The Memorist Tale: Once Annotated Lazy Program Cost is Fun

ANONYMOUS AUTHOR(S)

Lazy evaluation offers great flexibility in computing only what is necessary. However, reasoning about the cost of lazy programs is notoriously difficult because computation happens out of order and depends on future demands. Recently, researchers are looking into alternative semantics that models the cost of lazy evaluation that enables simple cost analysis without reasoning about states. However, existing approaches either require dealing with nondeterminism, or require a complex bidirectional semantics. In this paper, we introduce the *Memorist Semantics*, a novel semantics to analyse the cost of lazy programs by tracking the cost and dependency of computing each component. Our approach annotates each component of a term with fine-grained cost information and usage sets, resulting in a deterministic semantics that can be encapsulated with a simple monadic interface. We formalize our semantics in the Rocq Prover and verify its soundness with respect to the existing Clairvoyance Semantics.

Additional Key Words and Phrases: computation cost, lazy evaluation, operational semantics

ACM Reference Format:
Anonymous Author(s). 2025. The Memorist Tale: Once Annotated Lazy Program Cost is Fun. In *Proceedings of ESOP*. ACM, New York, NY, USA, 31 pages. https://doi.org/XX.XXXX/XXXXXXX

1 Introduction

A key feature of pure functional programming languages is referential transparency: pure functions yield identical outputs for identical inputs. This facilitates formally verifying functional correctness, as one can reason equationally about *what* the function does in *isolation*; without considering external state or any other external information.

However, since functional programs typically abstract away *how* programs evaluate, assessing the computational cost of a function presents challenges. In particular, how fast a function evaluates, does not only depend on the function definition but it is intricately tied to the evaluation strategy being used. Two common strategies used in functional programming languages are: *call-by-value* which is used in eager programming languages such as Standard ML, and *call-by-need* which is employed by lazy programming languages, such as Haskell.

The call-by-need evaluation strategy enables expressive and flexible programming by avoiding redundant or unnecessary computation. Unlike eager evaluation, lazy evaluation ensures that components of a term are evaluated at most once, and only if they are required. This property allows for elegant program structuring and efficiency improvements, but it comes at a cost: analysing the cost of lazy programs remains notoriously difficult.

Running Example. We consider the following simple example to explain the difference Let `truePrefix` be a function that takes a list of booleans as input and returns the prefix of the input list that only contains `true` elements. The function `truePrefixOfAppend`, that we use as a running example in this paper, takes two lists of booleans, `appends` them, and applies `truePrefix` on the

ESOP, ,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XX.XXXX/XXXXXXX

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

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

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

Fig. 1. The Gallina definitions of append, `truePrefix`, and `truePrefixOfAppend` placed in A-normal form.

appended list. Figure 1 presents an implementation of these functions in Gallina; the specification language of the Rocq Prover. Now consider applying this example to two concrete lists: `truePrefixOfAppend [true;false] [true]`.

We are interested in the cost in terms of *time* this program takes to evaluate. A straightforward way of calculating such a cost is by counting the number of function applications. While Gallina does not enforce a particular evaluation strategy, we can imagine that this code is a *shallow embedding* of a program in another language. For example, one can use `hs-to-coq` [Breitner et al. 2021; Spector-Zabusky et al. 2018] to automatically translate Haskell code to this form. The evaluation cost depends on the evaluation strategy used in the source language.

Eager vs. lazy evaluation. Under eager evaluation, the computation of `truePrefixOfAppend` first computes append of the two lists `[true;false]` and `[true]` in full, evoking 3 calls to append. Then the resulting value appending the two lists is passed to `truePrefix`. Every element is processed, thus evoking 4 calls to `truePrefix` (including one for the empty list). Overall, the program evaluates to `[true]` and incurs a cost of $3+4+1 = 8$, with one additional call to the top-level `truePrefixOfAppend`.

In comparison, lazy evaluation is demand-driven: computation is performed based on which parts of the output are demanded. Let's start by assuming that the demand is only the weak-head normal form (WHNF) of the list.

Figure 2 illustrates how the evaluation proceeds stepwise. In the case of lazy evaluation, we only need the result up to `cons true t`, with `t` pointing to a thunk that need not be evaluated. Only one step of `truePrefix`, and one step of append is needed. Adding an additional step due to the call to `truePrefixOfAppend` itself, the total computation cost adds up to 3. At this step, we have all we need for WHNF, so the evaluation stops at this step marked $(*)$.

However, evaluating to WHNF may not be the end of the story in every context—more of the list could be demanded in the future. For example, let's assume that another computation that happens later might demand the full list from `truePrefixOfAppend`. Fortunately, we have already computed such list `cons true t`, so we can resume from there instead of starting from the beginning—this is known as *sharing* in lazy evaluation. In this case, `t` must be evaluated as well; that is, we need the result to be fully reduced to `cons true nil`. One more step of `truePrefix` and of append is needed, making the computation cost 5 in total.

Even in this case, the cost is smaller than that incurred by eager evaluation, as computations not necessary for producing the final result are never performed. This includes, for instance, the expression `append [] [true]` that is bound to z_2 in the illustration.

Challenges. The example demonstrates several challenges in analysing the cost of lazy evaluation:

- *The cost of lazy evaluation is not local.* Unlike eager evaluation, the cost incurred by a function may not happen at the call site of the function—it might happen later inside

$$
\begin{array}{rl}
& \texttt{truePrefixOfAppend}\ (\texttt{true} :: \texttt{false} :: [])\ (\texttt{true} :: []) \\
= & \checkmark \texttt{let}\ zs := \mathsf{append}\ (\texttt{true} :: \texttt{false} :: [])\ (\texttt{true} :: [])\ \texttt{in}\ \texttt{truePrefix}\ zs \\
= & \texttt{let}\ zs := \checkmark(\texttt{let}\ z_1 := \mathsf{append}\ (\texttt{false} :: [])\ (\texttt{true} :: [])\ \texttt{in}\ \texttt{true} :: z_1)\ \texttt{in}\ \texttt{truePrefix}\ zs \\
= & \texttt{let}\ z_1 := \mathsf{append}\ (\texttt{false} :: [])\ (\texttt{true} :: [])\ \texttt{in}\ \texttt{truePrefix}\ (\texttt{true} :: z_1) \\
= & \texttt{let}\ z_1 := \ldots\ \texttt{in}\ \checkmark \texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{true} \looparrowright \texttt{true} :: w_1 \\
= & \texttt{let}\ z_1 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{true} :: w_1 \qquad (*) \\
= & \texttt{let}\ z_1 := \checkmark(\texttt{let}\ z_2 := \mathsf{append}\ []\ (\texttt{true} :: [])\ \texttt{in}\ \texttt{false} :: z_2)\ \texttt{in} \\
& \quad \texttt{let}\ w_1 := \texttt{truePrefix}\ z_1\ \texttt{in}\ \texttt{true} :: w_1 \\
= & \texttt{let}\ z_2 := \mathsf{append}\ []\ (\texttt{true} :: [])\ \texttt{in}\ \texttt{let}\ w_1 := \texttt{truePrefix}\ (\texttt{false} :: z_2)\ \texttt{in}\ \texttt{true} :: w_1 \\
= & \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := \checkmark(\texttt{let}\ w_2 := \texttt{truePrefix}\ z_2\ \texttt{in}\ \texttt{false} \looparrowright \texttt{false} :: w_2)\ \texttt{in}\ \texttt{true} :: w_1 \\
= & \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{let}\ w_1 := (\texttt{let}\ w_2 := \ldots\ \texttt{in}\ [])\ \texttt{in}\ \texttt{true} :: w_1 \\
= & \texttt{let}\ z_2 := \ldots\ \texttt{in}\ \texttt{true} :: []
\end{array}
$$

Fig. 2. A stepwise illustration of lazily evaluating the program `truePrefixOfAppend`. Here we choose to count function applications. A tick (✓) is placed immediately after unfolding a step of function application to indicate that a cost is incurred.

another function's application. The cost of a function can depend on future demand. This makes it challenging to analyse individual lazy functions in isolation.

- *Evaluation steps of different functions are interleaved.* As demonstrated in Figure 2, the evaluation steps of `truePrefix` and append are interleaved to evaluate every element in the result list of `truePrefixOfAppend`.
- *The evaluation is stateful.* This is especially important for modelling sharing, as the evaluation needs to remember what has been previously computed, and what has not.

Existing approaches. How can we deal with these challenges and analyse lazy computation cost then? We can treat any lazy programs as a stateful program and analyse them using program logics for reasoning about states, such as separation logic. Pottier et al. [2024] took this approach utilizing the Iris$^{\$}$ framework [Mével et al. 2019] and the Iris separation logic [Spies et al. 2022]. But what if we just reason about lazy functional programs using simple tools like *equational reasoning*? Danielsson [2008] proposed a simple approach based on a graded monad that tracks the computation cost The approach was later adopted by Handley et al. [2020] to be used in Liquid Haskell [Vazou 2016]. However, this approach requires the user to know future demand to correctly deal with sharing.

Instead of directly dealing with the stateful natural semantics of laziness [Launchbury 1993], another approach is to use an *alternative semantics* that is equivalent to lazy evaluation in terms of computational cost. The key idea is that it only matters *whether* a computation happens, not *when* the computation happens, when analyzing computational cost. This makes it possible to *localise* lazy computation cost, provided there is a way to know future demands. The Clairvoyance Semantics [Hackett and Hutton 2019] takes this approach and it simulates future demands via *nondeterminism*. Li et al. [2021] further show that this semantics can be encoded in a simple interface using the clairvoyance monad. However, nondeterminism makes both testing and formal reasoning challenging. Indeed, Li et al. [2021] proposed an additional logic similar to Incorrectness Logic [O'Hearn 2020] for this reason.

To avoid nondeterminism, Xia et al. [2024] proposed the Demand Semantics based on the prior work of Bjerner and Holmström [1989]. The Demand Semantics employs two evaluation directions: a *forward* evaluation that, given input, computes output as normal; and a *demand* evaluation that, given pure input and *an output demand*, calculates *the minimal input demand* and the corresponding computation cost. This approach was shown to be effective in formal reasoning, but the two evaluation directions mean that one function needs to be translated into two different functions for formal reasoning, making the translation error-prone and bringing in new challenges to formal reasoning. We defer a more detailed discussion contrasting these approaches to Section 6.

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 terms. The key idea of the Memorist Semantics is to run an eager evaluation, while giving every piece of data a unique name and "memorising" which other information was used for computing this data. The Memorist Semantics enables analysing computation cost *locally* in isolation without considering states, similar to the Clairvoyance Semantics and the Demand Semantics. In the meantime, the Memorist Semantics improves on the Clairvoyance Semantics in that it is *deterministic* and it improves on the Demand Semantics, in that it only employs one semantics and does not require code duplication.

To achieve this, we address two key challenges: The first challenge is *precise cost attribution* for different components of a term or value. Since lazy evaluation may only require parts of a term, we must ensure that unnecessary computation is excluded from cost calculations. Our approach achieves this by tracking the cost of computing each component of a value separately, annotating each component of a data structure with its cost. The second challenge is *accounting for shared computation.* Because lazy evaluation ensures that each component of a term is evaluated at most once, even if its value is used multiple times, our semantics must prevent duplicate cost accounting. We achieve this by tracking *usage sets* and using set union for bookkeeping, ensuring that each component's cost is included at most once, regardless of how often the value is reused.

Contributions. We make the following contributions:

- We introduce the Memorist Semantics, a deterministic cost semantics for call-by-need that tracks both the cost and usage of subexpressions (Section 3).
- A proof of the correctness of our semantics, as well as a proof that our cost model is sound and that it correctly predicts the execution cost of a program under call-by-need (Section 4).
- We show that the Memorist Semantics can be encoded using a simple monadic interface just like the Clairvoyance Semantics via a proof of concept implementation in Gallina (Section 5).
- A formalization of our results in the Rocq Prover (formerly Coq) [Coq development team 2024], ensuring rigorous correctness proofs.

We describe the intuition behind the Memorist Semantics in Section 2 and discuss related work in Section 6. We conclude the paper and discuss future work in Section 8.

2 The Memorist Approach

Imagine a person who has a remarkably retentive memory, whom we call a *Memorist.* When evaluating a program, the Memorist does all the computations eagerly; in the meantime, they are able to memorize, for each computation that would be thunked in a lazy evaluation, what other thunks (or more specifically, the evaluated results of which) it uses for its evaluation and the computation cost of this evaluation. At the end of evaluating the entire program, the Memorist learns all the thunks used by computing each part of the value, along with their individual computation cost. If someone or some program demands parts of the value, they can tell all the thunks that are actually used in the evaluation up to these parts. Now a corresponding lazy evaluation would also have to evaluate exactly these thunks, one can thus infer the lazy evaluation cost based on the information provided by the Memorist.

Thunks and annotations. To realise this approach, we wrap every piece of data inside a *thunk* that tracks the data's usage and cost during evaluations. Thunks here are intended to simulate thunks in lazy evaluation, but they are not encoded as suspended computations. Instead, each thunk in the Memorist Semantics has a unique name to distinguish it from others, and an *annotation*, which is a pair of cost component and a set of thunk names. The former tracks the cost incurred by evaluating the thunk to its WHNF, *without* the cost contributed by evaluating any other thunks. The latter represents all "used thunks", *i.e.,* thunks whose results are used in that evaluation. We refer to these sets of names of used thunks as *usage sets* in what follows. We annotate output values similarly, with the annotation corresponding to the cost and thunk usage incurred by evaluating the expression to its WHNF (given all needed thunks are evaluated).

For example, let's consider the program `truePrefixOfAppend [true;false] [true]` from a Memorist's perspective. To account for the demand of part of a list, we embed thunks into lists by putting both arguments to the constructor cons in thunks. We also thunk the entire lists when using them as function arguments, as function applications do not necessarily use their arguments. Representing a thunk with a name i and an annotation a by $x_{i@a}$, where x is the expression being thunked, the two input lists, after thunking, can be represented by

$$\left\{\!\!\left\{ \mathtt{true}_{i_4@a_{i_4}} :: \left\{\!\!\left\{ \mathtt{false}_{i_3@a_{i_3}} :: [\,]_{i_1@a_{i_1}} \right\}\!\!\right\}_{i_2@a_{i_2}} \right\}\!\!\right\}_{i_7@a_{i_7}} \quad \text{and} \quad \left\{\!\!\left\{ \mathtt{true}_{i_6@a_{i_6}} :: [\,]_{i_5@a_{i_5}} \right\}\!\!\right\}_{i_8@a_{i_8}}$$

for some unique names $i_1, \ldots, i_8$ and annotations $a_{i_1}, \ldots, a_{i_8}$, with $::$ denoting cons. We delay the formal definitions to Section 3.

The Memorist Semantics in action. Under the Memorist Semantics, we evaluate the expression `truePrefixOfAppend [true;false] [true]` eagerly and track thunk usage and cost in the meantime. We show the detailed evaluation steps in Fig. 3.

We first compute list appending (Fig. 3a). At each step, we need to "unthunk" the first argument of the function to access the list inside. For example, at the first step, we unthunk thunk i_7 wrapping the entire first argument. By doing so, we say that we *used* thunk i_7 at this step. Thunk i_7 is the only thunk we need to use at this step. We then recursively apply the function append to the tail of the first input list until we reach the empty list. When we reach the empty list, we need to unthunk both thunks i_1 and i_8, and return the second input list. After this step, we wrap the returned result (bound to z_2 in the figure) in a new thunk with a fresh name j_1.

The result thunked in j_1 is obtained from computing append $[\,]_{i_1}$ $\left\{\!\!\left\{ \mathtt{true}_{i_6} :: [\,]_{i_5} \right\}\!\!\right\}_{i_8}$, which incurs a cost of 1 due to one call to append and uses the thunks i_1 and i_8. Hence, its cost annotation is 1. For its usage set, we need to include the names i_1 and i_8, as the corresponding thunks are used. Additionally, these two thunks also have their own thunk usage, recorded in the usage sets in their annotations a_{i_1} and a_{i_8}. We denote the usage sets annotated to these two thunks by s_{i_1} and s_{i_8} below respectively. These usage set annotations suggest that, if i_1 and i_8 are to be used, all the thunks in s_{i_1} and s_{i_8} must be used. To avoid repeated look up of usage set annotations subsequently, we propagate the usage of i_1 and i_8 by adding all thees thunk names to the usage set of j_1. Below we denote the set of $\{i_1\} \cup s_{i_1}$ by $s(i_1)$ for short, and similar for other thunks. The usage set annotation to j_1 is then $s(i_1) \cup s(i_8)$.

The rest of the computation proceeds in a similar fashion. Note that when annotating each new thunk, we do not include the cost and thunk usage incurred by evaluating any thunks inside the values. For instance, the annotation to the thunk j_2 captures only the cost and thunk usage incurred *after* the thunk j_1 is created. By so doing, we effectively tracks the localised cost and usage to each thunk, so that the information recorded for a thunk k never includes the cost and usage from another thunk m unless the evaluated result of k is always needed to evaluate m. This will become helpful later when we analyse the lazy evaluation behaviour and cost. Lastly, the final output is also annotated in this way. We note that this final annotation represents exactly the thunk usage and cost due to evaluating `append [true;false] [true]` to its WHNF.

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

(a) Evaluation steps of append

$$
\begin{aligned}
& \mathtt{truePrefix}\ \{\!\{\mathtt{true}_{i_4} :: \{\!\{\mathtt{false}_{i_3} :: \{\!\{\mathtt{true}_{i_6} :: [\,]_{i_5}\}\!\}_{j_1}\}\!\}_{j_2}\}\!\}_{j_3} \\
= {} & \checkmark \mathtt{let}\ w_1 := \mathtt{truePrefix}\ \{\!\{\mathtt{false}_{i_3} :: \{\!\{\mathtt{true}_{i_6} :: [\,]_{i_5}\}\!\}_{j_1}\}\!\}_{j_2}\ \mathtt{in}\ \mathtt{true}_{i_4} \hookrightarrow \mathtt{true}_{i_4} :: \underline{w_1} && \rightsquigarrow j_3 \\
= {} & \mathtt{let}\ w_1 := \checkmark \left(\mathtt{let}\ w_2 := \mathtt{truePrefix}\ \{\!\{\mathtt{true}_{i_6} :: [\,]_{i_5}\}\!\}_{j_1}\ \mathtt{in}\ \mathtt{false}_{i_3} \hookrightarrow \mathtt{false}_{i_3} :: \underline{w_2}\right) \mathtt{in} \ldots && \rightsquigarrow j_2 \\
= {} & \mathtt{let}\ w_1 := \left(\mathtt{let}\ w_2 := \checkmark \left(\mathtt{let}\ w_3 := \mathtt{truePrefix}\ [\,]_{i_5}\ \mathtt{in}\ \mathtt{true}_{i_6} \hookrightarrow \mathtt{true}_{i_6} :: \underline{w_3}\right) \mathtt{in} \ldots\right) \mathtt{in} \ldots && \rightsquigarrow j_1 \\
= {} & \mathtt{let}\ w_1 := \left(\mathtt{let}\ w_2 := \left(\mathtt{let}\ w_3 := [\,]\ \mathtt{in}\ \mathtt{true}_{i_6} \hookrightarrow \mathtt{true}_{i_6} :: \underline{w_3}\right) \mathtt{in} \ldots\right) \mathtt{in} \ldots && \rightsquigarrow i_5 \\
= {} & \mathtt{let}\ w_1 := \left(\mathtt{let}\ w_2 := \left(\mathtt{true}_{i_6} \hookrightarrow \mathtt{true}_{i_6} :: [\,]_{k_1@(1, s(i_5))}\right) \mathtt{in}\ \mathtt{false}_{i_3} \hookrightarrow \mathtt{false}_{i_3} :: \underline{w_2}\right) \mathtt{in} \ldots \\
= {} & \mathtt{let}\ w_1 := \left(\mathtt{let}\ w_2 := \mathtt{true}_{i_6} :: [\,]_{k_1}\ \mathtt{in}\ \mathtt{false}_{i_3} \hookrightarrow \mathtt{false}_{i_3} :: \underline{w_2}\right) \mathtt{in}\ \mathtt{true}_{i_4} \hookrightarrow \mathtt{true}_{i_4} :: \underline{w_1} && \rightsquigarrow i_6 \\
= {} & \mathtt{let}\ w_1 := \mathtt{false}_{i_3} \hookrightarrow \mathtt{false}_{i_3} :: \{\!\{\mathtt{true}_{i_6} :: [\,]_{k_1}\}\!\}_{k_2@a_{k_2}}\ \mathtt{in}\ \mathtt{true}_{i_4} \hookrightarrow \mathtt{true}_{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)) \\
= {} & \mathtt{let}\ w_1 := [\,]\ \mathtt{in}\ \mathtt{true}_{i_4} \hookrightarrow \mathtt{true}_{i_4} :: \underline{w_1} && \rightsquigarrow i_3 \\
= {} & \mathtt{true}_{i_4} \hookrightarrow \mathtt{true}_{i_4} :: [\,]_{k_3@(1, s(j_2) \cup s(i_3))} \\
= {} & \mathtt{true}_{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

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

(c) Evaluation steps of truePrefixOfAppend

Fig. 3. A stepwise illustration of the Memorist evaluation for the program truePrefixOfAppend. A tick ($\checkmark$) is placed immediately after unfolding a step of function application to indicate that a cost is incurred. The notation $\underline{w_1}$ denotes the thunking of w_1, and $e \hookrightarrow xs$ is shorthand for if e then xs else $[\,]$. The notation $\rightsquigarrow i, j, \ldots$ denotes that thunks named $i, j, \ldots$ are used in the last step proceeding to the expression on the current line. Annotations to thunks already shown before are omitted in subsequent lines. The notation $s(i)$ denotes the union of $\{i\}$ and the usage set annotated to i. Annotations to the output are given at the final steps next to the output value.

Lazy computation cost. With the information in the output annotation and annotations, we can analyse the cost of append with respect to *any demand* on the output list. We can do so by collecting all the thunks used by a demand and aggregating their individual cost, as each thunk has already recorded the cost and thunk usage in its annotation.

For example, if only the first element in the output list of append is demanded, we only need to consider the thunks used to evaluate the output value to its outermost constructor, *i.e.,* $s(i_7)$ in the annotation, the thunk i_4 wrapping the first element, and all the thunks used by i_4. That is, we consider the set $s(i_4) \cup s(i_7)$. However, we are interested in the lazy cost of append itself, not in the previous computations leading to its input lists. Consequently, we do not want to account for the cost incurred by the evaluation to the values inside the thunks in the input. Instead, we remove all the thunks existing prior to the evaluation of `append [true;false] [true]`. In this case, we arrive at the empty set after such removal, suggesting there is no thunked cost we need to consider, Therefore, we only look at the cost annotation to the final output, which is 1 as shown in the last evaluation step of Fig. 3a. The inferred lazy cost is thus 1, as expected.

If the output is demanded to the first two elements instead, we need to additionally take the thunk j_2 wrapping the first cons cell and i_3 wrapping the second element in the output list into account. That is, we consider all thunks in the set $s(i_7) \cup s(i_4) \cup s(j_2) \cup s(i_3)$ with $s(j_2) = \{j_2\} \cup s(i_2)$. However, only j_2 in this set is created during the computation of append, which has a cost annotation of 1. Adding this to the output cost annotation gives us $1 + 1 = 2$, indicating that the lazy cost is 2 under the demand in question. We can analyse cost for other demand in the same manner. 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 advantage of the Memorist Semantics is that we can analyze cost locally and compose cost analysis. To see why, let's consider the evaluation of `truePrefix` and `truePrefixOfAppend`. Figure 3b illustrates the evaluation of `truePrefix` applied to the result of append. As before, we wrap the input list in a thunk with a fresh name j_3. The thunk j_3 also has its annotation which comes from some previous computations (for example, from thunking the result of append in computing `truePrefixOfAppend`). The evaluation of `truePrefix` proceeds in a similar manner as that of append.

There are two things worth noticing in these evaluation steps. First, each time after returning from a recursive call, the value of the head element in the input list needs to be examined. If the head element is `true`, we cons the head element to the recursive result. Otherwise, we simply return the empty list. Accessing the value in the list requires *using* the thunk wrapping that list element. For example, consider the thunk k_2 wraps the result of computing $\texttt{true}_{i_6} \looparrowright \texttt{true}_{i_6} :: []_{k_1}$ bound to w_2. Computing the latter requires accessing the value inside the thunk i_6 to determine which branch to take for the if-expression. The thunk i_6 is used, and the usage $s(i_6)$ is thus taken as part of the usage set annotation to k_2. A similar thing happens with k_3 and with the final output.

Second, if a part of a computation is eventually unnecessary, its cost and thunk usage will not be included anywhere in the final result. For example, thunks k_1 (wrapping the empty list) and k_2 (wrapping the third element of the input list) are 'thrown away' after encountering `false`, so none of the associated cost or thunk usage is included in any part of the final output. Indeed, nothing after the second element in the input list is needed to compute the final result in lazy evaluation.

Now consider the top-level program `truePrefixOfAppend` in Fig. 3c. Evaluating the program `truePrefixOfAppend [true;false] [true]` under the Memorist Semantics essentially first computes append on the two lists (thunked and annotated), thunks the resulting list, and then computes `truePrefix` on that thunked list.

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

Fig. 4. Language Syntax

$$
\textsc{TBool}\ \frac{b \in \{\mathsf{true}, \mathsf{false}\}}{\gamma \vdash b : \mathsf{bool}}
\qquad
\textsc{TNil}\ \frac{}{\gamma \vdash \mathsf{nil} : \mathsf{list}\ A}
\qquad
\textsc{TCons}\ \frac{\gamma \vdash M : \mathrm{T}\ A \qquad \gamma \vdash N : \mathrm{T}\ (\mathsf{list}\ A)}{\gamma \vdash \mathsf{cons}\ M\ N : \mathsf{list}\ A}
$$

$$
\textsc{TPair}\ \frac{\gamma \vdash M : A \qquad \gamma \vdash N : B}{\gamma \vdash (M, N) : A \times B}
\qquad
\textsc{TFst}\ \frac{\gamma \vdash M : A \times B}{\gamma \vdash \mathsf{fst}\ M : A}
\qquad
\textsc{TSnd}\ \frac{\gamma \vdash M : A \times B}{\gamma \vdash \mathsf{snd}\ M : B}
\qquad
\textsc{TVar}\ \frac{\gamma(x) = A}{\gamma \vdash x : A}
$$

$$
\textsc{TIf}\ \frac{\gamma \vdash M_1 : \mathsf{bool} \qquad \gamma \vdash M_2 : A \qquad \gamma \vdash M_3 : A}{\gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 : A}
\qquad
\textsc{TLet}\ \frac{\gamma \vdash M : A \qquad \gamma, x : A \vdash N : B}{\gamma \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : B}
$$

$$
\textsc{TFoldr}\ \frac{\gamma, x : \mathrm{T}\ A, y : \mathrm{T}\ B \vdash M_1 : B \qquad \gamma \vdash M_2 : B \qquad \gamma \vdash M_3 : \mathsf{list}\ A}{\gamma \vdash \mathsf{foldr}\ (\lambda xy.\ M_1)\ M_2\ M_3 : B}
$$

$$
\textsc{TTick}\ \frac{\gamma \vdash M : A}{\gamma \vdash \mathsf{tick}\ M : A}
\qquad
\textsc{TLazy}\ \frac{\gamma \vdash M : A}{\gamma \vdash \mathsf{lazy}\ M : \mathrm{T}\ A}
\qquad
\textsc{TForce}\ \frac{\gamma \vdash M : \mathrm{T}\ A}{\gamma \vdash \mathsf{force}\ M : A}
$$

Fig. 5. Typing Rules

We can then perform thunk collection based on demand as before. If the output list is demanded only to its first element, the set of thunks collected is $s(j_3) \cup s(i_4)$, among which only j_3 is not presented in the input environment of this program. The cost is thus the sum of the cost annotation given to the final output and to the thunk j_3, which is $2 + 1 = 3$. If the demand is to construct the entire list, we take also $s(k_3)$ into account, where k_3 is the thunk wrapping the empty tail. With k_3 and j_2 being the two thunks not presented in the input alongside j_3 and each having a cost annotation of 1, the cost is inferred to be $2 + 1 + 1 + 1 = 5$.

Notice that, regardless of the demand, the cost and thunk usage associated with computing the base case of append will not be considered in this instance, aligning with the fact that a lazy evaluation of the program will not process the rest of the first input list after the element `false`.

3 The Memorist Semantics

We now formally define the Memorist Semantics.

3.1 Language Syntax and Type System

We define the Memorist semantics for a typed total language $\mathcal{L}$ with booleans, lists, pairs, explicit thunks, ticks, and structural recursion on lists, similar to the language considered by Xia et al. [2024]. Its syntax and typing rules of the language are defined in Fig. 4 and 5, where well-typed terms are given by judgements of the form $\gamma \vdash M : A$. Thunks are represented explicitly with the T type. The constructs lazy and force manipulates thunks explicitly, with the former thunking a computation, and the latter forcing the evaluation of the thunk. The tick construct simulates a computation that incurs a unit cost. The construct foldr is included as a primitive of the language to facilitate structural recursions on lists. The language can be regarded as an intermediate representation into which an ordinary functional language can be translated. Having thunks as a syntactic construct enables representing and studying laziness directly.

3.2 Memorist Semantics

We define the Memorist semantics for the language $\mathcal{L}$ formally below. Under the Memorist Semantics, terms are evaluated into values, which are defined by

$$\text{Value} \quad v ::= \text{true} \mid \text{false} \mid \text{nil} \mid \text{cons}\ v_1\ v_2 \mid (v_1, v_2) \mid \text{thunk}_i\ v$$

with an additional typing rule

$$\frac{\gamma \vdash v : A}{\gamma \vdash \text{thunk}_i\ v : \mathrm{T}\ A}\ \textsc{TMThunk}$$

Thunks are represented by values of the thunk type. As previously described in Section 2, each thunk is assigned a unique name $i \in \mathbb{N}$ when created to distinguish it from other thunks.

The semantics associates the output value and each thunk with an annotation that records cost and thunk usage related information. Each annotation α is a pair (c, s) of a cost annotation c and a set s containing the names of thunks. The cost annotation c is the cost accumulated during the evaluation that is never thunked by the end of the evaluation. As already can be seen from the motivating example presented earlier, the cost annotation c is typically only part of the cost of evaluating an expression to a value. To derive the evaluation cost, we must at least consider c together with thunks whose names are in the set s. The set s in the annotation is a collection of names of all the thunks whose evaluation must be forced to arrive at the WHNF of the value in question. We call this set of a *usage set* of this value. Each component of an annotation can be extracted via the usual first and second projections on pairs, denoted by $\pi_1(\cdot)$ 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 a *annotation context*, $\mathcal{A} ::= \emptyset \mid \mathcal{A}, i \mapsto \alpha_i$, that maps a name i of a thunk to its annotation α_i, in order to track and record the annotations so that they can be referred to later. In other words, two separate contexts are kept track of in the evaluation. This 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 $\mathcal{A}'$. This judgement is defined by the operational semantic rules presented in Figure 6. The evaluation itself is performed eagerly by default, which is easier to reason about than its lazy counterpart. We claim (and prove in the next section) that the recorded thunk usage reflects lazy behaviours, in the sense that it captures exactly the thunks needed to be evaluated and no more in a corresponding lazy evaluation, from which we can derive the lazy evaluation cost.

The EMBasic rule introduces straightforwardly the base cases. The EMVar rule looks up the variable in the environment for the value it binds to. The rules EMCons, EMPair and EMLet

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

$$\textsc{EMIfTrue}\quad \frac{\Gamma; \mathcal{A} \vdash M_1 \Downarrow \mathcal{A}_1; \langle \text{true}, \alpha_1\rangle \qquad \Gamma; \mathcal{A}_1 \vdash M_2 \Downarrow \mathcal{A}_2; \langle v, \alpha_2\rangle}{\Gamma; \mathcal{A} \vdash \text{if}\ M_1\ \text{then}\ M_2\ \text{else}\ M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_1 \oplus \alpha_2\rangle} \qquad \textsc{EMFst}\quad \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}_1; \langle (v_1, v_2), \alpha_1\rangle}{\Gamma; \mathcal{A} \vdash \text{fst}\ M \Downarrow \mathcal{A}_1; \langle v_1, \alpha_1\rangle}$$

$$\textsc{EMIfFalse}\quad \frac{\Gamma; \mathcal{A} \vdash M_1 \Downarrow \mathcal{A}_1; \langle \text{false}, \alpha_1\rangle \qquad \Gamma; \mathcal{A}_1 \vdash M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_2\rangle}{\Gamma; \mathcal{A} \vdash \text{if}\ M_1\ \text{then}\ M_2\ \text{else}\ M_3 \Downarrow \mathcal{A}_2; \langle v, \alpha_1 \oplus \alpha_2\rangle} \qquad \textsc{EMSnd}\quad \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}_1; \langle (v_1, v_2), \alpha_1\rangle}{\Gamma; \mathcal{A} \vdash \text{snd}\ M \Downarrow \mathcal{A}_1; \langle v_2, \alpha_1\rangle}$$

$$\textsc{EMPair}\quad \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}_1; \langle v_1, \alpha_1\rangle \qquad \Gamma; \mathcal{A}_1 \vdash N \Downarrow \mathcal{A}_2; \langle v_2, \alpha_2\rangle}{\Gamma; \mathcal{A} \vdash (M, N) \Downarrow \mathcal{A}_2; \langle (v_1, v_2), \alpha_1 \oplus \alpha_2\rangle} \qquad \textsc{EMTick}\quad \frac{\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, \alpha\rangle}{\Gamma; \mathcal{A} \vdash \text{tick}\ M \Downarrow \mathcal{A}'; \langle v, \alpha \oplus 1\rangle}$$

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

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

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

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

Fig. 6. 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 evaluates to an annotation composed of the sum of the costs and the union of the sets of the two input annotations. That is, $\alpha_1 \oplus \alpha_2 := (\pi_1(\alpha_1) + \pi_1(\alpha_2),\ \pi_2(\alpha_1) \cup \pi_2(\alpha_2))$. 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 $(\pi_1(\alpha),\ \pi_2(\alpha) \cup s)$.

proceed by evaluating the subterms separately but consecutively. The annotations to values of the two sub-evaluations are combined by adding the cost and taking the union of the resulting usage sets. With set union, the semantics is able to account for potential sharing of thunks. The rules EMIfTrue and EMIfFalse follow the similar vein, though the then- and else-branches are evaluated only when the if-condition evaluates to true and false respectively. The rules for pair projections, EMFst and EMSnd, evaluate a term M to a value of pair and return the corresponding component. The rule EMTick increments the cost count by one.

In the EMLazy rule, the term M is evaluated to a value before being wrapped inside a thunk constructor. The computation cost and usage set annotation associated with this evaluation is annotated to the thunked value v that M evaluates to, by extending the annotation context with this annotation for the introduced thunk. Accordingly, the cost and usage set associated with evaluating the halted computation $\mathrm{thunk}_i\ v$ is empty, as a thunked computation incurs no cost and utilises no additional thunks per se. Its associated cost and usage set are taken into effect only when the thunk is needed, forcing the computation to actually take place.

Forcing of the evaluation of thunks is explicitly done via force in this language, which is 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 via set union, the thunked cost is not added in yet at this point. This avoids the duplication of cost when the same thunk is needed more than once in the evaluation. The total evaluation cost is to be derived afterwards, based on the output usage set and the annotations.

The term $\mathrm{foldr}\ (\lambda x\, y\,.\, M_1)\ M_2\ M_3$ is evaluated based on whether or not M_3 is the empty list. When the argument list is empty, the result is simply the value of M_2, as specified by the rule EMFoldrNil. Otherwise, the evaluation follows the rule EMFoldrCons, which first recursively evaluates foldr on the tail of the list, and then applies $\lambda x\, y.\, M_1$ to the head of the list and the result of the recursive call. Notice that the results of the recursive calls are thunked: the step function $\lambda x\, y\,.\, M_1$ does not always have to use any of its argument.

We are only interested in annotation contexts that are *valid* in the following sense. Below let $\mathrm{dom}(\cdot)$ and $\mathrm{im}(\cdot)$ denote the domain and the image of a function, respectively.

Definition 3.1 (Valid Memorist annotation contexts). An annotation context $\mathcal{A}$ is *valid* if for every annotation $(c, s) \in \mathrm{im}(\mathcal{A})$, we have $s \subseteq \mathrm{dom}(\mathcal{A})$. An annotation context $\mathcal{A}$ is *valid* for a value v if it is itself valid, and for any name i assigned to a thunk inside v, $i \in \mathrm{dom}(\mathcal{A})$. Moreover, $\mathcal{A}$ is *valid* for an environment Γ if $\forall x \in \mathrm{dom}(\Gamma)$, $\mathcal{A}$ is valid for $\Gamma(x)$.

By 'thunks inside v', we mean thunks wrapping the arguments to the (outermost and nested) constructors of v. These include, for instance, all four thunks appearing in the list

$$\mathrm{cons}\ (\mathrm{thunk}_i\ \mathrm{true})\ (\mathrm{thunk}_j\ (\mathrm{cons}\ (\mathrm{thunk}_k\ \mathrm{false})\ (\mathrm{thunk}_m\ \mathrm{nil}))).$$

Basic Properties of the Annotation Context. The Memorist semantics has the following properties. An evaluation may extend the annotation context, but it never modifies the existing annotations. Furthermore, validity of annotation contexts is preserved by evaluation.

Lemma 3.2 (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 3.3 (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*, we have that* $\mathcal{A}'$ *is valid for* Γ *and* v*, and that* $\pi_2(\alpha) \subseteq \mathrm{dom}(\mathcal{A}')$.

3.3 Lazy Cost and Usage Analysis using Annotations Derived from Memorist Evaluation

In the Memorist semantics, each annotation is uniquely associated with the outermost portion of the annotated value. Hence, to derive the complete usage set and cost due to evaluating the expression to this value, we must consider every thunk that is inferred to be needed. Consider an evaluation $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s) \rangle$. As (c, s) is the annotation given to the value v, all the thunks recorded in the usage set s are required in order to lazily evaluate to the outermost portion of v. The total lazy cost of the evaluation, i.e., the cost that would be incurred if the evaluation were call-by-need, is then the sum of all annotated costs given to every thunk in s, together with c.

To perform such calculations, we define the following operation

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

for an annotation context $\mathcal{A}$ and a usage set s. The total lazy cost of the above evaluation to the outermost portion of v (i.e., to the expression in WHNF) can be then expressed as

$$c + \text{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s).$$

We use set difference on the extended and initial annotation contexts ($\mathcal{A}'$ and $\mathcal{A}$ respectively, treated as functions, i.e, set of pairs of mappings) to exclude thunks already evaluated prior to the evaluation we are considering. Set difference is essential; otherwise, we would also add in any cost incurred by computations prior to the current one.

Sometimes more than the outermost portion of a value may be used. For example, suppose the above evaluation yields

$$\left\langle \text{cons (thunk}_i \text{ true) (thunk}_j \text{ nil)}, \quad (c, \{k, m\}) \right\rangle,$$

and that it occurs in a larger context where the value of the head of the list is eventually also used. In this case, the needed thunks are not simply the thunks k and m, but also the thunk i and any further thunks used by the evaluation to the value inside that thunk. In other words, the complete usage set with respect to the demand in question should be $s' = \{k\} \cup \{i\} \cup \pi_2((\mathcal{A}'(i)))$. The cost can then be computed accordingly by $c + \text{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s')$.

When the set s' reflects exactly what is needed to evaluate under a call-by-need strategy, the cost computed as above will result in the same cost as that produced by the corresponding call-by-need evaluation. In particular, collecting all the relevant names into a set before adding up the cost ensures that no thunk can contribute to the total cost more than once. This avoids duplicating the cost of a thunk even if it is used multiple times. It also explains why we collect the name of a forced thunk (along with names in its annotated usage set) into the usage set in the annotation given to the final value without adding in the thunked cost immediately in the evaluation rule EMForce: we will not duplicate the cost if the thunk is forced more than once in the evaluation.

4 Correctness of the Memorist Semantics

We prove that the Memorist Semantics is correct, by relating it to the Clairvoyance Semantics. Initially introduced by Hackett and Hutton [2019], the Clairvoyance Semantics models call-by-need evaluation with nondeterminism, and is shown in the same paper to be equivalent to the standard call-by-need semantics [Launchbury 1993]. The Clairvoyance Semantics nondeterministically chooses to proceed with or skip the evaluation of an expression when it is first encountered, instead of delaying the evaluation altogether until it is actually needed (if at all).

Reasoning about the relation between the Memorist semantics and the Clairvoyance Semantics directly is challenging. Instead, we define, as a stepping stone, a variant of the Clairvoyance Semantics in which thunks are named and have their own cost annotations. We prove that the annotated variant is equivalent to the original one, and that the Memorist semantics corresponds to this annotated variant. This correspondence takes evaluation cost into account; hence it shows the Memorist Semantics not only evaluates correctly but also infers correctly the lazy cost. All theorems and proofs presented here have been formalized in Rocq Prover, the code of which can be found in the accompanying artefact.

4.1 The Clairvoyance Semantics

Here we present a Clairvoyance Semantics to the language $\mathcal{L}$, adapted directly from the formalisation by Xia et al. [2024] which is itself based on a monadic variant of the Clairvoyance Semantics due to [Li et al. 2021]. Types are interpreted as

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

The thunk type T is interpreted as a set whose elements are either of the form thunk $\hat{v}$ representing a thunk evaluated to a value v, or $\bot$ representing skipped computation. The interpretation extends to the type context Γ, so that an element of the interpreted $[\![\Gamma]\!]$ is an environment of the evaluation.

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 numbers $c \in \mathbb{N}$ of the cost incurred by the evaluation. The detailed definition of the monadic semantics can be found in Li et al. [2021]; Xia et al. [2024]. The output is denoted as a set, as the evaluation is nondeterministic. When evaluating lazy M and foldr which involve creating thunks, the semantics nondeterministically chooses to evaluate a subterm to a value and wrap it in the thunk constructor to represent an evaluated thunk, or skips the evaluation by producing the placeholder $\bot$. As the nondeterministic choice is done immediately when a thunk is created, forcing a thunk via the term force N is then simply accessing the already evaluated value in the thunk, if thunk is evaluated at all when created. The evaluation of force N fails if the computation of N is skipped.

In addition to computing the values, the semantics also keeps track of cost, which is the number c in the output of the function $[\![\cdot]\!]$ on terms. This is the total evaluation cost, which varies along 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.

4.2 A Cost Annotated Variant of the Clairvoyance Semantics

We define a new variant of the Clairvoyance Semantics that is equivalent to the monadic Clairvoyance Semantics, and then establish the correspondence of the Memorist Semantics with the new variant. In this variant, we name thunks, and give and the final value cost annotations, similar to the Memorist Semantics except there are no usage sets. Values are defined as

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

with additional typing rules

$$\frac{\gamma \vdash \tilde{v} : A}{\gamma \vdash \mathrm{thunk}_i\ \tilde{v} : \mathrm{T}\ A}\ \textsc{TAThunk} \qquad \frac{}{\gamma \vdash \bot : \mathrm{T}\ A}\ \textsc{TABot}$$

where $i \in \mathbb{N}$ is a unique name assigned to a thunk. Similar to the monadic Clairvoyance Semantics but unlike the Memorist Semantics, values of the thunk type T can take two forms: either an evaluated thunk, $\mathrm{thunk}_i\ v$, or a skipped computation $\bot$.

Evaluations occur in an environment defined as $\tilde{\Gamma} ::= \emptyset \mid \tilde{\Gamma}, x \mapsto \tilde{v}$. We also define a cost annotation context $C ::= \emptyset \mid C, i \mapsto c_i$ that maps thunk names to the corresponding annotations.

$$\textsc{EABasic}\quad \frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{\tilde\Gamma; C \vdash t \Downarrow^A C; \langle t, 0\rangle}
\qquad
\textsc{EACons}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C_1; \langle \tilde v_1, c_1\rangle \qquad \tilde\Gamma; C_1 \vdash N \Downarrow^A C_2; \langle \tilde v_2, c_2\rangle}{\tilde\Gamma; C \vdash \text{cons}\ M\ N \Downarrow^A C_2; \langle \text{cons}\ \tilde v_1\ \tilde v_2, c_1 + c_2\rangle}$$

$$\textsc{EAIfTrue}\quad \frac{\tilde\Gamma; C \vdash M_1 \Downarrow^A C_1; \langle \text{true}, c_1\rangle \qquad \tilde\Gamma; C_1 \vdash M_2 \Downarrow^A C_2; \langle \tilde v, c_2\rangle}{\tilde\Gamma; C \vdash \text{if}\ M_1\ \text{then}\ M_2\ \text{else}\ M_3 \Downarrow^A C_2; \langle \tilde v, c_1 + c_2\rangle}
\qquad
\textsc{EAFst}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C_1; \langle (\tilde v_1, \tilde v_2), c\rangle}{\tilde\Gamma; C \vdash \text{fst}\ M \Downarrow^A C_1; \langle \tilde v_1, c\rangle}$$

$$\textsc{EAIfFalse}\quad \frac{\tilde\Gamma; C \vdash M_1 \Downarrow^A C_1; \langle \text{false}, c_1\rangle \qquad \tilde\Gamma; C_1 \vdash M_3 \Downarrow^A C_3; \langle \tilde v, c_2\rangle}{\tilde\Gamma; C \vdash \text{if}\ M_1\ \text{then}\ M_2\ \text{else}\ M_3 \Downarrow^A C_2; \langle \tilde v, c_1 + c_2\rangle}
\qquad
\textsc{EASnd}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C_1; \langle (\tilde v_1, \tilde v_2), c\rangle}{\tilde\Gamma; C \vdash \text{snd}\ M \Downarrow^A C_1; \langle \tilde v_2, c\rangle}$$

$$\textsc{EAForce}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C'; \langle \text{thunk}_i\ \tilde v, c\rangle}{\tilde\Gamma; C \vdash \text{force}\ M \Downarrow^A C'; \langle \tilde v, c\rangle}
\qquad
\textsc{EAPair}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C_1; \langle \tilde v_1, c_1\rangle \qquad \tilde\Gamma; C_1 \vdash N \Downarrow^A C_2; \langle \tilde v_2, c_2\rangle}{\tilde\Gamma; C \vdash (M, N) \Downarrow^A \mathcal{A}_2; \langle (\tilde v_1, \tilde v_2), c_1 + c_2\rangle}$$

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

$$\textsc{EAVar}\quad \frac{\tilde\Gamma(x) = \tilde v}{\tilde\Gamma; C \vdash x \Downarrow^A C; \langle \tilde v, 0\rangle}
\qquad
\textsc{EALet}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C_1; \langle \tilde v_1, c_1\rangle \qquad (\tilde\Gamma, x \mapsto \tilde v_1); C_1 \vdash N \Downarrow^A C_2; \langle \tilde v_2, c_2\rangle}{\tilde\Gamma; C \vdash \text{let}\ x = M\ \text{in}\ N \Downarrow^A C_2; \langle \tilde v_2, c_1 + c_2\rangle}$$

$$\textsc{EATick}\quad \frac{\tilde\Gamma; C \vdash M \Downarrow^A C'; \langle \tilde v, c\rangle}{\tilde\Gamma; C \vdash \text{tick}\ M \Downarrow^A C'; \langle \tilde v, c + 1\rangle}
\qquad
\textsc{EAFoldrNil}\quad \frac{\tilde\Gamma; C \vdash M_3 \Downarrow^A C_1; \langle \text{nil}, c_1\rangle \qquad \tilde\Gamma; C_1 \vdash M_2 \Downarrow^A C_2; \langle \tilde v, c_2\rangle}{\tilde\Gamma; C \vdash \text{foldr}\ (\lambda x\ y.\ M_1)\ M_2\ M_3 \Downarrow^A C_2; \langle \tilde v, c_1 + c_2\rangle}$$

$$\textsc{EAFoldrCons}\quad \frac{\begin{array}{c}\tilde\Gamma; C \vdash M_3 \Downarrow^A C_1; \langle \text{cons}\ \tilde v\ (\text{thunk}_i\ \tilde{vs}), c_1\rangle \qquad \tilde\Gamma; C_1 \vdash \text{foldr}\ (\lambda x\ y.\ M_1)\ M_2\ \tilde{vs} \Downarrow^A C_2; \langle \tilde v_2, c_2\rangle \\ j \notin \text{dom}(C_2) \cup \{i\} \qquad x' \neq y' \qquad x', y' \notin \text{dom}(\tilde\Gamma) \cup \text{FV}(M_1) \cup \text{FV}(M_2) \cup \text{FV}(M_3) \\ \tilde\Gamma, (x' \mapsto \tilde v), (y' \mapsto \text{thunk}_i\ \tilde v_2); C_2, (j \mapsto c_2) \vdash M_1[x', y'/x, y] \Downarrow^A C_3; \langle \tilde v_3, c_3\rangle\end{array}}{\tilde\Gamma; C \vdash \text{foldr}\ (\lambda x\ y.\ M_1)\ M_2\ M_3 \Downarrow^A C_3; \langle \tilde v_3, c_1 + c_3\rangle}$$

$$\textsc{EAFoldrConsSkip}\quad \frac{\begin{array}{c}\tilde\Gamma; C \vdash M_3 \Downarrow^A C_1; \langle \text{cons}\ \tilde v\ \tilde{vs}, c_1\rangle \qquad x' \neq y' \qquad x', y' \notin \text{dom}(\tilde\Gamma) \cup \text{FV}(M_1) \cup \text{FV}(M_2) \cup \text{FV}(M_3) \\ \tilde\Gamma, (x' \mapsto \tilde v_1), (y' \mapsto \bot); C_2 \vdash M_1[x', y'/x, y] \Downarrow^A C_3; \langle \tilde v_2, c_3\rangle\end{array}}{\tilde\Gamma; C \vdash \text{foldr}\ (\lambda x\ y.\ M_1)\ M_2\ M_3 \Downarrow^A C_3; \langle \tilde v_2, c_1 + c_3\rangle}$$

Fig. 7. The Annotated Clairvoyance Semantics. The set of free variables in M is denoted $\text{FV}(M)$.

The semantics is defined by evaluation judgements of the form $\tilde\Gamma; C \vdash M \Downarrow^A C'; \langle \tilde v, c\rangle$. It states that a well-typed term M is evaluated in an environment $\tilde\Gamma$ and a cost annotation context C to the value $\tilde v$ with a cost annotation c and with the cost annotation context C'. The cost annotations here are essentially the same thing as in the Memorist Semantics.

We show all the semantics rules in Fig. 7. Most rules are similar to the evaluation rules of the Memorist Semantics, sans the tracking of thunk usage with sets. The main difference resides in the evaluation of lazy and foldr, where thunks are created. Under Clairvoyance Semantics, the evaluation nondeterministically evaluates the term and wraps the result with a thunk (rules EALazy and EAFoldrCons), or skips the evaluation and outputs $\bot$ (rule EALazySkip and EAFoldrConsSkip). Likewise, the evaluation of a term force M only succeeds if M evaluates to an evaluated thunk, and fails if the evaluation gives $\bot$ that represents a skipped computation.

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

Lemma 4.1 (Evaluation Only Extends Cost Annotation Context). *If* $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c \rangle$, *then* $\forall i \in \mathrm{dom}(C), C'(i) = C(i)$.

We can analogously define a notion of validity of the Clairvoyance cost annotation contexts and show that it is preserved by evaluation.

Definition 4.2 (Valid Clairvoyance cost annotation context). An cost annotation context C is *valid* for a value $\tilde{v}$, if for any name i assigned to a thunk nested inside $\tilde{v}$, $i \in \mathrm{dom}(C)$. Moreover, C is *valid* for an environment $\tilde{\Gamma}$ if $\forall x \in \mathrm{dom}(\tilde{\Gamma})$, C is valid for $\tilde{\Gamma}(x)$.

Lemma 4.3 (Evaluation Preserves Cost Annotation Context Validity). *If* $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c \rangle$ *and* C *is valid for* $\tilde{\Gamma}$, *then* C' *is valid for* $\tilde{\Gamma}$ *and for* $\tilde{v}$.

Again, we only consider valid cost annotation contexts in what follows.

4.3 Clairvoyance evaluation cost

The total evaluation cost under the Annotated Clairvoyance Semantics can be recovered by adding up each individual cost as appeared in the annotations given to the thunks created during this evaluation, as well as the cost annotation given to the output value. We define the following operation for summing over cost annotations from a cost annotation context C:

$$\mathrm{sumcost}_A(C) := \sum_{c \in \mathrm{im}(C)} c$$

For an evaluation $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c \rangle$, the cost incurred by the evaluation can be computed by

$$c + \mathrm{sumcost}_A(C' \setminus C)$$

Like before, the difference $C' \setminus C$ removes all thunks and their annotations existing in the initial context, as we do not want to take into account any cost due to computations prior to the current one. Depending on the nondeterministic choices made during the evaluation, this total cost may be the call-by-need cost or, instead, a more eager cost.

Correspondence between the two Clairvoyance Semantics. The Annotated Clairvoyance Semantics is sound and complete with respect to the original one, and the corresponding evaluations produce exactly the same cost. To prove this claim, we first define a correspondence between values and between environments of the two semantics.

Definition 4.4 (Corresponding Clairvoyance values and environments). Let $\tilde{v}$ be a value of the Annotated Clairvoyance Semantics and $\hat{v}$ a value of the monadic Clairvoyance Semantics. They are *corresponding values* modulo names, denoted $\tilde{v} \sim \hat{v}$, if the rules in Figure 8 apply. This relation extends to environments naturally.

$$\frac{t \in \{\text{true}, \text{false}, \text{nil}, \bot\}}{t \sim t} \qquad \frac{\tilde{v}_1 \sim \hat{v}_1 \qquad \tilde{v}_2 \sim \hat{v}_2}{\text{cons } \tilde{v}_1\ \tilde{v}_2 \sim \text{cons } \hat{v}_1\ \hat{v}_2} \qquad \frac{\tilde{v}_1 \sim \hat{v}_1 \qquad \tilde{v}_2 \sim \hat{v}_2}{(\tilde{v}_1, \tilde{v}_2) \sim (\hat{v}_1, \hat{v}_2)} \qquad \frac{\tilde{v} \sim \hat{v}}{\text{thunk}_i\ \tilde{v} \sim \text{thunk } \hat{v}}$$

Fig. 8. The value relation $\sim$ between annotated clairvoyance and original clairvoyance values.

If $\tilde{v}$ amd $\hat{v}$ are related by $\sim$, the latter can be regarded as effectively abstracting away the name from the former. Furthermore, a value $\hat{v}$ in the monadic Clairvoyance Semantics corresponds to an infinite set of values $\tilde{v}$ in the Annotated Clairvoyance 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. In these two theorems we consider the evaluation of a well-typed term $\gamma : M \vdash A$.

Theorem 4.5 (Soundness of the Annotated Clairvoyance semantics wrt the monadic Clairvoyance Semantics). *Let $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c \rangle$, with the cost annotation context 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 $c + \text{sumcost}_A(C' \setminus C) = c'$.*

Theorem 4.6 (Completeness of the Annotated Clairvoyance semantics wrt the monadic Clairvoyance Semantics). *Let $\hat{\Gamma} \in [\![\gamma]\!]$ with $\tilde{\Gamma} \sim \hat{\Gamma}$ and let C be a cost annotation context valid for $\tilde{\Gamma}$. Then for all $(\hat{v}, c) \in [\![M]\!](\hat{\Gamma})$, there is an evaluation $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c' \rangle$ with $\tilde{v} \sim \hat{v}$ and $c' + \text{sumcost}_A(C' \setminus C) = c$.*

4.4 Correspondence between 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 indeed corresponds to the lazy semantics, we need to *choose* the right nondeterministic branch of Clairvoyance evaluation for a particular demand, which is difficult to do directly.

In our work, we instead take an approach inspired by the cost minimality and existence proofs of Xia et al. [2024]. That is, we prove that the cost inferred from the Memorist Semantics is never larger than any nondeterministic branch of a corresponding Clairvoyance evaluation, and that there is always a Clairvoyance branch that produces the same cost. Together they imply that the inferred cost is exactly the minimal cost that can be successfully produced on any branch of a corresponding Clairvoyance evaluation, that is, the inferred cost is indeed the correct lazy cost. However, unlike their Demand Semantics which can be viewed as backwardly inferring the demand on input from that on the output to guide the nondeterministic evaluation choices, the Memorist Semantics instead provides a summary of all individual pieces of information at the end which we can cherry-pick according to the demand of interest and derive thunk usage and cost from them. As a result, we cannot simply sum up and compare the evaluation cost step by step, since essential information including thunk usage is generally not available at the point a thunk is created (which is when it would be already evaluated or skipped in a Clairvoyance evaluation).

To address this problem we prove first that the thunk usage tracked by the Memorist semantics is both *minimal* and *sufficient* with respect to the Clairvoyance Semantics. By minimality, we mean the inferred usage set is never more than what is actually evaluated in any corresponding Clairvoyance evaluation. By sufficiency, we assert there is some corresponding Clairvoyance evaluation that evaluates exactly those thunks as recorded by the Memorist Semantics. By showing the usage captures exactly the lazy behaviour, we can then straightforwardly derive the cost correctness.

$$\frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{t \sim_f t} \qquad \frac{\tilde{v}_1 \sim_f v_1 \qquad \tilde{v}_2 \sim_f v_2}{\text{cons}\ \tilde{v}_1\ \tilde{v}_2 \sim_f \text{cons}\ v_1\ v_2} \qquad \frac{\tilde{v}_1 \sim_f v_1 \qquad \tilde{v}_2 \sim_f v_2}{(\tilde{v}_1, \tilde{v}_2) \sim_f (v_1, v_2)}$$

$$\frac{\tilde{v} \sim_f v \qquad f(i) = j}{\text{thunk}_i\ \tilde{v} \sim_f \text{thunk}_j\ v} \qquad \frac{}{\bot \sim_f \text{thunk}_j\ v}$$

Fig. 9. The relation $\sim_f$ between Memorist and Annotated Clairvoyant values using renaming function f.

Renaming. Corresponding thunks in Memorist Semantics and Annotated Clairvoyance may have different names. To address this discrepancy, we define renaming functions of type $\mathcal{N}_A \to \mathcal{N}_M$ that rename 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 will never evaluate fewer thunks than the corresponding Annotated 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. A renaming function is also required to be injective so that two names, if related under this function, are uniquely related.

Definition 4.7 (Validity of renaming functions). A renaming function $f : \mathcal{N}_A \to \mathcal{N}_M$ is *valid* with respect to some valid Memorist annotation context C and valid Clairvoyance cost annotation context C if f is injective with $\text{dom}(f) = \text{dom}(C)$ and $\text{im}(f) \subseteq \text{dom}(\mathcal{A})$.

4.5 Functional 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, parametrised by a renaming function $f : \mathcal{N}_A \to \mathcal{N}_M$.

Definition 4.8 (Value-name and environment-name correspondence). Given a renaming function f as above, value-name correspondence $\tilde{v} \sim_f v$ is defined inductively by the rules in Fig. 9. Environment-name correspondence $\tilde{\Gamma} \sim_f \Gamma$ for an annotated Clairvoyance environment $\tilde{\Gamma}$ and a Memorist environment Γ holds if $\text{dom}(\tilde{\Gamma}) = \text{dom}(\Gamma)$ and $\forall x \in \text{dom}(\Gamma), \tilde{\Gamma}(x) \sim_f \Gamma(x)$.

The relation $\sim_f$ describes the partial correspondence on evaluated values/environments and names between the two semantics. Given two environments related by $\sim_f$ through a valid renaming function f, 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 name from the Memorist evaluation.

Similarly, we define a thunkwise cost correspondence:

Definition 4.9 (Thunkwise cost correspodence). Let $\mathcal{A}$ and C be a Memorist and a Clairvoyance annotation context respectively, and f a valid thunk renaming function with respect to $\mathcal{A}, C$. $\mathcal{A}$ is *thunkwise cost corresponded* with respect to C under f, denoted $\mathcal{A} \dot{\sim}_f C$, if $\forall i \in \text{dom}(C)$, $C(i) = \pi_1(\mathcal{A}(f(i)))$.

The theorem below guarantees that, for each thunk evaluated in a Clairvoyance evaluation, there is a thunk evaluated in the corresponding Memorist evaluation with their names related by a thunk renaming function and their cost annotations being equal. Intuitively, the equality holds because any potential difference in evaluation cost is due to the Clairvoyance evaluation nondeterministically evaluating or skipping a thunk, while a cost annotation to a value records only the portion of cost incurred by an evaluation to that value that is never thunked before.

$$\frac{t \in \{\mathrm{true}, \mathrm{false}, \mathrm{nil}\}}{t \preccurlyeq t} \qquad \frac{\hat{v}_1 \preccurlyeq v_1 \qquad \hat{v}_2 \preccurlyeq v_2}{\mathrm{cons}\ \hat{v}_1\ \hat{v}_2 \preccurlyeq \mathrm{cons}\ v_1\ v_2} \qquad \frac{\hat{v}_1 \preccurlyeq v_1 \qquad \hat{v}_2 \preccurlyeq v_2}{(\hat{v}_1, \hat{v}_2) \preccurlyeq (v_1, v_2)}$$

$$\frac{\hat{v} \preccurlyeq v}{\mathrm{thunk}\ \hat{v} \preccurlyeq \mathrm{thunk}_i\ v} \qquad \frac{}{\bot \preccurlyeq \mathrm{thunk}_i\ v}$$

Fig. 10. Definition of $\preccurlyeq$ between an unnamed partially evaluated Clairvoyant value and a Memorist value

Theorem 4.10 (Functional and cost annotation correctness). *Let* $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ *and* $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c'\rangle$*, and* f *be a valid renaming function wrt* $\mathcal{A}$ *and* C*. If* $\tilde{\Gamma} \sim_f \Gamma$ *and* $\mathcal{A} \mathrel{\dot{\sim}}_f C$*, then* $\tilde{v} \sim_{f'} v$*,* $c = c'$ *and* $C' \mathrel{\dot{\sim}}_{f'} \mathcal{A}'$*, for some* f' *extending* f *and valid wrt* $\mathcal{A}'$ *and* C'*.*

4.5.1 Deriving usage based on demands. What needs to be evaluated in a lazy evaluation is driven by demand which cannot always be determined locally. The behaviour of Clairvoyance evaluation, in particular, varies according to such external demand. In Memorist Semantics, we consider evaluation with respect to demands on output via usage sets.

In the Memorist Semantics, each of the thunked and final values is annotated with a cost and a usage set. This usage set contains names of thunks needed for the evaluation to reach the outermost portion of the annotated value. As discussed previously, if more of the value is demanded, we can compute a set of names of all thunks that are needed with respect to this external demand, by collecting altogether (1) all the thunks in the value according to the demand, (2) the thunks in their usage set annotations, and (3) thunks in the usage set annotated to the output value.

Since the demand is provided externally and we do not generally know what names are given to a thunk from the outset, we need a way to abstract away from names when representing demand. For the current proof, the partially evaluated values (with unnamed thunks) defined for the monadic Clairvoyance Semantics in Section 4.1 provides a convenient way in this regards. To distinguish between the values from the two Clairvoyance Semantics in what follows, we sometimes refer to the values for the monadic Clairvoyance Semantics as *thunk-unnamed* partially evaluated values and those for the annotated variant as *thunk-named* partially evaluated values. We define the following relation to relate thunk-unnamed values with the Memorist values.

Definition 4.11 (Partially evaluated values and environments). Given a Memorist value v and a thunk-unnamed partially evaluated value $\hat{v}$ of the same type, $\hat{v}$ is a partially evaluated version of v, denoted by $\hat{v} \preccurlyeq v$, if the rules in Fig. 10 apply. The relation extends naturally to environments defined for the same type context.

Thus, for a Memorist value v, we use $\hat{v}$ with $\hat{v} \preccurlyeq v$ to represent a demand on v. The following relation characterises the set of names of all thunks needed to be evaluated given such a demand. It is easy to show that, such a set always exists for any external demand $\hat{v} \preccurlyeq v$, and that a $\hat{v}$ satisfying $\preccurlyeq$ always exists given v and a set s of thunk names.

Definition 4.12 (Usage representation sets). Given a Memorist value v, a valid annotation context $\mathcal{A}$, and a partially evaluated value $\hat{v}$ with $\hat{v} \preccurlyeq v$, the relation $\mathrm{Repr}_{\mathcal{A}}(v, \hat{v}, s)$ is defined by the rules in Figure 11. The relation extends to environments in the following way. Given a Memorist environment Γ and a partially evaluated environment $\hat{\Gamma}$ defined for the same type context with $\hat{\Gamma} \preccurlyeq \Gamma$, $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, s)$ if

$$s = \bigcup_{x \in \mathrm{dom}(\Gamma)} \{i \in s' \mid \mathrm{Repr}_{\mathcal{A}}(\Gamma(x), \hat{\Gamma}(x), s')\}$$

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

$$\frac{\mathrm{Repr}_{\mathcal{A}}(v_1, \hat{v}_1, s_1) \qquad \mathrm{Repr}_{\mathcal{A}}(v_2, \hat{v}_2, s_2)}{\mathrm{Repr}_{\mathcal{A}}((v_1, v_2), (\hat{v}_1, \hat{v}_2), s_1 \cup s_2)} \qquad \frac{\mathrm{Repr}_{\mathcal{A}}(v, \hat{v}, s) \qquad \pi_2(\mathcal{A}(i)) = s'}{\mathrm{Repr}_{\mathcal{A}}(\mathrm{thunk}_i\ v, \mathrm{thunk}\ \hat{v}, \{i\} \cup s \cup s')}$$

Fig. 11. Definition of the relation $\mathrm{Repr}_{\mathcal{A}}$, parametrised on an annotation context $\mathcal{A}$, between a Memorist value v, a demand $\hat{v}$ on v, and a set s of thunk names.

Furthermore, let (c, s') be the annotation given to v. We call the set $s \cup s'$ the *usage representation set* for the demand $\hat{v}$ on v.

Lemma 4.13. *Let v and $\hat{v}$ be a Memorist value and a thunk-unnamed partially evaluated 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 $\mathrm{Repr}_{\mathcal{A}}(v, \hat{v}, s)$. The same holds for environments in both directions.*

4.5.2 Usage minimality. As previously mentioned, cost correctness will be proved via the correctness of thunk usage. The latter is to be established in two parts, usage minimality and usage sufficiency, that together characterise the correspondence with the laziest possible Clairvoyance evaluation. We first prove that the usage representation set inferred from the Memorist Semantics is minimal, in the sense that it is never more than what is actually evaluated in any corresponding successful Clairvoyance evaluation. More 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 Annotated Clairvoyance evaluation, as related by some valid thunk renaming function.

One observation can immediately be made. If the minimality in the above sense indeed holds between a Memorist evaluation and a corresponding annotated Clairvoyance evaluation, it will be the case that, for every Memorist thunk j that corresponds to an evaluated thunk in the Clairvoyance evaluation (with their names related by a thunk renaming function f), its annotation usage set must always be a subset of the image of f, $\mathrm{im}(f)$. Recall that the $\mathrm{im}(f)$ is the set of names of all Memorist thunks that indeed correspond to some Clairvoyance thunks. This is desired; otherwise we would have discovered a thunk inferred to be needed by the Memorist evaluation whereas the successful Clairvoyance evaluation did not actually evaluate the thunk, failing minimality in the above sense. We define a relation to capture this observation.

Definition 4.14 (Thunkwise usage minimal). A Memorist annotation context $\mathcal{A}$ is *thunkwise usage minimal* with respect to a Clairvoyance annotation context C under a valid thunk renaming function f, denoted $\mathcal{A} \Subset_f C$, if $\forall i \in \mathrm{dom}(C)$, we have $\pi_2(\mathcal{A}(f(i))) \subseteq \mathrm{im}(f)$.

The validity of f ensures that $i \in \mathrm{dom}(f)$ and $f(i) \in \mathrm{dom}(\mathcal{A})$. The following lemma states that any two corresponding evaluations of some well-typed term $\gamma \vdash M : A$ preserve the thunkwise usage minimality in the above sense. In addition, the usage representation set in the annotation to the output value also satisfies a similar requirement: it is a subset of the image of the extended renaming function.

Lemma 4.15 (Thunkwise usage minimality). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ and $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c'\rangle$, and let f be a valid thunk renaming function. If $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f C$ and $\mathcal{A} \Subset_f C$, then $\mathcal{A}' \Subset_{f'} C'$ and $s \subseteq \mathrm{im}(f')$, as well as $\tilde{v} \sim_{f'} v$ and $C' \dot{\sim}_{f'} \mathcal{A}'$, for some valid renaming function f' extending f.*

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

Theorem 4.16 (Usage minimality). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ and $\tilde{\Gamma}; C \vdash M \Downarrow^{A} C'; \langle \tilde{v}, c'\rangle$, and f be a valid thunk renaming function, with $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f C$ and $\mathcal{A} \Subset_f C$. Apart from satisfying the thunkwise usage minimality as stated above for some valid renaming function f' extending f, the following also holds:*

- *Given any thunk-unnamed partially evaluated value $\hat{v}$ with $\tilde{v} \sim \hat{v}$ and $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$, we have $s \cup s' \subseteq f'[\mathrm{dom}(C')]$, and moreover, $(s \cup s') \setminus \mathrm{dom}(\mathcal{A}) \subseteq f'[\mathrm{dom}(C' \setminus C)]$.*

The additional conclusion stated with set difference allows us to excluding thunks 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 in question and inferred to be needed by the Memorist Semantics through usage tracking, always correspond (under thunk renaming f') to a subset of all thunks evaluated on any successful branch in the corresponding Clairvoyance evaluation, where the latter is captured by $\mathrm{dom}(C' \setminus C)$. That is, a corresponding Clairvoyance evaluation may evaluate more thunks, but it can never evaluated less. Thus, the inferred usage by the Memorist Semantics is minimal in this sense.

4.5.3 Usage sufficiency. Usage minimality essentially states that the usage set inferred from the Memorist semantics is ‘not too big’. Next, we prove usage sufficiency, which states the inferred usage set is ‘not too small’: for any Memorist evaluation, there is a corresponding Annotated Clairvoyance evaluation such that what is evaluated in the latter corresponds exactly to the usage set inferred by the former.

In general, we cannot assert there is always a successful Clairvoyance evaluation given an arbitrary environment $\tilde{\Gamma}$ even if it satisfies $\tilde{\Gamma} \sim_f \Gamma$ for some Memorist evaluation environment Γ and a valid renaming function f. For instance, evaluating force x in an environment $\tilde{\Gamma}$ that maps x to a skipped thunk $\perp$ can never succeed. To address this, we utilise the usage representation set, which supposedly contains all thunks needed to be evaluated, to infer a minimally workable environment. We have shown that this set is minimal in that every thunk in it must be evaluated in all corresponding successful Clairvoyance evaluation. It remains a question whether it might be ‘too minimal’ such that it misses some thunks that should have been evaluated, though it does not matter for now since we are not asserting anything new about the derived environment. We only rely on it to set a ‘lower bound’ on how less evaluated the Clairvoyance environment can be.

To do so, we first observe that, 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 $\mathcal{A}$ (where the latter contains all thunks that already exist before the evaluation), the resulting set s should contain all existing thunks that are inferred to be needed by the evaluation of interest. From s we can construct a partially evaluated environment $\hat{\Gamma}$ satisfying $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, s)$, which can be viewed as the minimal demand on the environment. We can then consider any Clairvoyance environment that is related to the given Memorist environment under $\sim_f$ and no less eagerly evaluated than the minimum environment $\hat{\Gamma}$.

We define a relation to express what it means for an Annotated Clairvoyance value or environment to be no less eagerly evaluated as a given demand.

Definition 4.17 (No less eagerly evaluated values and environments). Let $\tilde{v}$ and $\hat{v}$ be an Annotated Clairvoyance value and a thunk-unnamed partially evaluated value respectively of the same type. $\tilde{v}$ is no less eagerly evaluated than $\hat{v}$, denoted $\hat{v} \lesssim \tilde{v}$, if the rules in Figure 12 apply.

$$\frac{t \in \{\text{true}, \text{false}, \text{nil}\}}{t \lesssim t} \qquad \frac{\hat{v}_1 \lesssim \tilde{v}_1 \qquad \hat{v}_2 \lesssim \tilde{v}_2}{\text{cons}\ \hat{v}_1\ \hat{v}_2 \lesssim \text{cons}\ \tilde{v}_1\ \tilde{v}_2} \qquad \frac{\hat{v}_1 \lesssim \tilde{v}_1 \qquad \hat{v}_2 \lesssim \tilde{v}_2}{(\hat{v}_1, \hat{v}_2) \lesssim (\tilde{v}_1, \tilde{v}_2)}$$

$$\frac{\hat{v} \lesssim \tilde{v}}{\text{thunk}\ \hat{v} \lesssim \text{thunk}_i\ \tilde{v}} \qquad \frac{}{\bot \lesssim \text{thunk}_i\ \tilde{v}}$$

Fig. 12. Definition of $\lesssim$ between a thunk-unnamed partially evaluated value and a named one (i.e. a value in the Annotated Clairvoyance Semantics) of the same type.

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

We can now state and prove usage sufficiency. Below we consider a well-typed term $\gamma \vdash M : A$.

Theorem 4.18 (Usage sufficiency). *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* $\text{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$ *for some set* s' *of thunk names. There is some* $\hat{\Gamma} \in [\![\gamma]\!]$ *satisfying* $\text{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, (s \cup s') \cap \text{dom}(\mathcal{A}))$, *such that for all Annotated Clairvoyance environment* $\tilde{\Gamma}$ *with* $\hat{\Gamma} \lesssim \tilde{\Gamma}$, *a valid annotation context* C *for* $\tilde{\Gamma}$ *and a valid thunk renaming function* f *with* $\tilde{\Gamma} \sim_f \Gamma$, *the following holds.*

- *If* $f, \mathcal{A}$ *and* C *satisfy the assumptions in the demand minimality theorem, then there is some* $\tilde{\Gamma}; C \vdash M \Downarrow^A C'; \langle \tilde{v}, c' \rangle$ *with* $\hat{v} \lesssim \tilde{v}$ *and* $(s \cup s') \setminus \text{dom}(\mathcal{A}) = f[\text{dom}(C' \setminus C)]$, *where* f' *is some valid renaming function extending* f.

The theorem intuitively states that, given a Memorist evaluation with some demand, we can always construct a corresponding Clairvoyance evaluation, such that all the thunks created during the Memorist evaluation and inferred to be actually needed (as captured by the set $(s \cup s') \setminus \text{dom}(\mathcal{A})$) coincide with all the thunks evaluated within the corresponding Clairvoyance evaluation (as captured by the set $\text{dom}(C' \setminus C)$), when thunk renaming is accounted for. In other words, there is a always a corresponding Clairvoyance evaluation that evaluates exactly as much as the inferred usage from the Memorist evaluation. Hence, we say the inferred usage is sufficient.

Together, the usage minimality and sufficiency theorems imply that the inferred lazy thunk usage from the Memorist Semantics is minimal but nontrivial: it is never more than what will be evaluated in any corresponding successful Clairvoyance evaluation, while indeed corresponding to some successful Clairvoyance evaluation. To conclude, the inferred usage represents exactly what is evaluated on the laziest branch in a corresponding successful Clairvoyance evaluation.

4.6 Lazy cost correspondence

In the Memorist Semantics, lazy evaluation cost is derived based on the inferred evaluation usage and thunks' annotated cost. We have shown that, the individual cost annotations are correct (Theorem 4.10), and the inferred thunk usage corresponds to what is actually evaluated in the corresponding laziest possible Clairvoyance evaluation (Theorems 4.16 and 4.18). Consequently, the derived lazy cost from the Memorist Semantics coincides with the evaluation cost of the laziest possible Clairvoyance evaluation.

Again, since it is not straightforward to characterise the lazy Clairvoyance evaluation branch directly, we prove instead the following *cost minimality* and *cost existence* results. We consider evaluation of a well-typed term $\gamma \vdash M : A$.

Theorem 4.19 (Cost minimality). *Let $\Gamma; \mathcal{A} \vdash M \Downarrow \mathcal{A}'; \langle v, (c, s)\rangle$ and $\tilde{\Gamma}; C \vdash M \Downarrow^{A} C'; \langle \tilde{v}, c'\rangle$, and f be a valid thunk renaming function. Let $\tilde{\Gamma} \sim_f \Gamma$, $\mathcal{A} \dot{\sim}_f C$ and $\mathcal{A} \Subset_f C$.*

Then given any thunk-unnamed partially evaluated value $\hat{v}$ with $\tilde{v} \sim \hat{v}$ and $\mathrm{Repr}_{\mathcal{A}'}(v, \hat{v}, s')$, we have

$$c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s \cup s') \leq c' + \mathrm{sumcost}_A(C' \setminus C).$$

Theorem 4.20 (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 set s' of thunk names. There is some $\hat{\Gamma} \in [\![\gamma]\!]$ satisfying $\mathrm{Repr}_{\mathcal{A}}(\Gamma, \hat{\Gamma}, (s \cup s') \cap \mathrm{dom}(\mathcal{A}))$, such that for all Annotated Clairvoyance environment $\tilde{\Gamma}$ with $\hat{\Gamma} \lesssim \tilde{\Gamma}$, a valid annotation context C for $\tilde{\Gamma}$ and a valid thunk renaming function f with $\tilde{\Gamma} \sim_f \Gamma$.*

- *If $f, \mathcal{A}$ and C satisfy the assumptions in the usage minimality theorem, then there is some $\tilde{\Gamma}; C \vdash M \Downarrow^{A} C'; \langle \tilde{v}, c'\rangle$ with $\hat{v} \lesssim \tilde{v}$, and*

$$c + \mathrm{sumcost}_M(\mathcal{A}' \setminus \mathcal{A}, s \cup s') = c' + \mathrm{sumcost}_A(C' \setminus C).$$

In essence, cost minimality states that the derived lazy cost from the Memorist evaluation is no larger than the cost incurred by any corresponding Clairvoyance evaluation. Cost existence states that there is some corresponding Clairvoyance evaluation whose evaluation cost equals to the derived lazy cost from the Memorist evaluation. We thus conclude that the Memorist Semantics indeed derives the correct lazy evaluation cost.

5 Implementation

In this section, we provide a proof of concept implementation of the Memorist Semantics in Rocq Prover. Our implementation uses shallow embedding and encapsulates all the operations over thunks and annotations using a simple monadic interface.

First, we define datatypes for representing thunks and annotations:

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

We model named thunks using the above `Th` type. An annotation is a pair of a natural number (`nat`) and a finite set of natural number (`NatSet`), representing the cost and the usage set, respectively.

An inductive type like `lists` is encoded as follows:

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

We implement the Memorist evaluation using a monad `M`, encapsulating the manipulations of thunks and the accumulation of cost. The definition of `M` is as follows:

```
Record Result (A : Type) : Type := MkRes
  { val : A; annot : Annot; cont : AC; annotC := fst annot; annotU := snd annot }.
Notation "<[v, a, ac]>" := ( MkRes v a ac ).

Definition M (A : Type) : Type := AC -> Result A.
```

It represents the evaluation of an expression on an initial annotation context (`AC`) to a result (`Result`). The annotation context is encoded using an association list. The result is a record containing the result value (`val`), the annotation of the result (`annot`), and the final annotation context (`cont`). In addition, we also use `annotC` and `annotU` to represent the all thunk names and all the usage sets of `cond`, respectively. The monad `M` is essentially a generalized state monad, also known as an *update monad* [Ahman and Uustalu 2013].

```
Definition ret {A} (x : A) : M A := fun ac => <[x, (0,emptyset), ac]>.

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

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

Definition collect (i : nat) : M unit := fun ac =>
  <[tt, (0, \{i\} ∪ mapS 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 => <[tt, (1,emptyset), ac]>.
```

Fig. 13. Definitions of the monad operations and other basic operations of the metalanguage. All level and associativity declarations for notations are omitted. Given a annotation context `ac`, the function `nextIdx ac` returns a name not in the domain of `ac`, and `ext ac a` extends `ac` by mapping the next new name (as will be returned by `nextIdx`) to the annotation `a`. The expression `mapS ac i` maps a name `i` to its usage set annotation under `ac`. The notation `\{i\}` denotes a singleton set containing `i`.

All the operations over thunks and annotations in the Memorist Semantics can be encapsulated using a few combinators of the `M` monad, shown in Fig. 13. The `ret` (for return) and `bind` functions are standard monadic combinators. The `lazy` function wraps the value in `M` with a thunk; the `forcing` and `force` functions unwrap thunks. Finally, the `tick` function increases the cost by 1. The set of combinators is the same as that of the Clairvoyance Monad [Li et al. 2021].

To analyse a program, we can use the operational semantics of the Memorist Semantics (Fig. 6) as a recipe to translate a pure program to a monadic program that *reifies* the cost. Let's consider again the functions described in Section 2. We show the translation of those functions in Fig. 14.

We affix the names of the translated functions with an `M` to distinguish them from the pure original version. The translated program operates on thunked lists. Its structure follows closely that of the original definition. Function and constructor arguments are generally thunked. Whenever a function needs to apply to an argument, the thunks needs to be unwrapped first, which is done via forcing (`$!`). We separate the transformed definition of `append` into a top-level `appendM` and an auxiliary `appendM_` (similar for `truePrefix`) so they can pass Rocq Prover's termination checker directly. Since the cost model employed here is the number of function calls, we put a `tick` at the start of each call to increment cost by one. Note that we do not put ticks in the top-level definitions `appendM` and `truePrefixM`, as they are simply unwrapping the thunks around the arguments.

```
Fixpoint appendM_ {A} (xs : ListT A) (tys : Th (ListT A)) : M (ListT A) :=
  tick >>
  match xs with
  | NilT => force tys
  | ConsT tx txs =>
      lazy_letM tzs := (fun xs' => appendM_ xs' tys) $! txs in
      ret (ConsT tx tzs)
  end.

Definition appendM {A} (txs tys : Th (ListT A)) : M (ListT A) :=
  (fun xs => appendM_ xsT tys) $! txs.

Fixpoint truePrefixM_ (xs : ListT bool) : M (ListT bool) :=
  tick >>
  match xs with
  | NilT => ret NilT
  | ConsT tx txs =>
      lazy_letM tzs := truePrefixM_ $! txs in
      (fun x => if x then ret (ConsT tx tzs) else ret NilT) $! tx
  end.

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

Definition truePrefixOfAppendM (txs : Th (ListT bool)) (tys : Th (ListT bool))
  : M (ListT bool) := tick >>
  lazy_letM tzs := appendM txs tys in truePrefixM tzs.
```

Fig. 14. Translation of the functions append, `truePrefix` and `truePrefixOfAppend`.

To analyse cost, we still need some additional operations to collect thunks and summing up their cost. For the current example, we are interested in the lazy cost with respect to the length of a list demanded; accordingly, we can define the following functions to collect the names of thunks in the value that are demanded, along with their thunk usage.

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

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

As explained when introducing the formal semantics, to derive the lazy cost from the annotated result, we need to consider both thunks collected from the value according to the demand *and* the thunks in the output annotation. Moreover, we need to remove all thunks already present in the input environment. These can be realised using set union and difference. The output cost annotation must also be added in. We thus capsulate this calculation into the following function. In the definition, the expression `dom ac` gives the domain of the annotation context `ac` as a `NatSet`, and `sumcost` sums over the cost annotated to the thunk names in the given set.

```
Definition infcost {A} (r : Result A) (s : NatSet) (ac : AC) : nat :=
  let s1 := diff (annotU r ∪ s) (dom ac) in sumcost (cont r) s1 + annotC r
```

Now let us consider a concrete example. Previously in Section 2, we analysed informally the cost of the example program `truePrefixOfAppend [true;false] [true]`. Because the semantics is computable, we can obtain the concrete cost of this program by translating the program into the monadic language, running the translated program directly and applying `infcost` to derive the cost. In general, the input is a value obtained from some previous computations and thunked. For this concrete example, we simply lift them to the type `Th (ListT bool)`. This can be done automatically; more details can be found in the artefact. The following two are example transformations of the two source input lists respectively:

```
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 let `tl1` and `tl2` refer to these two lifted lists respectively and let `ac` refer to the annotation context produced by the transformation. This `ac` records the annotations of all the thunks in the lifted lists. Simply lifting the lists means each thunk so far is given the empty annotation `(0,emptyset)`, although this does not affect reasoning about the cost of `truePrefixOfAppendM` since these thunks, as part of the input, are removed via set difference when deriving the program cost.

The cost of the program can be inferred by applying `infcost` on the translated program and the set of thunk names collected according to some demand on the output. As before, we consider two different demands on the same output: one demanding only to the outermost constructor of the output list (*i.e.,* demanding 0 cons cells), and the other demanding one more cons cell—which happens to also be the last and only cons cells in the output.

```
Compute let r := truePrefixOfAppendM 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 $(3, 5)$, that is, the lazy cost with respect to the first demand is 3 while it is 5 for the second. We intentionally present it this way to highlight that, with the Memorist semantics, the program only needs to be evaluated once for analysing different demand.

The exact names of thunks and the annotations to thunks in the input lists may vary with different initial annotation contexts and different previous computations to constructing the input lists. Nevertheless, this will not change the inferred cost, as previous thunks are removed via set difference when inferring cost. Readers can find such examples of running the same program on different initial contexts in the artefact. Together with the semantics being computable, we contemplate that it may be possible to develop the semantics into a framework for testing.

More generally, we can also state and prove specifications of lazy cost for these functions. The first theorem below states that the lazy cost of `appendM` is linear in the length demanded from the output list but no more than the size of the first input list (since `append` never steps into the second input list), where the size here is measured using the function `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 described when previously when introducing the formal semantics. The `NameOk` predicate is defined for all types belonging to the `TNamed` class.

```
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.
```

We can then prove the following cost specification for `truePrefixOfAppendM`, directly expressing that the lazy cost of `truePrefixOfAppendM` in terms of the cost of `appendM` and of `truePrefixOfAppendM`. The ‘plus 1’ at the end is due to the one additional cost incurred by `truePrefixOfAppendM` itself.

```
Theorem truePrefixOfAppendM_cost :
  forall ac txs tys n d (r := truePrefixOfAppendM 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.
```

The proof, omitted here, makes direct use of the above theorems for `appendM` and `truePrefixOfAppendM`. We can also prove the following bound for `truePrefixOfAppendM`, especially when the size of the output list is smaller than that of the input list (with other conditions being the same):

```
  infcost r (usageL r n) ac <= 2 * sizeT (val r) + 1
```

This states that the cost is at most linear in the size of the output list. It is a somewhat loose bound and does not explicitly consider any demand on the final output. However, it reflects the inherent laziness in this program: when evaluating lazily, `append` need not step further after the first `false` is encountered in the (first) input lists, as it is where `truePrefix` stops processing the rest of the list.

6 Related Work

The standard call-by-need semantics. The standard operational semantics for the call-by-need evaluation is Launchbury's Natural Semantics [Launchbury 1993]. Expressions are stored unevaluated in mutable heaps, which are evaluated when needed and the results are memoised by modifying existing bindings in the heaps. An extension of the semantics in the same paper is to annotate the evaluation judgements with a count of function applications for tracking computation cost. While the Natural Semantics intuitively captures both demand-drivenness and sharing, it is difficult to reason about, thus promoting various efforts in developing new semantics for analysing call-by-need programs and their computation cost.

Clairvoyance Semantics and Clairvoyance Monad. The Clairvoyance Semantics provides an alternative approach to the standard semantics by interpreting the call-by-need evaluation as call-by-value with nondeterministic choices of proceeding or skipping certain computations [Hackett and Hutton 2019]. Li et al. [2021] defined a monadic variant of the Clairvoyance Semantics and developed it into a formal framework for verifying lazy evaluation cost. They proposed an option over a monadic *thunk* datatype to model the choice of whether to evaluate or skip the evaluation of a value of that datatype.

The monad is used for encapsulating a computation that produces a value and incurring some cost (in terms of number of ticks) and a proposition specifying some property about the computation results or cost. Due to the nondeterministic nature of the semantics, the embedded program does not compute in the usual sense, but instead leads to Rocq Prover propositions as proof obligations. The key benefit of this monadic formalisation of the Clairvoyance Semantics is that it avoids the exponential explosion in the cost analysis of the underlying nondeterministic semantics. However, it also makes programs non-executable. A user of the framework generally specifies two kinds of specifications for a program, one for all nondeterministic evaluation branches that succeed in computation, and one for the existence of a successful branch that exhibits some more specific properties (similar to Incorrectness Logic [O'Hearn 2020]). When verifying a specification, one typically needs to explicitly reason about the nondeterminism presented in the semantics by manually making the appropriate choice at the nondeterministic branches. In comparison, the Memorist Semantics is deterministic, executable, and tracks and propagates all relevant information alongside the computation. On the other hand, the Memorist Semantics tracks usage sets, which can be potentially challenging for formal reasoning in the Rocq Prover.

Demand Semantics. The more recent Demand Semantics by Xia et al. [2024] is another approach to reason about demand and cost for lazily evaluated programs. The semantics is adapted from the semantics of Bjerner and Holmström [1989] to a total and typed setting. It also adopts the monadic thunk type from the Clairvoyance framework, but assigns it an additional meaning as a representation of demand. Two kinds of demand are considered in the semantics: the output demand, which is externally given and represents the demand on the output value in the usual sense, and the input demand, which describes how evaluated the input needs to be for a call-by-need evaluation driven the output demand. The semantics is mainly concerned with inferring the minimum input demand from the output demand. To achieve this purpose, the Demand Semantics defines a forward evaluation and a backward evaluation. Both directions are deterministic, thus guaranteeing that backward inference of a minimal input demand is always possible for a given output demand. However, having two separate semantics and two copies of the code is not ideal.

The Demand Semantics aims at analysing the minimal demand on the input under lazy evaluation with respect to the demand on the output. Accordingly, the backward inference is defined to always take a specific output demand (along with the fully evaluated value) as input. For the same forward function applied on the same input (the usual input to the forward evaluation) but with a different output demand specified, the semantics must redo the evaluation with the new output demand. Meanwhile, the Memorist Semantics focuses on utilising information gathered once and for all in the analysis of subsequent computations. Analysis is based on information that is tracked and propagated during evaluation, which does not vary according to output demand. Hence, analysis for the same function requires no re-evaluation when different demands on outputs are specified.

Formally verifying the cost of lazy evaluation. Handley et al. [2020] implemented a monadic framework for analysing the computation cost of pure Haskell programs in Liquid Haskell [Vazou 2016], a proof assistant based on extending Haskell with refinement types. Nonstrictness is built into the relevant datatypes in this framework, sharing—or memoisation—must be handled by users explicitly, by inserting a `pay` construct into appropriate places of the code. This approach is inherited from an earlier work by Danielsson [2008] for verifying time cost bounds for lazy functional data structures in the theorem prover Agda, and is based on an amortised analysis technique by Okasaki [1999]. Danielsson [2008]'s library employs a type-based approach by encoding and keeping track of cost at the type level. The core element of the library is a monadic dependent *thunk* type with embedded information about cost. Cost analysis is facilitated by Agda's type inference system.

There are quite a number of recent efforts focusing on formally reasoning about the amortized cost of lazy functional data structures presented by Okasaki [1999]. For example, Pottier et al. [2024] used the Iris$^{\$}$ framework [Mével et al. 2019] to verify the banker's queue, the physicist's queue, and the implicit queue in the Rocq Prover. Their approach directly reasons about mutable cells using the Iris separation logic [Spies et al. 2022], instead of relying on any alternative semantics. Xia et al. [2024] used the Demand Semantics to formally reason about the amortized cost of the banker's queue and the implicit queue. They also formally proved that these data structures are persistent. Van Brügge [2024] used LiquidHaskell to prove the amortized cost of a simple stack, binomial heaps, and Claessen's variant of finger trees [Claessen 2020], but these proofs are done in a non-lazy setting where there is no sharing. Our work does not focus on formal reasoning, but we would also like to explore applying the Memorist Semantics to reason about amortized cost and persistence in the future.

Analysing and inferring resource bounds for call-by-need programs. Sands [1990] describes a method to mechanically analyse the time cost bounds for call-by-need functional programs based on strictness analysis which can be applied to higher-order functions by transforming programs to include additional structures containing information about function applications and the associated cost. On the automated reasoning side, Jost et al. [2017] present a type-based analysis that automatically infers the cost of call-by-need programs using the Automatic Amortised Resource Analysis (AARA) technique introduced by Hofmann and Jost [2003]. The core idea of AARA shares much resemblance to a variant of the amortised analysis technique. The notion of prepaying cost from the amortised analysis is employed to avoid duplicating the evaluation cost of shared expressions. Moreira et al. [2020] further extend the analysis from inferring linear bounds to handle univariate polynomial bounds, based on the method introduced by Hoffmann and Hofmann [2010].

Resource analysis for non-lazy languages. An abundance of work exists on resource analysis for languages that are not lazy. As an analysis technique and system for automatically inferring resource bounds for programs, the Automatic Amortised Resource Analysis mentioned above is first introduced by Hofmann and Jost [2003] for reasoning about linearly-bounded heap space cost of first-order strict functional programs, and has subsequently gone through further extension, e.g., [Hoffmann et al. 2017; Hoffmann and Hofmann 2010; Jost et al. 2010]. The analysis has been implemented in the Resource Aware ML language[Hoffmann et al. 2012]. Carbonneaux et al. [2017] develop a framework based on this analysis technique for certified automatic inference of resource bounds for low-level programs. Alongside inferring bounds for programs, the framework also generates Rocq Prover proofs to verify the bounds automatically.

Some recent efforts focus on building frameworks that unify the analysis of call-by-value and call-by-name languages, though call-by-need is often not taken into account due to difficulties with modelling laziness. The λ-amor framework developed by Rajani et al. [2021] is a time complexity analysis framework that is based on amortisation and affine types that provides unified analysis for call-by-value and call-by-name languages by directly simulating the former with the latter using a monadic approach. The dependently typed logical framework **calf** for complexity analysis, introduced by Niu et al. [2022], handles the two strategies uniformly via the use of a call-by-push-value language[Levy 2004]. The framework has been used to develop cost-aware denotational semantics for functional languages [Niu and Harper 2023] and extended by Grodin et al. [2024] to also handle other computational effects by incorporating inequational reasoning into the framework's type theory.

Another line of work follows a different approach for analysing time complexity. Namely, this approach extracts recurrence relations in terms of input size from programs, whose closed-form solution is a function expressing the cost [Cutler et al. 2020; Danner and Licata 2022; Danner et al. 2015, 2013; Kavvos et al. 2020].

On the verification side, Wang et al. [2017] introduce the TiML language with built-in support for automatic verification of time complexity bounds of programs written in this language, by allowing users to provide complexity bound specifications in type annotations which are verified via type inference. The type system of their work is inspired by dependent types and refinement types. McCarthy et al. [2018] implement a Rocq Prover library for verifying program time complexity by annotating information about cost in a monadic type. Guéneau et al. [2018] formalised the big-O notation and developed a framework based on an extension of the Separation Logic for verifying worst-case time complexity bounds for higher-order imperative programs in Rocq Prover.

7 Limitations

We note that the current work has some limitations on expressivity. Presently, the Memorist Semantics focuses on total programs with structural recursions but without general recursion. By limiting to total programs, we can keep the semantics simple and enable it to be shallow-embedded in Rocq Prover, whose specification language is itself total. However, we are also interested in extending the the current work to allow reasoning about some potentially non-terminating programs (that do terminate) and programs that work with infinite data structures. In practice, one potential way to simulate general recursion is to use fuel to limit the number of steps. Since our goal is to reason about computation cost, we will generally be considering programs that halt eventually when evaluated lazily, for which there must be fuel that is sufficient.

Another limitation is the lack of higher-order functions. Higher-order functions are powerful constructs and are frequently used in functional programming. Including them in a cost analysis framework is thus desirable, although they can also introduce quite a few complications. In the future, we would like to address the handling of higher-order functions in the Memorist Semantics.

8 Conclusion and Future Work

We have presented the Memorist Semantics, a novel cost semantics for call-by-need evaluation that tracks both usage and computation cost in a deterministic manner. Our approach advances on prior approaches in that it enables cost analysis that is independent of demand context, avoids code duplication, and provides fine-grained cost attribution to each individual component of a term. Crucially, this semantics enables analyzing the cost of lazy programs while preserving the natural deterministic compositional call-by-value evaluation structure, making it an intuitive practical foundation for further cost-aware program reasoning. Our semantics opens the door for future work on verifying compiler optimizations, such as dead code elimination based on our usage sets and verified resource-bounded computation. By formalizing our model in the Rocq Prover, we provide a rigorous basis that opens the door for the future development of cost analysis verification frameworks for real-world lazy functional languages such as Haskell. Furthermore, with the semantics being computable, lazy cost can be inferred directly via executing the shallowly embedded programs, and in principle, the evaluation only needs to be performed once for analysing the same program with various demands. Such features can potentially be helpful in areas such as testing. Another future goal is thus to investigate the application of the semantics to property-based testing.

References

Danel Ahman and Tarmo Uustalu. 2013. Update Monads: Cointerpreting Directed Containers. In *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France (LIPIcs, Vol. 26)*, Ralph Matthes and Aleksy Schubert (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1–23. https://doi.org/10.4230/LIPICS.TYPES.2013.1

Bror Bjerner and S. Holmström. 1989. 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 (FPCA '89)*. ACM Press, 157–165. https://doi.org/10.1145/99370.99382

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua M. Cohen, and Stephanie Weirich. 2021. Ready, Set, Verify! Applying `hs-to-coqm` to real-world Haskell code. *J. Funct. Program.* 31 (2021), e5.

https://doi.org/10.1017/S0956796820000283

Quentin Carbonneaux, Jan Hoffmann, Thomas W. Reps, and Zhong Shao. 2017. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10427)*, Rupak Majumdar and Viktor Kuncak (Eds.). Springer, 64–85. https://doi.org/10.1007/978-3-319-63390-9_4

Koen Claessen. 2020. Finger trees explained anew, and slightly simplified (functional pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2020, Virtual Event, USA, August 7, 2020*, Tom Schrijvers (Ed.). ACM, 31–38. https://doi.org/10.1145/3406088.3409026

Joseph W. Cutler, Daniel R. Licata, and Norman Danner. 2020. Denotational recurrence extraction for amortized analysis. *Proc. ACM Program. Lang.* 4, ICFP (2020), 97:1–97:29. https://doi.org/10.1145/3408979

Nils Anders Danielsson. 2008. Lightweight semiformal time complexity analysis for purely functional data structures. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 133–144. https://doi.org/10.1145/1328438.1328457

Norman Danner and Daniel R. Licata. 2022. Denotational semantics as a foundation for cost recurrence extraction for functional languages. *J. Funct. Program.* 32 (2022), e8. https://doi.org/10.1017/S095679682200003X

Norman Danner, Daniel R. Licata, and Ramyaa. 2015. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 140–151. https://doi.org/10.1145/2784731.2784749

Norman Danner, Jennifer Paykin, and James S. Royer. 2013. A static cost analysis for a higher-order language. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard (Eds.). ACM, 25–34. https://doi.org/10.1145/2428116.2428123

Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proc. ACM Program. Lang.* 8, POPL (2024), 273–301. https://doi.org/10.1145/3632852

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *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)*, Amal Ahmed (Ed.). Springer, 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* 3, ICFP (2019), 114:1–114:23. https://doi.org/10.1145/3341718

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2020. Liquidate your assets: reasoning about resource usage in liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL (2020), 24:1–24:27. https://doi.org/10.1145/3371092

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings (Lecture Notes in Computer Science, Vol. 7358)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, 781–786. https://doi.org/10.1007/978-3-642-31424-7_64

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential. In *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)*, Andrew D. Gordon (Ed.). Springer, 287–306. https://doi.org/10.1007/978-3-642-11957-6_16

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 185–197. https://doi.org/10.1145/604131.604148

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 223–236. https://doi.org/10.1145/1706299.1706327

Steffen Jost, Pedro B. Vasconcelos, Mário Florido, and Kevin Hammond. 2017. Type-Based Cost Analysis for Lazy Functional Languages. *J. Autom. Reason.* 59, 1 (2017), 87–120. https://doi.org/10.1007/S10817-016-9398-9

G. A. Kavvos, Edward Morehouse, Daniel R. Licata, and Norman Danner. 2020. Recurrence extraction for functional programs through call-by-push-value. *Proc. ACM Program. Lang.* 4, POPL (2020), 15:1–15:31. https://doi.org/10.1145/3371083

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 144–154. https://doi.org/10.1145/158511.158618

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer.

Yao Li, Li-yao Xia, and Stephanie Weirich. 2021. Reasoning about the garden of forking paths. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–28. https://doi.org/10.1145/3473585

Coq development team. 2024. *The Coq proof assistant*. https://doi.org/10.5281/zenodo.14542673

Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. 2018. A Coq library for internal verification of running-times. *Sci. Comput. Program.* 164 (2018), 49–65. https://doi.org/10.1016/J.SCICO.2017.05.001

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *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)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Sara Moreira, Pedro B. Vasconcelos, and Mário Florido. 2020. Resource Analysis for Lazy Evaluation with Polynomial Potential. In *IFL 2020: 32nd Symposium on Implementation and Application of Functional Languages, Virtual Event / Canterbury, UK, September 2-4, 2020*, Olaf Chitil (Ed.). ACM, 104–114. https://doi.org/10.1145/3462172.3462196

Yue Niu and Robert Harper. 2023. 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*. IEEE, 1–14. https://doi.org/10.1109/LICS56636.2023.10175777

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–31. https://doi.org/10.1145/3498670

Peter W. O'Hearn. 2020. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL (2020), 10:1–10:32. https://doi.org/10.1145/3371078

Chris Okasaki. 1999. *Purely functional data structures*. Cambridge University Press.

François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. 2024. Thunks and Debits in Separation Logic with Time Credits. *Proc. ACM Program. Lang.* 8, POPL (2024). https://hal.science/hal-04238691/file/main.pdf

Vineet Rajani, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2021. A unifying type-theory for higher-order (amortized) cost analysis. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434308

David Sands. 1990. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 432)*, Neil D. Jones (Ed.). Springer, 361–376. https://doi.org/10.1007/3-540-52592-0_74

Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. 2018. Total Haskell is reasonable Coq. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 14–27. https://doi.org/10.1145/3167092

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proc. ACM Program. Lang.* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

Jan van Brügge. 2024. Liquid Amortization: Proving Amortized Complexity with LiquidHaskell (Functional Pearl). In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 97–108. https://doi.org/10.1145/3677999.3678282

Niki Vazou. 2016. *Liquid Haskell: Haskell as a Theorem Prover*. Ph. D. Dissertation. University of California, San Diego, USA. http://www.escholarship.org/uc/item/8dm057ws

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 79:1–79:26. https://doi.org/10.1145/3133903

Li-yao Xia, Laura Israel, Maite Kramarz, Nicholas Coltharp, Koen Claessen, Stephanie Weirich, and Yao Li. 2024. Story of Your Lazy Function's Life: A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs. *Proc. ACM Program. Lang.* 8, ICFP (2024), 30–63. https://doi.org/10.1145/3674626