition 4 (Validity of renaming functions). *A renaming function* $f : \mathcal{N}_A \to \mathcal{N}_M$ *is* valid *with respect to some valid annotation context* $\mathcal{A}$ *and valid cost annotation context* $\mathcal{C}$ *if* f *is injective,* $\mathrm{dom}(f) = \mathrm{dom}(\mathcal{C})$ *and* $\mathrm{im}(f) \subseteq \mathrm{dom}(\mathcal{A})$*.*

4.4 Functional and Cost Correctness of Memorist Semantics

We define a relation $\tilde{v} \sim_f v$ on an annotated Clairvoyance value $\tilde{v}$ and a Memorist value v, parametrized by a renaming function $f : \mathcal{N}_A \to \mathcal{N}_M$.

Definition 5 (Value-name and environment-name correspondence). *Given a renaming function f as above, the value-name correspondence $\tilde{v} \sim_f v$ is defined inductively by the rules below. The environment-name correspondence $\tilde{\Gamma} \sim_f \Gamma$ for an Annotated Clairvoyance environment $\tilde{\Gamma}$ and a Memorist environment Γ holds if* $\mathrm{dom}(\tilde{\Gamma}) = \mathrm{dom}(\Gamma)$ *and* $\forall x \in \mathrm{dom}(\Gamma)$, $\tilde{\Gamma}(x) \sim_f \Gamma(x)$.

$$\frac{t \in \{\mathrm{true}, \mathrm{false}, \mathrm{nil}\}}{t \sim_f t} \qquad \frac{\tilde{v}_1 \sim_f v_1 \quad \tilde{v}_2 \sim_f v_2}{\mathrm{cons}\, \tilde{v}_1\, \tilde{v}_2 \sim_f \mathrm{cons}\, v_1\, v_2} \qquad \frac{\tilde{v} \sim_f v \quad f(i) = j}{\mathrm{th}_i\, \tilde{v} \sim_f \mathrm{th}_j\, v} \qquad \frac{i \in \mathcal{N}_M}{\bot \sim_f \mathrm{th}_i\, v}$$

The relation $\sim_f$ describes the partial correspondence on values/environments and names between the two semantics. Given two environments related by $\sim_f$ where f is valid, a Memorist evaluation and an Annotated Clairvoyance evaluation of the same term produce $\sim_f$-related values, and the extended renaming function remains valid. Validity of the renaming functions ensure that each evaluated thunk in the Clairvoyance evaluation corresponds to a unique thunk from the Memorist evaluation. We also define a thunkwise cost correspondence:

Definition 6 (Thunkwise cost correspodence). *Let f be a valid thunk renaming function with respect to an annotation context $\mathcal{A}$ and a cost annotation context $\mathcal{C}$. $\mathcal{A}$ is* thunkwise cost corresponded *with $\mathcal{C}$ under f, denoted $\mathcal{A} \dot{\sim}_f \mathcal{C}$, if* $\forall i \in \mathrm{dom}(\mathcal{C})$, $\mathcal{C}(i) = \pi_1(\mathcal{A}(f(i)))$.

The theorem below states that, for each thunk evaluated in a Clairvoyance evaluation, there is a corresponding thunk in the Memorist evaluation with equal cost annotation. Intuitively, the cost annotation to a thunk is not associated with evaluating other thunks, and thus must be the same across the two semantics.

Theorem 3 (Functional and cost annotation correctness). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ and $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$, and f be a valid renaming function wrt $\mathcal{A}$ and $\mathcal{C}$. If $\tilde{\Gamma} \sim_f \Gamma$ and $\mathcal{A} \dot{\sim}_f \mathcal{C}$, then $\tilde{v} \sim_{f'} v$, $c = c'$ and $\mathcal{C}' \dot{\sim}_{f'} \mathcal{A}'$, for some f' extending f and valid wrt $\mathcal{A}'$ and $\mathcal{C}'$.*

Deriving usage from demand. Lazy evaluation is driven by demand which cannot always be determined locally. In Memorist Semantics, we consider such demand via usage sets. As discussed previously, if more than the outermost portion of a value is demanded, we collect all the demanded thunks in the value, along with thunks in the usage sets annotated to them and to the final output.

Since we do not generally know the exact names given to thunks from the outset, we need a way to abstract away from names when representing demand. For the current proof, the partially evaluated values in the monadic Clairvoyance Semantics in Section 4.1 provides a convenient way. We refer to them as *thunk-nameless partial values* below, and relate them with Memorist values as follows.

Definition 7 (Partial evaluatedness). *A thunk-nameless partial value $\hat{v}$ is a partially evaluated version of a Memorist value v of the same type, denoted $\hat{v} \preccurlyeq v$, if the rules below apply. The relation extends naturally to environments.*

$$\frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{t \preccurlyeq t} \qquad \frac{\hat{v}_1 \preccurlyeq v_1 \quad \hat{v}_2 \preccurlyeq v_2}{\text{cons}\, \hat{v}_1\, \hat{v}_2 \preccurlyeq \text{cons}\, v_1\, v_2} \qquad \frac{\hat{v} \preccurlyeq v}{\text{th}\, \hat{v} \preccurlyeq \text{th}_i\, v} \qquad \frac{}{\bot \preccurlyeq \text{th}_i\, v}$$

We use $\hat{v}$ with $\hat{v} \preccurlyeq v$ to represent a demand on a Memorist value v. The following relation characterizes the set of names of all thunks needed given such a demand.

Definition 8 (Usage representation sets). *Given values v and $\hat{v}$ with $\hat{v} \preccurlyeq v$ and a valid annotation context $\mathcal{A}$. $\text{Repr}_{\mathcal{A}}(v, \hat{v}, s)$ if the rules below apply.*

$$\frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{\text{Repr}_{\mathcal{A}}(t, t, \emptyset)} \qquad \frac{\text{Repr}_{\mathcal{A}}(v_1, \hat{v}_1, s_1) \quad \text{Repr}_{\mathcal{A}}(v_2, \hat{v}_2, s_2)}{\text{Repr}_{\mathcal{A}}(\text{cons}\, v_1\, v_2, \text{cons}\, \hat{v}_1\, \hat{v}_2, s_1 \cup s_2)}$$

$$\frac{}{\text{Repr}_{\mathcal{A}}(\text{th}_i\, v, \bot, \emptyset)} \qquad \frac{\text{Repr}_{\mathcal{A}}(v, \hat{v}, s)}{\text{Repr}_{\mathcal{A}}(\text{th}_i\, v, \text{th}\, \hat{v}, \{i\} \cup s \cup \pi_2(\mathcal{A}(i)))}$$

The relation extends to environments: given a Memorist environment Γ and a thunk-nameless partial environment $\hat{\Gamma}$ with $\hat{\Gamma} \preccurlyeq \Gamma$, $\text{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, s')$ if $s' = \bigcup_{x \in \text{dom}(\Gamma)} \{i \in s_0 \mid \text{Repr}_{\mathcal{A}}(\Gamma(x), \hat{\Gamma}(x), s_0)\}$. Let (c, s_1) be the annotation given to v. We call the set $s \cup s_1$ the usage representation set *for the demand $\hat{v}$ on v.*

Lemma 5. *Let v and $\hat{v}$ be a Memorist value and a thunk-nameless partial value $\hat{v}$ respectively, of the same type. Let $\mathcal{A}$ be a valid annotation context for v. We have $\hat{v} \preccurlyeq v$ if and only if there is a set s of thunk names with $\text{Repr}_{\mathcal{A}}(v, \hat{v}, s)$. The same holds for environments in both directions.*

Usage minimality. We now prove that the usage representation set inferred from the Memorist Semantics is minimal. Specifically, we show that every name in such a usage representation set always corresponds to the name of an evaluated thunk in a corresponding successful Clairvoyance evaluation after renaming.

We can immediately make one observation. If the above minimality holds, then for every Memorist thunk j corresponding to an evaluated Clairvoyance thunk under a thunk renaming function f, its usage set annotation must be included in the image of f, $\text{im}(f)$, that captures all Memorist thunks corresponding to some evaluated Clairvoyance thunks. This is desired; otherwise we would have discovered a Memorist thunk inferred to be needed whereas a successful Clairvoyance evaluation did not evaluate the thunk, failing the minimality.

Definition 9 (Thunkwise usage minimal). *An annotation context $\mathcal{A}$ is* thunkwise usage minimal *with respect to a cost annotation context $\mathcal{C}$ under a valid thunk renaming function f, denoted $\mathcal{A} \Subset_f \mathcal{C}$, if $\forall i \in \text{dom}(\mathcal{C})$, we have $\pi_2(\mathcal{A}(f(i))) \subseteq \text{im}(f)$.*

The validity of f ensures that $i \in \text{dom}(f)$ and $f(i) \in \text{dom}(\mathcal{A})$. The following lemma states that any two corresponding evaluations of some well-typed term M preserve the thunkwise usage minimality.

Lemma 6 (Thunkwise usage minimality). *Let* $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ *and* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$. *If* $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f \mathcal{C}$ *and* $\mathcal{A} \Subset_f \mathcal{C}$ *for some valid thunk renaming function* f, *then* $\mathcal{A}' \Subset_{f'} \mathcal{C}'$ *and* $s \subseteq \mathrm{im}(f')$ *for a valid* f' *extending* f.

From here we can establish the desired usage minimality, generalizing to the usage representation set with respect to an arbitrary demand on the output value. The notation $f[\cdot]$ denotes the image of a set under some function f.

Theorem 4 (Usage minimality). *Let* $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ *and* $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$, *with* $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f \mathcal{C}$ *and* $\mathcal{A} \Subset_f \mathcal{C}$ *for some valid* f. *Given any thunk-nameless* $\hat{v}$ *with* $\tilde{v} \sim \hat{v}$ *and* $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$ *for some* s', *we have* $s \cup s' \subseteq f'[\mathrm{dom}(\mathcal{C}')]$ *and* $(s \cup s') \setminus \mathrm{dom}(\mathcal{A}) \subseteq f'[\mathrm{dom}(\mathcal{C}' \setminus \mathcal{C})]$ *for a valid* f' *extending* f.

The conclusion stated with set difference allows us to exclude thunks that already exist prior to this evaluation. In short, the theorem states that thunks captured by $(s \cup s') \setminus \mathrm{dom}(\mathcal{A})$, i.e., all thunks created during the evaluation and inferred to be needed by the Memorist Semantics, always correspond (under thunk renaming f') to a subset of all thunks evaluated on any successful branch in the corresponding Clairvoyance evaluation as captured by $\mathrm{dom}(\mathcal{C}' \setminus \mathcal{C})$. That is, a corresponding Clairvoyance evaluation can never evaluate fewer thunks.

Usage sufficiency. Usage minimality essentially states that the inferred usage is "not too big". Next, we prove it is "not too small" either: for any Memorist evaluation and some demand, there is a corresponding Annotated Clairvoyance evaluation evaluating exactly what is inferred to be needed by the former.

In general, we cannot assert there is always a successful Clairvoyance evaluation of a well-typed term given an arbitrary environment $\tilde{\Gamma}$: *e.g.,* evaluating $\mathrm{force}\, x$ in an environment $\tilde{\Gamma}$ with $\tilde{\Gamma}(x) = \bot$ can never succeed. To address this, we utilize the usage representation set to characterize a minimal environment. We have shown that every thunk in this set must be evaluated in all corresponding successful Clairvoyance evaluation. It remains a question whether it might be "too minimal" and miss some thunks that should have been evaluated, though it does not matter for now since we only rely on it to set a lower bound on how less evaluated the Clairvoyance environment can be.

Given the union of a usage representation set and the output usage annotation set, if we take the intersection of this union and the domain of the initial annotation context $\mathcal{A}$, the resulting set s should contain all existing thunks inferred to be needed by the evaluation. From s we can construct a partial environment $\hat{\Gamma}$ satisfying $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, s)$ to use as the "minimal" environment. We then consider any Clairvoyance environment no less evaluated than $\hat{\Gamma}$.

Definition 10 (No less evaluated values and environments). *An Annotated Clairvoyance* $\tilde{v}$ *is no less evaluated than a thunk-nameless partial value* $\hat{v}$ *of the same type, denoted* $\hat{v} \lesssim \tilde{v}$, *if the rules below apply.*

$$\frac{t \in \{\mathrm{true}, \mathrm{false}, \mathrm{nil}\}}{t \lesssim t} \qquad \frac{\hat{v}_1 \lesssim \tilde{v}_1 \quad \hat{v}_2 \lesssim \tilde{v}_2}{\mathrm{cons}\, \hat{v}_1\, \hat{v}_2 \lesssim \mathrm{cons}\, \tilde{v}_1\, \tilde{v}_2} \qquad \frac{\hat{v} \lesssim \tilde{v}}{\mathrm{th}\, \hat{v} \lesssim \mathrm{th}_i\, \tilde{v}} \qquad \frac{}{\bot \lesssim \mathrm{th}_i\, \tilde{v}}$$

The relation extends to environments naturally: let $\tilde{\Gamma}$ and $\hat{\Gamma}$ be an Annotated Clairvoyance environment and a thunk-nameless environment respectively, defined on the same typing context. $\hat{\Gamma} \lesssim \tilde{\Gamma}$ if $\forall x \in \mathrm{dom}(\hat{\Gamma})$, $\hat{\Gamma}(x) \lesssim \tilde{\Gamma}(x)$.

Below we consider a well-typed term $\gamma \vdash M : A$.

Theorem 5 (Usage sufficiency). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ with $\mathcal{A}$ valid for Γ. Let $\hat{v} \in [\![A]\!]$ with $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$ for some s'. There is some $\hat{\Gamma} \in [\![\gamma]\!]$ with $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, (s \cup s') \cap \mathrm{dom}(\mathcal{A}))$, such that for all Annotated Clairvoyance environment $\tilde{\Gamma}$ with $\hat{\Gamma} \lesssim \tilde{\Gamma}$, a valid cost annotation context $\mathcal{C}$ for $\tilde{\Gamma}$ and a valid thunk renaming function f with $\tilde{\Gamma} \sim_f \Gamma$, if $f, \mathcal{A}$ and $\mathcal{C}$ satisfy the assumptions in Theorem 4, then there is some $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$ with $\hat{v} \lesssim \tilde{v}$ and $(s \cup s') \setminus \mathrm{dom}(\mathcal{A}) = f'[\mathrm{dom}(\mathcal{C}' \setminus \mathcal{C})]$, for a valid f' extending f.*

The theorem states that, given a Memorist evaluation with some demand, we can always construct a corresponding Clairvoyance evaluation, such that all the thunks in the Memorist evaluation inferred to be needed (as captured by $(s \cup s') \setminus \mathrm{dom}(\mathcal{A})$) coincide with all the thunks evaluated during the corresponding Clairvoyance evaluation (as captured by $\mathrm{dom}(\mathcal{C}' \setminus \mathcal{C})$), after thunk renaming. In other words, there is always a corresponding Clairvoyance evaluation that evaluates exactly as per the inferred usage from the Memorist evaluation.

Together, Theorems 4 and 5 imply that the inferred thunk usage from the Memorist Semantics is minimal but nontrivial. It represents exactly what is evaluated on the laziest branch in a corresponding Clairvoyance evaluation.

4.5 Lazy cost correspondence

In the Memorist Semantics, lazy evaluation cost is derived based on the inferred thunk usage and the annotated cost. We have shown that the cost annotations are correct (Theorem 3), and so is the inferred thunk usage (Theorems 4 and 5). Consequently, the derived cost coincides with the evaluation cost of the laziest possible Clairvoyance evaluation. Below consider a well-typed term $\gamma \vdash M : A$.

Theorem 6 (Cost minimality). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ and $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$, with $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f \mathcal{C}$ and $\mathcal{A} \Subset_f \mathcal{C}$ for some valid thunk renaming function f. Given any thunk-nameless value $\hat{v}$ with $\tilde{v} \sim \hat{v}$ and $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$ for some s', we have $c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s \cup s') \leq c' + \mathrm{sumcost}_A(\mathcal{C}' \setminus \mathcal{C})$.*

Theorem 7 (Cost existence). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ with $\mathcal{A}$ a valid annotation context for Γ. Let $\hat{v} \in [\![A]\!]$ with $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$ for some s'. There is some $\hat{\Gamma} \in [\![\gamma]\!]$ with $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, (s \cup s') \cap \mathrm{dom}(\mathcal{A}))$, such that for all Annotated Clairvoyance environment $\tilde{\Gamma}$ with $\hat{\Gamma} \lesssim \tilde{\Gamma}$, a valid cost annotation context $\mathcal{C}$ for $\tilde{\Gamma}$ and a valid thunk renaming function f with $\tilde{\Gamma} \sim_f \Gamma$, if $f, \mathcal{A}$ and $\mathcal{C}$ satisfy the assumptions in Theorem 4, then there is some $\tilde{\Gamma}; \mathcal{C} \vdash M \Downarrow^A \mathcal{C}'; \langle \tilde{v}, c'\rangle$ with $\hat{v} \lesssim \tilde{v}$, and $c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s \cup s') = c' + \mathrm{sumcost}_A(\mathcal{C}' \setminus \mathcal{C})$.*

The two theorems state that the derived lazy cost from the Memorist evaluation is no larger than the cost incurred by any corresponding Clairvoyance evaluation and equal to the cost by some corresponding Clairvoyance evaluation. Thus, the Memorist Semantics derives the correct lazy evaluation cost.

5 Implementation

We provide a proof of concept implementation of the Memorist Semantics in Rocq. Our implementation uses shallow embedding and encapsulates all the operations over thunks and annotations using a simple monadic interface. We model named thunks using the `Th` type below, and represent an annotation as a pair of a natural number (`nat`) and a finite set of natural numbers (`NatSet`).

```
Record Th (A : Type) : Type := MkTh {name: nat; val: A}.
Definition Annot : Type := nat * NatSet.
```

We implement the Memorist evaluation using a monad `M`, encapsulating the manipulations of thunks and the accumulation of cost:

```
Record Result (A : Type) : Type := MkRes {val: A; annot: Annot; cont: AC}.
Definition M (A : Type) : Type := AC -> Result A.
```

It represents the evaluation of an expression in an annotation context (`AC`), encoded using an association list, to a result (`Result`). The result is a record containing the value (`val`), the output annotation (`annot`), and the final annotation context (`cont`). The monad `M` is essentially a generalized state monad, also known as an *update monad* [1]. All operations over thunks and annotations can be encapsulated using a few combinators of the monad `M`, shown in Fig. 6. Apart from the standard `ret` and `bind`, we also define `lazy` to wrap the value in `M` with a thunk, `forcing` and `force` to unwrap thunks, and `tick` to increment cost. The set of combinators is the same as that of the Clairvoyance Monad [26].

To analyse a program, we can use the operational semantics in Fig. 4 as a recipe to translate a pure program to a monadic program that *reifies* the cost. Let's consider the example in Section 2. We need to encode `lists` first:

```
Inductive ListT (A : Type) : Type :=
  NilT : ListT A | ConsT : Th A -> Th (ListT A) -> ListT A.
```

We show the translations in Fig. 7. The structure of the translated program follows the original one closely. Whenever the value inside a thunk is accessed, the thunk needs to be unwrapped first via forcing (`$!`). We separate the translation of `append` into a top-level `appendM` and an auxiliary `appendM_` (similar for `truePrefix`) so they can pass Rocq's termination checker directly. We put a `tick` at the start of each call to increment cost by one, though not in the top-level `appendM` and `truePrefixM` as they are simply unwrapping the thunks around the arguments.

We are interested in the lazy cost with respect to the length of a list demanded; accordingly, we define the following functions to collect needed thunks.

```
Fixpoint usageL_ (ac : AC) {A} (xs : ListT A) (n : nat) : NatSet :=
  match xs, n with
  | ConsT (MkTh i x) (MkTh j xs'), S n' =>
      mapSu ac i ∪ mapSu ac j ∪ usageL_ ac xs' n'
  | _, _ => emptyset
  end.
Definition usageL {A} (r : Result (ListT A)) (n : nat) : NatSet :=
  usageL_ (cont r) (val r) n.
```

```
Definition ret {A} (x : A) : M A := fun ac => MkRes x (0,emptyset) ac.
Definition bind {A B} (m : M A) (f : A -> M B) : M B := fun ac =>
  let 'MkRes x a1 ac1 := m ac in let 'MkRes y a2 ac2 := (f x) ac1 in
  MkRes y (fst a1 + fst a2, snd a1 ∪ snd a2) ac2.
Notation "x >> y" := (bind x (fun _ => y)).

Definition lazy {A} (m : M A) : M (Th A) := fun ac =>
  let 'MkRes x a ac1 := m ac in
  MkRes (MkTh (nextIdx ac1) x) (0,emptyset) (ext ac1 a).
Notation "'lazyLetM' x := y 'in' z" := (bind (lazy y) (fun x => z)).

Definition collect (i : nat) : M unit := fun ac =>
  MkRes tt (0, mapSu ac i) ac.

Definition forcing {A B} (th : Th A) (f : A -> M B) : M B :=
  match th with MkTh i v => collect i >> f v end.
Notation "f $! x" := (forcing x f).
Definition force {A} (th : Th A) : M A := forcing th ret.

Definition tick : M unit := fun ac => MkRes tt (1,emptyset) ac.
```

Fig. 6: Definitions of the monadic combinators for `M`, omitting level and associativity declarations for notations. The expression `nextIdx ac` returns a name not in the domain of the annotation context `ac`, and `ext ac a` extends `ac` by mapping the next new name to an annotation `a`. The expression `mapSu ac i` maps a thunk name `i` to a set containing `i` and the names in `i`'s usage set annotation under `ac`.

```
Fixpoint appendM_ {A}
  (xs : ListT A) (ys : Th (ListT A))
  : M (ListT A) := tick >>
  match xs with
  | NilT => force ys
  | ConsT x xs' =>
      lazyLetM zs := forcing xs'
        (fun xs_ => appendM_ xs_ ys)
      in ret (ConsT x zs)
  end.

Definition appendM {A}
  (xs ys : Th (ListT A))
  : M (ListT A) :=
  (fun xs_ => appendM_ xs_ ys) $! xs
```

```
Fixpoint truePrefixM_ (xs: ListT bool)
  : M (ListT bool) := tick >>
  match xs with
  | NilT => ret NilT
  | ConsT y ys =>
      lazyLetM zs:= truePrefixM_ $! ys
      in forcing y (fun y_ =>
        if y_ then ret (ConsT y zs)
        else ret NilT)
  end.

Definition truePrefixM
  (xs : Th (ListT bool))
  : M (ListT bool) :=
  truePrefixM_ $! xs.
```

```
Definition truePrefixAppendM (xs ys: Th (ListT bool)) : M (ListT bool) :=
  tick >> lazyLetM zs := appendM xs ys in truePrefixM zs.
```

Fig. 7: Translation of the functions `append`, `truePrefix` and `truePrefixAppend`.

Here `n` is the number of thunked cons cells demanded in the output list. The notation `mapSu ac i` maps the thunk name `i` to a set containing `i` and names of thunks in its usage set annotation. In general, different usage collection functions need to be defined for other datatypes and demands.

The following function captures the lazy cost derivation described in Section 3, where s is the usage representation set collected using *e.g.,* `usageL`. The expression `dom ac` gives the domain of the annotation context `ac` as a `NatSet`, and `sumcost` sums over the cost annotated to the thunks in the given set.

```
Definition infcost {A} (r : Result A) (s : NatSet) (ac : AC) : nat :=
  sumcost (cont r) (diff (snd (annot r) ∪ s) (dom ac)) + fst (annot r)
```

Now consider the example program, `truePrefixAppend [true;false] [true]`. We can obtain the concrete cost by running the translated program and applying `infcost` to the result. The input may be from some previous computation or from lifting some pure lists. Suppose we have

```
MkTh 4 (ConsT (MkTh 3 true) (MkTh 2 (ConsT (MkTh 1 false) (MkTh 0 NilT))))
MkTh 7 (ConsT (MkTh 6 true) (MkTh 5 NilT))
```

Below we refer to them as `tl1` and `tl2`, and the annotation context at this stage as `ac`. To infer cost, we apply `infcost` on the result and the usage representation set for some demand. As before, we consider two demands on the same output: one only to WHNF, *i.e.,* 0 cons cells, and the other demanding one more.

```
Compute let r := truePrefixAppendM tl1 tl2 ac in
        let cost0 := infcost r (usageL r 0) ac in
        let cost1 := infcost r (usageL r 1) ac in (cost0, cost1).
```

The above computes to $(3, 5)$, *i.e.,* the lazy cost is 3 with respect to the first demand and 5 with respect to the second. Tracking localized cost and usage information along computations enables the derivation of cost via program execution. This stands in contrast to Clairvoyance Monad where we cannot execute programs for cost but only to verify user-provided cost specifications. It also allows the Memorist Semantics to evaluate a program only once for analysing different demands in principle, as suggested by this presentation. In comparison, the Demand Semantics [42] must redo evaluation for each demand. We contemplate that such features may make the Memorist Semantics a suitable basis for developing a framework for testing, enabling more potential applications.

We can also prove specifications of lazy cost for these functions. For now, we focus on mechanically verifying user-provided specifications in Rocq. The first theorem below states that the lazy cost of `appendM` is linear in the length demanded from the output list and at most the size of the first input list, where size is measured using `sizeT` that counts the number of constructors in a list. The second states that the lazy cost of `truePrefixM` is linear in the length demanded from the output. The `ACOk` and `NameOk` premises ensures the existing thunk names and annotations are valid in the sense of Definition 1.

```
Theorem appendM_cost :
  forall ac {A} (txs tys : Th (ListT A)) n (r := appendM txs tys ac),
```

```
  ACOk ac -> NameOk txs ac -> NameOk tys ac -> n < sizeT (val r) ->
  infcost r (usageL r n) ac = min (n + 1) (sizeT (Th.val txs)).

Theorem truePrefixM_cost : forall ac txs n (r := truePrefixM txs ac),
  ACOk ac -> NameOk txs ac -> n < sizeT (val r) ->
  infcost r (usageL r n) ac = n + 1.
```

Directly using these theorems, we can prove the following that expresses the lazy cost of `truePrefixAppendM` in terms of that of `appendM` and `truePrefixAppendM`.

```
Theorem truePrefixAppendM_cost :
  forall ac txs tys n d (r := truePrefixAppendM txs tys ac),
  ACOk ac -> NameOk txs ac -> NameOk tys ac -> n < sizeT (val r) ->
  infcost r (usageL r n) ac = min (n+1) (sizeT (Th.val txs)) + (n+1) + 1.
```

Our experience with proving them suggests that more proof engineering improvements can be made in reasoning about usage sets. Nonetheless, our semantics offers the benefit of a simpler and more straightforward process, without the need of verifying two kinds of specifications using additional logic as in Clairvoyance Monad or dual translations for each function as in Demand Semantics.

6 Limitations

We choose to define and mechanize the Memorist Semantics based on the same source language as the Demand Semantics [42], because we are interested in defining a translation from a lazy functional language to Rocq in shallow embeddings. For this reason, we also inherit the same limitations of not covering first-class functions and general recursions.

First-class functions. The current mechanization does not account for first-class functions. This has the benefit of simplicity, since we need not handle closures, but excludes useful constructs like higher-order functions.

To show that the Memorist Semantics works for first-class functions, we additionally developed a pen-and-paper formalization of a heap-based Memorist Semantics on an untyped λ-calculus similar to the approach of Launchbury [24] and Hackett and Hutton [14]. We demonstrate that the Memorist Semantics does support first-class functions in this version. We provide the pen-and-paper formalization and a proof of correctness in the supplementary material.

We leave the mechanization of first-class functions to future work. Due to differences in the languages and the styles of the semantics, we anticipate challenges such as relating variables encoded as de Bruijn indices between the Memorist and Clairvoyance evaluations. The heap-based alternative version would require more explicit handling of index shifting. Moreover, Clairvoyance evaluations may skip let-bindings, potentially resulting in the same variable in the Clairvoyance and Memorist evaluations having different indices. Thus, we need to maintain another relations on indices intertwined with the shifting during proofs. In comparison, it is simpler to maintain thunk-renaming functions in

the current mechanization, since Clairvoyance evaluations may only skip thunks in lazy or foldr but not let-bindings, and the thunk names are separate from variables and unchanged during evaluations.

General recursion. Focusing on total programs with structural recursion allows us to keep the semantics simple and shallow-embeddable in Rocq, whose specification language is total, but limits the semantics due to no support of general recursion. In practice, one potential way to simulate general recursion is to limit the number of steps with fuel. When analysing cost, we consider only programs that halt eventually when evaluated lazily, for which there must be fuel that is sufficient. Thus, it may be possible to extend the semantics to handle recursive lets by introducing fuel, allowing us to analyse for infinite data structures and general recursive programs if they compute within some fuel.

7 Related Work

The standard call-by-need semantics. Launchbury's Natural Semantics [24], the standard call-by-need operational semantics, captures demand-drivenness and sharing by storing expressions unevaluated in mutual heaps, evaluating them when needed, and modifying the heaps to memoize the results. However, due to its statefulness, the semantics is difficult to reason about.

Clairvoyance Semantics. The Clairvoyance Semantics [14] instead interprets call-by-need as call-by-value with nondeterministic choices of proceeding or skipping a computation. To adapt it for formally verifying lazy evaluation cost, the Clairvoyance Monad [26] uses a monad to model the nondeterminism and encapsulate cost accumulation, with an option over a *thunk* datatype representing nondeterministic choices. It has a simple interface and avoids the exponential explosion in cost analysis of the underlying semantics. However, it is non-executable, as the embedded programs lead to Rocq propositions as proof obligations. Moreover, an additional logic similar to Incorrectness Logic [33] has to be introduced to reason about nondeterminism. One must provide and prove two kinds of specifications: one for all nondeterministic evaluation branches that succeed in computation, and another for the existence of a more specific branch. In proofs, one often needs to select the correct nondeterministic branch manually. In comparison, the Memorist Semantics is deterministic and executable, and tracks all relevant information alongside the computation, although usage sets can be potentially challenging for formal reasoning in Rocq.

Demand Semantics. The Demand Semantics [42], adapted from [2] to a total and typed setting, is another approach to reason about demand and cost for lazily evaluated programs. It considers two kinds of demand: the externally given demand on the output, and the input demand describing how evaluated the input needs to be per the output demand. The semantics infers the minimum input demand from the output demand, by having a forward evaluation for obtaining

fully evaluated output and a backward evaluation to infer the input demand from the output and the demand on it. Like our work, this semantics is deterministic, avoiding the complication of dealing with nondeterminism. However, it essentially has two separate semantics and requires translating a source program into two different versions, which results in code duplication and can be more error-prone. Moreover, the backward inference must always take a specific output demand as input. For the same output from a forward function with a different output demand, the semantics must redo the evaluation. In contrast, the Memorist Semantics tracks all necessary information during evaluation, which does not vary with output demand. Hence, analysis for the same function requires no re-evaluation even when different output demands are specified.

Formally verifying lazy evaluation cost. The monadic framework by [15] analyses computation cost of pure Haskell programs in Liquid Haskell [40], a proof assistant based on extending Haskell with refinement types. Nonstrictness is built into the relevant datatypes in the framework, while users must handle sharing explicitly by inserting a `pay` construct into code. This approach follows an earlier Agda library by [8] for verifying time cost bounds for lazy functional data structures using amortized analysis techniques from [34], and uses a monadic dependent *thunk* type with information about cost embedded at the type level.

Some recent efforts also focus on formally reasoning about the amortized cost of lazy functional data structures. The Iris$^{\$}$ framework [29] is used in [35] to verify the banker's queue, the physicist's queue, and the implicit queue in Rocq, via directly reasoning about mutable cells using the Iris separation logic [39]. The Demand Semantics is applied to formally reason about the amortized cost and persistence of the banker's queue and the implicit queue [42]. Proving the amortized cost of simple stacks, binomial heaps, and a variant of finger trees [6] has been done in Liquid Haskell, though in a non-lazy setting without sharing [4].

Analysing resource bounds for call-by-need programs. A strictness analysis based method by [37] mechanically analyses time cost bounds for call-by-need programs and generalizes to higher-order functions by introducing additional structures containing information about function applications and associated cost. A type-based analysis by [22] automatically infers linear cost bounds for call-by-need programs using Automatic Amortized Resource Analysis (AARA) introduced by [19] and employing the notion of prepaying to avoid duplicating shared cost, and extends to univariate polynomial bounds [30] based on the method by [18].

Resource analysis for strict languages. AARA initially deals with linearly-bounded heap space cost of first-order strict functional programs [19], and has been extended (*e.g.,* [18, 21, 17]), implemented in Resource Aware ML [16], and developed into a framework for certified automatic inference of resource bounds for low-level programs [5]. Another line of work analyses time complexity by extracting from programs the recurrence relations for cost [11, 10, 23, 7, 9].

Some recent efforts focus on unifying the analysis of call-by-value and call-by-name languages, though call-by-need is often not supported due to difficul-

ties with modelling laziness. Along this line, the λ-amor framework [36] is a time complexity analysis framework based on amortization and affine types, which monadically simulates call-by-value with call-by-name. The dependently typed logical framework **calf** [32, 31] unifies the two via a call-by-push-value language [25], and has been extended to handle other computational effects by incorporating inequational reasoning into the framework's type theory [12].

On the verification side, the TiML language [41] has built-in support for verifying time complexity bounds of programs using its dependent and refinement types inspired type system, with users providing complexity bound specifications in type annotations. The Rocq library by [28] verifies time complexity by annotating information about cost in a monadic type. Formalizing the big-O notation and extending the Separation Logic, the framework by [13] verifies worst-case time complexity bounds for higher-order imperative programs in Rocq.

8 Conclusion and Future Work

We have presented the Memorist Semantics, a novel cost semantics for call-by-need evaluation that tracks thunk usage and computation cost in a deterministic manner. Our approach improves prior approaches by enabling cost analysis that evaluates independently of demand context, avoids code duplication, and provides fine-grained cost attribution to individual components of a term. Crucially, this semantics enables cost analysis of lazy programs that preserves the deterministic and compositional call-by-value evaluation structure, making it an intuitive practical foundation for further cost-aware program reasoning.

We have presented a proof of concept implementation in Rocq and verified cost of some functions. Apart from addressing the aforementioned limitations, we also plan to apply it on larger case studies, including verifying amortized cost of lazy functional data structures [34]. We would like to optimize the framework for easier use in practice, conduct quantitative experiments and benchmark against related frameworks. By formalizing our model in Rocq, we provide a rigorous basis for developing cost analysis frameworks for real-world lazy functional languages such as Haskell, and pave the way for future work on verifying compiler optimizations, *e.g.,* dead code elimination. In the future, we plan to investigate these applications and the utilization of tools such as `hs-to-coq` [38, 3] to automatically translate Haskell code for analysis. Moreover, with an executable semantics, lazy cost can be inferred directly via program execution, and the same program only needs to be evaluated once for analysing various demands. Such features can potentially be helpful in property-based testing, another application area we plan to investigate in the future.

References

1. Ahman, D., Uustalu, T.: Update monads: Cointerpreting directed containers. In: Matthes, R., Schubert, A. (eds.) 19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse,

France. LIPIcs, vol. 26, pp. 1–23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013). https://doi.org/10.4230/LIPICS.TYPES.2013.1, https://doi.org/10.4230/LIPIcs.TYPES.2013.1

2. Bjerner, B., Holmström, S.: A composition approach to time analysis of first order lazy functional programs. In: Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89. pp. 157–165. FPCA '89, ACM Press (1989). https://doi.org/10.1145/99370.99382
3. Breitner, J., Spector-Zabusky, A., Li, Y., Rizkallah, C., Wiegley, J., Cohen, J.M., Weirich, S.: Ready, set, verify! applying hs-to-coqm to real-world haskell code. J. Funct. Program. **31**, e5 (2021). https://doi.org/10.1017/S0956796820000283, https://doi.org/10.1017/S0956796820000283
4. van Brügge, J.: Liquid amortization: Proving amortized complexity with liquidhaskell (functional pearl). In: Vazou, N., Morris, J.G. (eds.) Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024. pp. 97–108. ACM (2024). https://doi.org/10.1145/3677999.3678282, https://doi.org/10.1145/3677999.3678282
5. Carbonneaux, Q., Hoffmann, J., Reps, T.W., Shao, Z.: Automated resource analysis with coq proof objects. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 64–85. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_4, https://doi.org/10.1007/978-3-319-63390-9_4
6. Claessen, K.: Finger trees explained anew, and slightly simplified (functional pearl). In: Schrijvers, T. (ed.) Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020. pp. 31–38. ACM (2020). https://doi.org/10.1145/3406088.3409026, https://doi.org/10.1145/3406088.3409026
7. Cutler, J.W., Licata, D.R., Danner, N.: Denotational recurrence extraction for amortized analysis. Proc. ACM Program. Lang. **4**(ICFP), 97:1–97:29 (2020). https://doi.org/10.1145/3408979, https://doi.org/10.1145/3408979
8. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 133–144. ACM (2008). https://doi.org/10.1145/1328438.1328457, https://doi.org/10.1145/1328438.1328457
9. Danner, N., Licata, D.R.: Denotational semantics as a foundation for cost recurrence extraction for functional languages. J. Funct. Program. **32**, e8 (2022). https://doi.org/10.1017/S095679682200003X, https://doi.org/10.1017/S095679682200003X
10. Danner, N., Licata, D.R., Ramyaa: Denotational cost semantics for functional languages with inductive types. In: Fisher, K., Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015. pp. 140–151. ACM (2015). https://doi.org/10.1145/2784731.2784749, https://doi.org/10.1145/2784731.2784749
11. Danner, N., Paykin, J., Royer, J.S.: A static cost analysis for a higher-order language. In: Might, M., Horn, D.V., Abel, A., Sheard, T. (eds.) Proceedings of the 7th Workshop on Programming languages meets pro-

gram verification, PLPV 2013, Rome, Italy, January 22, 2013. pp. 25–34. ACM (2013). https://doi.org/10.1145/2428116.2428123, https://doi.org/10.1145/2428116.2428123

12. Grodin, H., Niu, Y., Sterling, J., Harper, R.: Decalf: A directed, effectful cost-aware logical framework. Proc. ACM Program. Lang. **8**(POPL), 273–301 (2024). https://doi.org/10.1145/3632852, https://doi.org/10.1145/3632852
13. Guéneau, A., Charguéraud, A., Pottier, F.: A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In: Ahmed, A. (ed.) Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10801, pp. 533–560. Springer (2018). https://doi.org/10.1007/978-3-319-89884-1_19, https://doi.org/10.1007/978-3-319-89884-1_19
14. Hackett, J., Hutton, G.: Call-by-need is clairvoyant call-by-value. Proc. ACM Program. Lang. **3**(ICFP), 114:1–114:23 (2019). https://doi.org/10.1145/3341718, https://doi.org/10.1145/3341718
15. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in liquid haskell. Proc. ACM Program. Lang. **4**(POPL), 24:1–24:27 (2020). https://doi.org/10.1145/3371092, https://doi.org/10.1145/3371092
16. Hoffmann, J., Aehlig, K., Hofmann, M.: Resource aware ML. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 781–786. Springer (2012). https://doi.org/10.1007/978-3-642-31424-7_64, https://doi.org/10.1007/978-3-642-31424-7_64
17. Hoffmann, J., Das, A., Weng, S.: Towards automatic resource bound analysis for ocaml. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 359–373. ACM (2017). https://doi.org/10.1145/3009837.3009842, https://doi.org/10.1145/3009837.3009842
18. Hoffmann, J., Hofmann, M.: Amortized resource analysis with polynomial potential. In: Gordon, A.D. (ed.) Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6012, pp. 287–306. Springer (2010). https://doi.org/10.1007/978-3-642-11957-6_16, https://doi.org/10.1007/978-3-642-11957-6_16
19. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: Aiken, A., Morrisett, G. (eds.) Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003. pp. 185–197. ACM (2003). https://doi.org/10.1145/604131.604148, https://doi.org/10.1145/604131.604148
20. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989). https://doi.org/10.1093/COMJNL/32.2.98, https://doi.org/10.1093/comjnl/32.2.98
21. Jost, S., Hammond, K., Loidl, H., Hofmann, M.: Static determination of quantitative resource usage for higher-order programs. In: Hermenegildo, M.V., Pals-

berg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 223–236. ACM (2010). https://doi.org/10.1145/1706299.1706327, https://doi.org/10.1145/1706299.1706327

22. Jost, S., Vasconcelos, P.B., Florido, M., Hammond, K.: Type-based cost analysis for lazy functional languages. J. Autom. Reason. **59**(1), 87–120 (2017). https://doi.org/10.1007/S10817-016-9398-9, https://doi.org/10.1007/s10817-016-9398-9
23. Kavvos, G.A., Morehouse, E., Licata, D.R., Danner, N.: Recurrence extraction for functional programs through call-by-push-value. Proc. ACM Program. Lang. **4**(POPL), 15:1–15:31 (2020). https://doi.org/10.1145/3371083, https://doi.org/10.1145/3371083
24. Launchbury, J.: A natural semantics for lazy evaluation. In: Deusen, M.S.V., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993. pp. 144–154. ACM Press (1993). https://doi.org/10.1145/158511.158618, https://doi.org/10.1145/158511.158618
25. Levy, P.B.: Call-By-Push-Value: A Functional/Imperative Synthesis, Semantics Structures in Computation, vol. 2. Springer (2004)
26. Li, Y., Xia, L., Weirich, S.: Reasoning about the garden of forking paths. Proc. ACM Program. Lang. **5**(ICFP), 1–28 (2021). https://doi.org/10.1145/3473585, https://doi.org/10.1145/3473585
27. Coq development team: The Coq proof assistant (Sep 2024). https://doi.org/10.5281/zenodo.14542673, https://doi.org/10.5281/zenodo.14542673
28. McCarthy, J.A., Fetscher, B., New, M.S., Feltey, D., Findler, R.B.: A coq library for internal verification of running-times. Sci. Comput. Program. **164**, 49–65 (2018). https://doi.org/10.1016/J.SCICO.2017.05.001, https://doi.org/10.1016/j.scico.2017.05.001
29. Mével, G., Jourdan, J., Pottier, F.: Time credits and time receipts in Iris. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11423, pp. 3–29. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_1, https://doi.org/10.1007/978-3-030-17184-1_1
30. Moreira, S., Vasconcelos, P.B., Florido, M.: Resource analysis for lazy evaluation with polynomial potential. In: Chitil, O. (ed.) IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020. pp. 104–114. ACM (2020). https://doi.org/10.1145/3462172.3462196, https://doi.org/10.1145/3462172.3462196
31. Niu, Y., Harper, R.: A metalanguage for cost-aware denotational semantics. In: 38th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2023, Boston, MA, USA, June 26-29, 2023. pp. 1–14. IEEE (2023). https://doi.org/10.1109/LICS56636.2023.10175777, https://doi.org/10.1109/LICS56636.2023.10175777
32. Niu, Y., Sterling, J., Grodin, H., Harper, R.: A cost-aware logical framework. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498670, https://doi.org/10.1145/3498670

33. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL), 10:1–10:32 (2020). https://doi.org/10.1145/3371078, https://doi.org/10.1145/3371078
34. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
35. Pottier, F., Guéneau, A., Jourdan, J.H., Mével, G.: Thunks and debits in separation logic with time credits. Proc. ACM Program. Lang. **8**(POPL) (2024), https://hal.science/hal-04238691/file/main.pdf
36. Rajani, V., Gaboardi, M., Garg, D., Hoffmann, J.: A unifying type-theory for higher-order (amortized) cost analysis. Proc. ACM Program. Lang. **5**(POPL), 1–28 (2021). https://doi.org/10.1145/3434308, https://doi.org/10.1145/3434308
37. Sands, D.: Complexity analysis for a lazy higher-order language. In: Jones, N.D. (ed.) ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings. Lecture Notes in Computer Science, vol. 432, pp. 361–376. Springer (1990). https://doi.org/10.1007/3-540-52592-0_74, https://doi.org/10.1007/3-540-52592-0_74
38. Spector-Zabusky, A., Breitner, J., Rizkallah, C., Weirich, S.: Total Haskell is reasonable Coq. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 14–27. ACM (2018). https://doi.org/10.1145/3167092, https://doi.org/10.1145/3167092
39. Spies, S., Gäher, L., Tassarotti, J., Jung, R., Krebbers, R., Birkedal, L., Dreyer, D.: Later credits: resourceful reasoning for the later modality. Proc. ACM Program. Lang. **6**(ICFP), 283–311 (2022). https://doi.org/10.1145/3547631, https://doi.org/10.1145/3547631
40. Vazou, N.: Liquid Haskell: Haskell as a Theorem Prover. Ph.D. thesis, University of California, San Diego, USA (2016), http://www.escholarship.org/uc/item/8dm057ws
41. Wang, P., Wang, D., Chlipala, A.: Timl: a functional language for practical complexity analysis with invariants. Proc. ACM Program. Lang. **1**(OOPSLA), 79:1–79:26 (2017). https://doi.org/10.1145/3133903, https://doi.org/10.1145/3133903
42. Xia, L., Israel, L., Kramarz, M., Coltharp, N., Claessen, K., Weirich, S., Li, Y.: Story of your lazy function's life: A bidirectional demand semantics for mechanized cost analysis of lazy programs. Proc. ACM Program. Lang. **8**(ICFP), 30–63 (2024). https://doi.org/10.1145/3674626, https://doi.org/10.1145/3674626