Story of Your Lazy Function's Life

A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs

LI-YAO XIA
LAURA ISRAEL, Portland State University, USA
MAITE KRAMARZ, McGill University, Canada
NICHOLAS COLTHARP, Portland State University, USA
KOEN CLAESSEN, Chalmers University of Technology, Sweden
STEPHANIE WEIRICH, University of Pennsylvania, USA
YAO LI, Portland State University, USA

Lazy evaluation is a powerful tool that enables better compositionality and potentially better performance in functional programming, but it is challenging to analyze its computation cost. Existing works either require manually annotating sharing, or rely on separation logic to reason about heaps of mutable cells. In this paper, we propose a bidirectional demand semantics that allows for reasoning about the computation cost of lazy programs without relying on special program logics. To show the effectiveness of our approach, we apply the demand semantics to a variety of case studies including insertion sort, selection sort, Okasaki's banker's queue, and the push function of the implicit queue. We formally prove that the banker's queue and the push function of the implicit queue are both amortized and persistent using the Rocq Prover (formerly known as Coq). We also propose the reverse physicist's method, a novel variant of the classical physicist's method, which enables mechanized, modular and compositional reasoning about amortization and persistence with the demand semantics.

ACM Reference Format:
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 (September 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The power of laziness is great, but formal reasoning with respect to its costs is notoriously elusive. After all, lazy evaluation is stateful, it produces interleaved computation, and function costs under it depend on future demand. We believe that mechanized reasoning can help ensure the rigor of the analysis needed to understand these costs. To realize this vision, we present a shallow-embedding-based model of cost analysis. Our approach allows us to mechanically reason about the computational cost of lazy functional programs and lazy, amortized, and persistent functional data structures.

Our solution is based on a *bidirectional demand semantics*. The semantics was first described by Bjerner and Holmström [1989] in an untyped setting. We adapt and expand it to a typed and total semantics. Given a lazy function $f : A \longrightarrow B$, we can use the bidirection demand semantics to

Authors' addresses: Li-yao Xia, lysxia@gmail.com; Laura Israel, laisrael@pdx.edu, Computer Science, Portland State University, Portland, OR, USA; Maite Kramarz, maite.kramarz@gmail.com, Computer Science, McGill University, Montreal, Quebec, Canada; Nicholas Coltharp, coltharp@pdx.edu, Computer Science, Portland State University, Portland, OR, USA; Koen Claessen, koen@chalmers.se, Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden; Stephanie Weirich, sweirich@cis.upenn.edu, Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA; Yao Li, liyao@pdx.edu, Computer Science, Portland State University, Portland, OR, USA.

systematically derive a demand function $f^D : A \to B^D \to \mathbb{N}\times A^D$, where A^D represents the demand on type A and $\mathbb{N}$ is the computation cost. That is, given the input to function f and the demand on its output, we can *calculate* the minimal demand on the input, as well as the computation cost required to obtain the demanded output. In a sense, the demand function tells the "story of a lazy function's life," including what happens when it is subjected to future demands. We can calculate such an input demand because, in a deterministic language, given any valid output demand, there exists *exactly one* minimal input demand. The use of input/output demand and demand functions distinguishes our work from alternative approaches based on heaps of mutable cells, such as Mével et al. [2019] and Pottier et al. [2024], which rely on separation logics.

To demonstrate the effectiveness of our demand semantics, we have developed a wide variety of case studies. We use our model to formally prove the computation cost of lazy insertion sort, lazy selection sort, Okasaki's *banker's queue* and *implicit queue* [Okasaki 1999]. For the banker's queue and the implicit queue, we also show that these data structures are both amortized and persistent.

To reason about amortization and persistence in a modular way, we propose the *reverse physicist's method*, a novel variant of the classical physicist's method for amortized computational complexity analysis in strict semantics [Tarjan 1985]. Similar to the classical physicist's method, the reverse physicist's method makes use of a *potential* function, which we apply to approximations of datatypes to describe their *accumulated* potential. All proofs are mechanized using the Rocq Prover (formerly known as the Coq theorem prover). All Rocq Prover definitions and proofs can be found in our supplementary materials.

In summary, we make the following contributions:

- We propose a bidirectional demand semantics for lazy functional programs based on Bjerner and Holmström [1989] (Section 3). Our demand semantics is typed and total, allowing it to be formalized in proof assistants such as the Rocq Prover.
- We formally prove that the bidirectional demand semantics is equivalent to the natural semantics of laziness [Launchbury 1993] in the Rocq Prover, by showing its equivalence to another semantics that is equivalent to natural semantics, namely clairvoyant semantics [Hackett and Hutton 2019; Li et al. 2021] (Section 3.3).
- We show how the demand semantics can be used to systematically derive demand functions for a realistic programming language by proving computation cost theorems for insertion sort and selection sort (Section 4).
- We propose the *reverse physicist's method*, a novel method for analyzing amortized computation cost and persistence for lazy functional data structures based on demand semantics (Section 5).
- We present a *mechanized proof* in the Rocq Prover which shows that Okasaki's banker's queue and implicit queue are amortized and persistent using the reverse physicist's method. Our mechanized proof does not rely on trusting the demand functions (Section 5.3–5.4).

In addition, we provide a sketch of our method with motivating examples in Section 2. We discuss related work in Section 6 and future work in Section 7.

2 MOTIVATING EXAMPLES

2.1 A Demand Semantics

To introduce and motivate our method of reasoning about lazy programs, we will consider `insertion_sort` as an example. This algorithm is known to run in $O(n^2)$ time for a list of length n under eager evaluation. In contrast, a lazy implementation only computes each successive sorted element when a result is demanded, potentially causing asymptotically lower time costs if only part of the list is needed.

```
Fixpoint insert (x : nat) (xs : list nat) : list nat :=
  match xs with
  | [] => x :: []
  | y :: ys => if y <=? x then
                 let zs := insert x ys in
                 y :: zs
               else x :: y :: ys
  end.

Fixpoint insertion_sort (xs : list nat) : list nat :=
  match xs with
  | nil => nil
  | y :: ys =>
      let zs := insertion_sort ys in
      insert y zs
  end.
```

Fig. 1. The Gallina implementation of take, insert, and insertion_sort in ANF (A-normal form) [Sabry and Felleisen 1992]. For simplicity, we define these functions on lists of natural numbers (nat in Gallina). The infix operator <=? shown in insert is Gallina's "less than or equal" operator on natural numbers, which returns a boolean.

To model the improvements to the computation cost of a lazy insertion_sort, we first compose the take and insertion_sort functions (Fig. 1). We implement these functions in Gallina, the underlying specification language of the Rocq Prover. Even though Gallina is not a lazy language, we can imagine a "translator" that converts these functions from a lazy functional language to Gallina (*e.g.*, hs-to-coq for Haskell [Breitner et al. 2021]).

Demand data types. To model laziness, we first need a notion of input and output demand. We use the *approximation data types* proposed by Li et al. [2021] to represent both *approximations* and *demands*. An approximation is a partial value with a placeholder for unevaluated thunks. For example, the finite list data structure has the approximation data type listA:

```
Inductive listA (a : Type) : Type :=
  NilA | ConsA (x : T a) (xs : T (listA a)).
```

The T data type is a sum type that can be either an unevaluated thunk Undefined (also denoted as ⊥) or an evaluated value Thunk a. Approximation data types can be used to specify demands—*e.g.*, a listA nat of ConsA (Thunk 0) Undefined represents a demand on a list nat such that only the first item in the list must be evaluated *and* it evaluates to 0.

Definedness relations and operations. Before we can state any theorems, we need a definedness order between ordinary data types and approximation data types, as well as between different approximation data types. For example, consider the following two approximation data types:

```
Definition lA1 := ConsA (Thunk 0) Undefined.
Definition lA2 := ConsA (Thunk 0) (ConsA (Thunk 1) Undefined).
```

We say that lA1 is *less defined* than lA2 because lA2 is defined on the first and second elements of the list, while lA1 is defined only on the first. We also say that both lA1 and lA2 are *approximations*

```
 1  Fixpoint insertD (x:nat) (xs: list nat)
 2    (outD : listA nat) : Tick (T (listA nat)) :=
 3    tick >>
 4    match xs, outD with
 5    | [], ConsA zD zsD =>
 6        ret (Thunk NilA)
 7    | y :: ys, ConsA zD zsD =>
 8       if y <=? x then
 9         let+ ysD := thunkD (insertD x ys) zsD in
10         ret (Thunk (ConsA (Thunk y) ysD))
11       else
12         ret zsD
13    | _ , _ => bottom (* absurdity case *)
14    end.
15
16  Fixpoint insertion_sortD (xs: list nat)  (outD : listA nat) :
17    Tick (T (listA nat)) :=
18    tick >>
19    match xs with
20    | [] => ret (Thunk NilA)
21    | y :: ys =>
22        let zs := insertion_sort ys in
23        let+ zsD := insertD y zs outD in
24        let+ ysD := thunkD (insertion_sortD ys) zsD in
25        ret (Thunk (ConsA (Thunk y) ysD))
26    end.
```

Fig. 2. The demand functions of `insert` and `insertion_sort`.

of `[0; 1; 2]`, but only `lA1` is an approximation of `[0; 2]`. We defer formal definitions of these definedness orders to Section 3.1.

Demand functions. Each lazy function has a corresponding *demand function*. Given an *output demand*, the demand function computes the *minimal input demand* required for the function to satisfy the output demand, as well as the time cost incurred. We show the demand functions of `insert` and `insertion_sort` in Fig. 2. As a convention, we suffix a function's name with capital letter `D` to indicate that it is a demand function.

We first look at `insertD`, the demand function of `insert`. In addition to the arguments of `insert` (line 1), the demand function `insertD` takes an extra argument `(outD : listA nat)`, which represents the output demand (line 2). The function returns the minimal input demand `T (listA nat)` and wraps it in a `Tick` data type (line 2). `Tick` is a monad defined as `Tick a = nat * a`, where `nat` is the type of natural numbers representing the time cost of a wrapped function. It is essentially a writer monad specialized to `nat`.

The `tick` operation increments the time cost by one (line 3). We count the number of function calls by invoking `tick` at the beginning of every function. The function then matches on its input as well as the output demand (line 4).

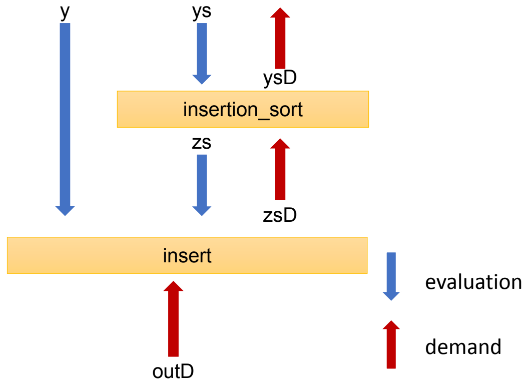

Fig. 3. An illustration of how the input demand is computed in `insertion_sortD`.

If `xs` is an empty list, we know from the definition of `insert` that the output demand must be an approximation of a `x :: nil`, so we also match `outD` to ensure that it has the form `ConsA zD zsD` (line 5). Based on the implementation of `insert`, we also must evaluate the pattern matching to determine that the input is empty, so we return the minimal input demand as `NilA` (line 6).

If `xs` is not an empty list, we know from `insert` that the output must be an approximation of a non-empty list, so we match `outD` to the form `ConsA zD zsD` (line 7). Like in `insert`, we then proceed to check whether `y <=? x` (line 8). If `y <=? x`, we make a recursive call to `insertD` to get the input demand of the recursive call to `insert` (lines 9–10). Otherwise, we return `zsD` (line 12). The `let+` notation on line 9 is a custom notation that represents a monadic bind. The `thunkD` combinator on line 9 is a function of type `(A -> B) -> T A -> T B`, *i.e.*, it applies a function to data wrapped in a Thunk.

The demand function `insertD` has a similar structure to `insert`. This is not a coincidence. We will show that, given a pure function, we can systematically derive its demand function in Section 3.

Bidirectionality. Figure 2 shows `insertion_sortD`, the demand function of `insertion_sort`, which has a more advanced implementation than `insertD`. The first case (line 20) is straightforward, but the second case (line 21) is more complex. This is because `insertion_sort` first calls itself recursively, and then applies `insert` to the result of the recursive call (lines 14–15 in Fig. 1). In Fig. 3, we illustrate how we compute the input demand for `insertion_sortD` via a demand dependency graph of `insertion_sort`. We start with the input `y` and `ys`, as well as the output demand `outD`. To calculate the input demand of a `selection_sort`, we need the input demand of the recursive call `ysD`. However, `ysD` relies on the input `ys` and the output demand of the recursive call `zsD`, which in turn relies on the input `y`, the output demand `outD`, and the input `zs` which is the result of evaluating `insertion_sort`.

Fig. 3 shows that the demand function needs to run computation in both directions: an *evaluation* direction that computes output from input (blue arrows in the figure), and a *demand* direction that computes the input demand from pure input and output demand (red arrows in the figure). It also reveals how we should define `insertion_sortD`. We first call `insertion_sort` (line 22, Fig. 2), then `insertD` (line 23), and finally `insertion_sortD` (line 24).

Specification and proof sketches. From here, we can prove properties about these demand functions. For example, we can prove the following theorems for `insertion_sortD`:

```
Theorem insertion_sortD_approx (xs : list nat) (outD : listA nat)
  : outD `is_approx` insertion_sort xs ->
    Tick.val (insertion_sortD xs outD) `is_approx` xs.

Theorem insertion_sortD_cost (xs : list nat) (outD : listA nat) :
  Tick.cost (insertion_sortD xs outD) <= (sizeX' 1 outD + 1) * (length xs + 1).
```

The theorem insertion_sortD_approx describes the *functional correctness* of insertion_sortD. It states that, if the given output demand is an approximation of the output, then the input demand is an approximation of the original input.

The theorem insertion_sortD_cost describes the *time cost* of insertion_sortD. It states that the cost of insertion_sort is bounded by $(\max(1, |\texttt{outD}|) + 1) \times (|\texttt{xs}| + 1)$. In other words, for each element of the sorted list we compute, we will have to linearly search through the input list once (plus some constant overhead). This is an overestimation—the input list will shrink after each recursive call and we don't have to go through the entire list when inserting a new elements—but this still provides a useful upper bound.

As programmers, we may also be interested in the demands exerted by functions that call the functions we analyze. Using take as an example, we can prove that the demand of takeD n xs outD has a size bounded by the length of outD. Using this lemma, we can then describe the cost of running a function composing take with insertion_sort, and then prove that the composed function satisfies that cost. This new cost is linear with respect to n so long as we use a constant n, which is an asymptotic improvement over the eager version of insertion_sort. This formalizes a common pattern in lazy programming—computing only the necessary parts of an expensive function call to reduce costs.

This example demonstrates another advantage of our approach: compositionality. Even though a functional call under lazy evaluation is not local (as a future demand may cause the function to run further), we can specify each function individually using demand semantics. If we wish to compose these functions with others and reason about the cost of their composition, we can do so by proving theorems for external functions individually and composing their specifications. For example, we may want to compose head with a different sorting algorithm, *e.g.*, selection_sort, or compose a sorting algorithm with the more general take function that takes the first *n* elements from a list. We demonstrate how we use demand semantics to reason about all these interactions in Section 4.

2.2 Amortized and Persistent Data Structures

Besides the usual benefits of lazy evaluation such as compositionality [Hughes 1989], lazy evaluation also enables a data structure to be both *amortized* and *persistent* [Okasaki 1999]. Take a first-in-first-out (FIFO) queue as an example. The queue is amortized if the amortized cost of each operation is constant, regardless of the cost of any individual expensive operation. The queue is persistent if the amortized constant cost holds regardless of how many times we use a copy of the queue. Due to the power of these properties, our work takes a particular interest in verifying that they hold for certain lazy functional data structures.

In this paper, we formally verify that both the banker's queue and the implicit queue are amortized and persistent using the Rocq Prover. Our final theorems show that the cost of executing a program trace of using either the banker's queue or the implicit queue is always bounded by a constant factor.

Proving these theorems directly is challenging. Instead of reckoning with arbitrary program traces for each queue, we develop a modular framework that allows us to reason about the cost of both queues' functions based on their respective demand functions. The framework is based on the *reverse physicist's method*, a novel variant of the classical physicist's method that we propose in this paper. We provide more details about the verification of the banker's queue and implicit queue in Section 5.

2.3 Limitations

While the demand semantics can be used to systematically translate from a pure function to a demand function, we have not implemented an automatic translator. This is because we are interested in reasoning algorithms and data structures implemented in real-world programming languages, but developing an automatic translator for such a programming language with complicated features is beyond the scope of this work. Instead, all the demand functions shown here are manually translated according to the demand semantics.

Our bidirectional demand semantics does not support general higher-order functions, which limits the expressiveness of our demand semantics. However, we show that the restrictive demand semantics is still useful for mechanically reasoning about the time cost of many lazy functions as well as amortized and persistent data structures.

The demand function can reason about total time cost, but not real-time cost, space cost or resource in general. This is because demand functions only calculate *if* a thunk is evaluated, not *when* a thunk is evaluated.

3 BIDIRECTIONAL DEMAND SEMANTICS

In this section, we show the definition of our demand semantics (Section 3.1) and its properties (Section 3.2). To show that the demand semantics correctly models laziness, we show its equivalence with natural semantics [Launchbury 1993] by showing its equivalence to clairvoyant semantics [Hackett and Hutton 2019; Li et al. 2021] (Section 3.3). All the lemmas and theorems shown in this section have been formally proven in the Rocq Prover.

3.1 Lazy semantics

Syntax. We consider a pure, total, first-order calculus with explicit thunks. Evaluation is eager by default. This calculus can be viewed either as a subset of ML with a type for memoized thunks (*e.g.*, `lazy` in OCaml, denoted `T` here), similar to the language used in Okasaki [1999], or as an intermediate representation which lazy languages such as Haskell can be translated into. Making explicit the constructs (lazy and force) to manipulate thunks makes our semantics rather simple.

We show the syntax of the language in Fig. 4 and the typing rules in Fig. 5. The language includes types such as booleans, lists, products, and a thunk type `T` that is a sum type of unevaluated thunk and evaluated value.[1] The language is first-order: higher-order functions are not allowed, as evidenced by the lack of function type, but it is equipped with a primitive foldr for defining recursions. Most of the language's operators are standard. The tick operation increases the count of the current computation cost by 1. The lazy and force operations are the opposite of each other: lazy creates a new thunk `T` and force forces a thunk `T` to evaluate.

Semantics. Given a lazy function, we can automatically translate it to a demand function. We present such a translation as a denotational semantics. The semantics is compositional: the denotation of a term is a function of the denotation of its immediate subterms. Thus, it can be viewed as a translation from our calculus to a metalanguage able to express that function for every term constructor, essentially requiring pattern-matching and recursion on the list approximation type.

[1] This is the same datatype `T` as shown in Section 2.

$$\begin{aligned}
A, B &::= \mathrm{bool} \mid \mathrm{list}\,A \mid \mathrm{T}\,A \mid A \times B \\
M, N &::= x \mid \mathrm{let}\,x = M\,\mathrm{in}\,N \mid \mathrm{tick}\,M \mid \mathrm{lazy}\,M \mid \mathrm{force}\,M \\
&\quad\mid \mathrm{cons}\,M\,N \mid \mathrm{nil} \mid \mathrm{foldr}\,(\lambda\,x\,y.\,M_1)\,M_2\,M_3 \\
&\quad\mid \mathrm{pair}\,M\,N \mid \mathrm{fst}\,M \mid \mathrm{snd}\,M \mid \mathrm{true} \mid \mathrm{false} \mid \mathrm{if}\,M_1\,M_2\,M_3
\end{aligned}$$

Fig. 4. Syntax

$$\textsc{Var}\ \frac{\Gamma(x) = A}{\Gamma \vdash x : A} \qquad \textsc{Let}\ \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash N : B}{\Gamma \vdash \mathrm{let}\,x = M\,\mathrm{in}\,N : B} \qquad \textsc{Tick}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{tick}\,M : A}$$

$$\textsc{Lazy}\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{lazy}\,M : TA} \qquad \textsc{Force}\ \frac{\Gamma \vdash M : T\,A}{\Gamma \vdash \mathrm{force}\,M : A}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash M : T\,A \qquad \Gamma \vdash N : T\,(\mathrm{list}\,A)}{\Gamma \vdash \mathrm{cons}\,M\,N : \mathrm{list}\,A}$$

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

Fig. 5. Typing rules.

$$\begin{aligned}
\llbracket A \rrbracket_{\mathrm{eval}} &: \mathrm{Set} \\
\llbracket \mathrm{bool} \rrbracket_{\mathrm{eval}} &= \{0, 1\} \\
\llbracket \mathrm{T}\,A \rrbracket_{\mathrm{eval}} &= \llbracket A \rrbracket_{\mathrm{eval}} \\
\llbracket \mathrm{list}\,A \rrbracket_{\mathrm{eval}} &= \{\mathrm{nil}\} \uplus \left\{\mathrm{cons}\,a\,b \mid a \in \llbracket A \rrbracket_{\mathrm{eval}}, b \in \llbracket \mathrm{list}\,A \rrbracket_{\mathrm{eval}}\right\}
\end{aligned}$$

$$\begin{aligned}
\llbracket A \rrbracket_{\mathrm{approx}} &: \mathrm{Set} \\
\llbracket \mathrm{bool} \rrbracket_{\mathrm{approx}} &= \{0, 1\} \\
\llbracket \mathrm{T}\,A \rrbracket_{\mathrm{approx}} &= \mathrm{T}\,\llbracket A \rrbracket_{\mathrm{approx}} \overset{\mathrm{def}}{=} \{\bot\} \uplus \left\{\mathrm{thunk}\,a \mid a \in \llbracket A \rrbracket_{\mathrm{approx}}\right\} \\
\llbracket \mathrm{list}\,A \rrbracket_{\mathrm{approx}} &= \{\mathrm{nil}\} \uplus \left\{\mathrm{cons}\,a\,b \mid a \in \mathrm{T}\,\llbracket A \rrbracket_{\mathrm{approx}}, b \in \mathrm{T}\,\llbracket \mathrm{list}\,A \rrbracket_{\mathrm{approx}}\right\}
\end{aligned}$$

Fig. 6. Sets of total values $\llbracket A \rrbracket_{\mathrm{eval}}$ and sets of approximations $\llbracket A \rrbracket_{\mathrm{approx}}$

The denotational semantics consists of two denotation functions on well-typed terms $\Gamma \vdash M : A$, a pure forward interpretation $\llbracket M \rrbracket_{\mathrm{eval}}$ and a demand interpretation $\llbracket M \rrbracket_{\mathrm{dem}}$.

$$
\begin{aligned}
\llbracket \Gamma \vdash M : A \rrbracket_{\text{eval}} &: \llbracket \Gamma \rrbracket_{\text{eval}} \longrightarrow \llbracket A \rrbracket_{\text{eval}} \\
\llbracket x \rrbracket_{\text{eval}}(g) &= g(x) \\
\llbracket \text{force}\, M \rrbracket_{\text{eval}}(g) &= \llbracket M \rrbracket_{\text{eval}}(g) \\
\llbracket \text{lazy}\, M \rrbracket_{\text{eval}}(g) &= \llbracket M \rrbracket_{\text{eval}}(g) \\
\llbracket \text{let}\, x = M \,\text{in}\, N \rrbracket_{\text{eval}}(g) &= \llbracket N \rrbracket_{\text{eval}}(\{g, x \mapsto \llbracket M \rrbracket_{\text{eval}}(g)\}) \\
\llbracket \text{cons}\, M\, N \rrbracket_{\text{eval}}(g) &= \textbf{cons}\ \llbracket M \rrbracket_{\text{eval}}(g)\ \llbracket N \rrbracket_{\text{eval}}(g) \\
\llbracket \text{nil} \rrbracket_{\text{eval}}(g) &= \textbf{nil} \\
\llbracket \text{foldr}\ (\lambda xy.M_1)\ M_2\ N \rrbracket_{\text{eval}}(g) &= \textbf{foldr}_{\text{eval}}(g, M_1, M_2, \llbracket N \rrbracket_{\text{eval}}(g)) \\
\textbf{foldr}_{\text{eval}}(g, M_1, M_2, \textbf{nil}) &= \llbracket M_2 \rrbracket_{\text{eval}}(g) \\
\textbf{foldr}_{\text{eval}}(g, M_1, M_2, (\textbf{cons}\, a_1\, a_2)) &= \llbracket M_1 \rrbracket_{\text{eval}}(\{g, x \mapsto a_1, y \mapsto \textbf{foldr}_{\text{eval}}(g, M_1, M_2, a_2)\})
\end{aligned}
$$

Fig. 7. Forward evaluation.

$$
\begin{aligned}
\llbracket \Gamma \vdash M : A \rrbracket_{\text{dem}} &: \llbracket \Gamma \rrbracket_{\text{eval}} \times \llbracket A \rrbracket_{\text{approx}} \rightharpoonup \mathbb{N} \times \llbracket \Gamma \rrbracket_{\text{approx}} \\
\llbracket x \rrbracket_{\text{dem}}(g, a) &= (0, \{x \mapsto a\}) \\
\llbracket \text{tick}\, M \rrbracket_{\text{dem}}(g, a) &= (1 + c, d) \qquad \text{where } (c, d) = \llbracket M \rrbracket_{\text{dem}}(g, a) \\
\llbracket \text{force}\, M \rrbracket_{\text{dem}}(g, a) &= \llbracket M \rrbracket_{\text{dem}}(g, \textbf{thunk}\, a) \\
\llbracket \text{lazy}\, M \rrbracket_{\text{dem}}(g, \bot) &= (0, \bot_g) \\
\llbracket \text{lazy}\, M \rrbracket_{\text{dem}}(g, \textbf{thunk}\, a) &= \llbracket M \rrbracket_{\text{dem}}(g, a) \\
\llbracket \text{let}\, x = M \,\text{in}\, N \rrbracket_{\text{dem}}(g, a) &= (c_N + c_M, d_N \sqcup d_M) \\
&\qquad \text{where } (c_N, \{d_N, x \mapsto b\}) = \llbracket N \rrbracket_{\text{dem}}(\{g, x \mapsto \llbracket M \rrbracket_{\text{eval}}(g)\}, a) \\
&\qquad \text{and } (c_M, d_M) = \llbracket M \rrbracket_{\text{dem}}(g, b) \\
\llbracket \text{cons}\, M\, N \rrbracket_{\text{dem}}(g, \textbf{cons}\, a\, b) &= \llbracket M \rrbracket_{\text{dem}}(g, a) \uplus \llbracket N \rrbracket_{\text{dem}}(g, b) \\
&\qquad \text{where } (c_M, d_M) \uplus (c_N, d_N) = (c_M + c_N, d_M \sqcup d_N) \\
\llbracket \text{foldr}\ (\lambda xy.M_1)\ M_2\ N \rrbracket_{\text{dem}}(g, d) &= (c, g') \uplus \llbracket N \rrbracket_{\text{dem}}(g, n') \\
&\qquad \text{where } (c, g', n') = \textbf{foldr}_{\text{dem}}(g, M_1, M_2, \textbf{thunk}\ (\llbracket N \rrbracket_{\text{eval}}(g)), \textbf{thunk}\, d)
\end{aligned}
$$

Fig. 8. Demand semantics: backward evaluation.

Forward evaluation. The "forward evaluation" $\llbracket M \rrbracket_{\text{eval}} : \llbracket \Gamma \rrbracket_{\text{eval}} \longrightarrow \llbracket A \rrbracket_{\text{eval}}$ is the natural functional interpretation. Our calculus is total, so all terms have a value. Thunks can always be evaluated, so the interpretation of a lifted type $\llbracket T\, A \rrbracket_{\text{eval}}$ is the same as the unlifted $\llbracket A \rrbracket_{\text{eval}}$. The demand semantics [Bjerner and Holmström 1989] is defined by "backwards evaluation": $\llbracket M \rrbracket_{\text{dem}} : \llbracket \Gamma \rrbracket_{\text{eval}} \times \llbracket A \rrbracket_{\text{approx}} \rightharpoonup \mathbb{N} \times \llbracket \Gamma \rrbracket_{\text{approx}}$.

Lattice of approximations. Intuitively, lazy evaluation is driven by demand: the evaluation of a term depends on how much of its result will be needed. Let us first describe the representation and structure of demands as *approximations*. The set of approximations $\llbracket A \rrbracket_{\text{approx}}$ consists of values

$$\begin{aligned}
\mathbf{foldr}_{\mathrm{dem}}(g, M_1, M_2, n, \bot) &= (0, \bot_g, \bot)\\
\mathbf{foldr}_{\mathrm{dem}}(g, M_1, M_2, \mathbf{thunk}\ \mathbf{nil}, \mathbf{thunk}\ d) &= \llbracket M_2 \rrbracket_{\mathrm{dem}}(g, d)\\
\mathbf{foldr}_{\mathrm{dem}}(g, M_1, M_2, \mathbf{thunk}\ (\mathbf{cons}\ a_1\ a_2), \mathbf{thunk}\ d) &= (c_1 + c_2, g_1 \sqcup g_2, \mathbf{thunk}\ (\mathbf{cons}\ a_1'\ a_2'))\\
\text{where } (c_1, \{g_1, x \mapsto a_1', y \mapsto b_2'\}) &= \llbracket M_1 \rrbracket_{\mathrm{dem}}(\{g, x \mapsto a_1, y \mapsto \mathbf{foldr}_{\mathrm{eval}}(g, M_1, M_2, a_2)\}, d)\\
\text{and } (c_2, g_2, a_2') &= \mathbf{foldr}_{\mathrm{dem}}(g, M_1, M_2, a_2, b_2')
\end{aligned}$$

Fig. 9. Definition of $\mathbf{foldr}_{\mathrm{dem}}$

$$\frac{}{a \le a}\text{reflexivity} \qquad \frac{}{\bot \le a}\bot\text{-least} \qquad \frac{a \le b}{\mathbf{thunk}\ a \le \mathbf{thunk}\ b}\mathbf{thunk} \qquad \frac{a \le c \qquad b \le d}{\mathbf{cons}\ a\ b \le \mathbf{cons}\ c\ d}\mathbf{cons}$$

Fig. 10. Definedness order $a \le b$

$$\frac{}{\bot \prec a}\bot\text{-least} \qquad \frac{c \in \{\mathrm{nil}, \mathrm{true}, \mathrm{false}\}}{c \prec c}{\prec}\text{-}c$$

$$\frac{a \prec b}{\mathbf{thunk}\ a \prec b}{\prec}\text{-}\mathbf{thunk} \qquad \frac{a \prec c \qquad b \prec d}{\mathbf{cons}\ a\ b \prec \mathbf{cons}\ c\ d}{\prec}\text{-}\mathbf{cons}$$

Fig. 11. Approximation relation $a \prec b$

with the same shape as in $\llbracket A \rrbracket_{\mathrm{eval}}$, possibly with some subterms replaced with a special value $\bot$ representing an unneeded thunk. $\llbracket A \rrbracket_{\mathrm{approx}}$ is defined formally in Figure 6. Approximations are ordered by definedness. This partial order, denoted $a \le b$, is defined inductively in Figure 11: $\bot \le a$ for all a. The $\le$ relation is reflexive, and all constructors (**cons** and **thunk** in our calculus) are monotone. By contrast, we say that elements of $\llbracket A \rrbracket_{\mathrm{eval}}$ are *total values*.

An approximation value $a' : \llbracket A \rrbracket_{\mathrm{approx}}$ is an approximation of a total value $a : \llbracket A \rrbracket_{\mathrm{eval}}$, a relation denoted $a' \prec a$, when informally "a' has the same shape as a", ignoring **thunk**s, and some subterms of a may have been replaced with $\bot$ in a'. The set of approximations of a is defined by

$$\llbracket A \rrbracket_{\mathrm{approx}}^{\prec a} \stackrel{\mathrm{def}}{=} \left\{a' \in \llbracket A \rrbracket_{\mathrm{approx}} \mid a' \prec a\right\}$$

The following transitivity property composes $\le$ and $\prec$ into $\prec$.

Lemma 3.1 (transitivity). *If $a' \le a''$ and $a'' \prec a$ then $a' \prec a$.*

The set $\llbracket A \rrbracket_{\mathrm{approx}}^{\prec a}$ is a semi-lattice: two approximations $a_1 \prec a$ and $a_2 \prec a$ can be joined into a supremum $a_1 \sqcup_A a_2$ (abbreviated as $a_1 \sqcup a_2$ when the type is clear): it is the smallest approximation of a more defined than both a_1 and a_2. The join is defined formally in Figure 12; it is also extended to environments $\{x_i \mapsto a_i\}_i : \llbracket \Gamma \rrbracket_{\mathrm{approx}}$. We remark that a meet $a_1 \sqcap_A a_2$ could also be defined so that approximations of an element form a lattice, but we won't need it.

Lemma 3.2 (Supremum). *For all $a_1, a_2 \prec a$,*

(1) $a_1 \sqcup a_2 \prec a$

$$
\begin{aligned}
\sqcup : [\![A]\!]^{\prec a}_{\text{approx}} \times [\![A]\!]^{\prec a}_{\text{approx}} &\to [\![A]\!]^{\prec a}_{\text{approx}} \\
\bot \sqcup_{T\,A} \bot &= \bot \\
\bot \sqcup_{T\,A} \textbf{thunk}\, a &= \textbf{thunk}\, a \\
\textbf{thunk}\, a \sqcup_{T\,A} \bot &= \textbf{thunk}\, a \\
\textbf{thunk}\, a \sqcup_{T\,A} \textbf{thunk}\, b &= \textbf{thunk}\, (a \sqcup_A b) \\
\textbf{cons}\, a\, b \sqcup_{\text{list}\,A} \textbf{cons}\, c\, d &= \textbf{cons}\, (a \sqcup_{T\,A} c)\, (b \sqcup_{T\,(\text{list}\,A)} d) \\
\textbf{nil} \sqcup_{\text{list}\,A} \textbf{nil} &= \textbf{nil} \\
\{x \mapsto g_x\}_{(x:A)\in\Gamma} \sqcup_\Gamma \{x \mapsto g'_x\}_{(x:A)\in\Gamma} &= \{x \mapsto g_x \sqcup_A g'_x\}_{(x:A)\in\Gamma}
\end{aligned}
$$

Fig. 12. Joining approximations (NB: cases with mismatched constructors cannot happen)

$$
\begin{aligned}
\bot_{-} &: [\![A]\!]_{\text{eval}} \to [\![A]\!]_{\text{approx}} \\
\bot_x &= \bot \quad \text{if } \exists B,\ A = TB \\
\bot_{\textbf{cons}\, a\, b} &= \textbf{cons}\, \bot\, \bot \\
\bot_{\textbf{nil}} &= \textbf{nil} \\
\bot_{\{x_i \mapsto a_i\}_{i\in I}} &= \{x_i \mapsto \bot_{a_i}\}_{i\in I}
\end{aligned}
$$

Fig. 13. Least approximation

(2) $a_1 \le a_1 \sqcup a_2$ *and* $a_2 \le a_1 \sqcup a_2$

(3) for all $a' \prec a$, $a_1 \le a' \wedge a_2 \le a' \implies a_1 \sqcup a_2 \le a'$

An element $a : [\![A]\!]_{\text{eval}}$ has a *least approximation* $\bot_a : [\![A]\!]_{\text{approx}}$. This is simply $\bot$ when A is of the form TB, but otherwise $\bot$ is not an element of $[\![A]\!]_{\text{approx}}$, and $\bot_a$ must be the head constructor of a applied to least approximations of its fields. The least approximation is defined formally in Figure 13, also extended to environments $\{x_i \mapsto a_i\}_i : [\![\Gamma]\!]_{\text{approx}}$. As its name implies, it is smaller than all other approximations of a.

Lemma 3.3 (bottom).

$$a' \prec a \implies \bot_a \le a'$$

Backwards evaluation. Given an input $g : [\![\Gamma]\!]_{\text{eval}}$ and an approximation a of the output ($a \prec [\![M]\!]_{\text{eval}}(g)$), the demand semantics $[\![M]\!]_{\text{dem}}(g, a) : \mathbb{N} \times [\![\Gamma]\!]^{\prec g}_{\text{approx}}$ describes the cost of *lazily evaluating* M *with demand* a, that is, evaluating M to a value a' "at least as defined as a," a relation which will be denoted by $a \le a'$. The resulting semantics is the cost of doing that evaluation, as well as the demand on the input, *i.e.,* the minimal approximation of the input g that is sufficient to match the output demand a. The demand semantics is defined in Figure 8.

3.2 Properties of demand semantics

We have seen that our semantics consists of a pair of forward and backward evaluation functions. Let us describe a few key properties of these functions.

For the rest of this section, let $\Gamma \vdash M : A$ be a well-typed term, $g \in [\![\Gamma]\!]_{\text{eval}}$, and $a = [\![M]\!]_{\text{eval}}(g)$.

Totality says that the demand semantics is defined for all approximations of outputs of the pure function $[\![M]\!]_{\text{eval}}$. This property is pictured as a commutative diagram in Figure 14, where the dotted

$$\begin{tikzcd}
g \arrow[r, "\llbracket M \rrbracket_{\text{eval}}"] & a \\
g' \arrow[u, dotted, "\prec"] & a' \arrow[l, dotted, "\llbracket M \rrbracket_{\text{dem}}(g, \cdot)"'] \arrow[u, "\prec"]
\end{tikzcd}$$

Fig. 14. Diagram of Lemma 3.4 (totality). Full arrows are quantified universally. Dotted arrows are quantified existentially.

.

arrows are existentially quantified. Let a be an output of $\llbracket M \rrbracket_{\text{eval}}$ (in Figure 14, this is represented by the arrow $g \xrightarrow{\llbracket M \rrbracket_{\text{eval}}} a$, meaning that $a = \llbracket M \rrbracket_{\text{eval}}(g)$). Given an approximation a' of the output ($a' \leq a$) the demand semantics is defined on a' (*i.e.,* there exists an arrow $g' \xleftarrow{\llbracket M \rrbracket_{\text{dem}}(g, \cdot)} a'$, meaning that $(n, g') = \llbracket M \rrbracket_{\text{dem}}(g, a')$ for some n) yielding an approximation of the input ($g' \leq g$). As the demand semantics is a partial function, we write $\exists(n, g') = \llbracket M \rrbracket_{\text{dem}}(g, a')$ to assert that $\llbracket M \rrbracket_{\text{dem}}(g, a')$ is defined. Lemma 3.4 expresses this property formally.

We abuse notation slightly, writing $\exists(n, g') = \llbracket M \rrbracket_{\text{dem}}(g, a') \wedge P$ as shorthand for $\exists(n, g'), (n, g') = \llbracket M \rrbracket_{\text{dem}}(g, a') \wedge P$, meaning that $\llbracket M \rrbracket_{\text{dem}}(g, a')$ is defined and its value (n, g') satisfies the proposition P.

Lemma 3.4 (Totality). *Let $g \in \llbracket \Gamma \rrbracket_{\text{eval}}$ and $a' \in \llbracket A \rrbracket_{\text{approx}}$ such that $a' \prec \llbracket M \rrbracket_{\text{eval}}(g)$.*

$$\exists(n, g') = \llbracket M \rrbracket_{\text{dem}}(g, a') \wedge g' \prec g$$

In Figure 8, $\llbracket M \rrbracket_{\text{dem}}$ was given a signature as a partial function. Knowing that $\llbracket M \rrbracket_{\text{dem}}$ satisfies that totality property, we can indeed view it as a total function with a type depending upon the first argument $g : \llbracket \Gamma \rrbracket_{\text{eval}}$:

$$\llbracket \Gamma \vdash M : A \rrbracket_{\text{dem}} : (g : \llbracket \Gamma \rrbracket_{\text{eval}}) \times \llbracket A \rrbracket_{\text{approx}}^{\prec \llbracket M \rrbracket_{\text{eval}}(g)} \rightarrow \mathbb{N} \times \llbracket \Gamma \rrbracket_{\text{approx}}^{\prec g}$$

The demand semantics is monotone: the more output is demanded from M, the more input it demands, and the higher cost it takes to produce the demanded output.

Lemma 3.5 (monotonicity). *Let $g \in \llbracket \Gamma \rrbracket_{\text{eval}}$ and $a_1, a_2 \in \llbracket A \rrbracket_{\text{approx}}$ such that $a_1 \leq a_2 \prec \llbracket M \rrbracket_{\text{eval}}(g)$.*

$$\llbracket M \rrbracket_{\text{dem}}(g, a_1) \leq \llbracket M \rrbracket_{\text{dem}}(g, a_2)$$

where $(n_1, g_1) \leq (n_2, g_2) \overset{\text{def}}{\iff} n_1 \leq n_2 \wedge g_1 \leq g_2$.

The demand semantics almost commutes with join ($\sqcup$). The input demand for a union of output demands is the union of their individual input demands. However, the cost may be less than the sum: the shared parts of the output demands only need to be evaluated once.

Lemma 3.6 ($\sqcup$-homomorphism). *Let $g \in \llbracket \Gamma \rrbracket_{\text{eval}}$ and $a_1, a_2 \prec \llbracket M \rrbracket_{\text{eval}}(g)$.*

$$\llbracket M \rrbracket_{\text{dem}}(g, a_1 \sqcup a_2) \leqq \llbracket M \rrbracket_{\text{dem}}(g, a_1) \uplus \llbracket M \rrbracket_{\text{dem}}(g, a_2)$$

where $(n_1, g_1) \uplus (n_2, g_2) \overset{\text{def}}{=} (n_1 + n_2, g_1 \sqcup g_2)$ and $(n_1, g_1) \leqq (n_2, g_2) \overset{\text{def}}{\iff} n_1 \leq n_2 \wedge g_1 = g_2$.

3.3 Correctness: Correspondence with Clairvoyant Semantics

Our framework relies on interpreting lazy programs as demand functions. Bjerner and Holmström [1989] introduced demand semantics in an untyped setting, but they did not prove a correspondence with any other semantics. Here, we formally relate that semantics to *clairvoyant semantics* [Hackett

$$\begin{tikzcd}
g \arrow[r, "\llbracket M \rrbracket_{\mathrm{eval}}"] & a \\
g' \arrow[u, "\prec"] \arrow[r, "\llbracket M \rrbracket_{\mathrm{cv}}"] & a' \arrow[u, dotted, "\prec"']
\end{tikzcd}$$

(a) Theorem 3.7 (functional correctness)

$$\begin{tikzcd}
g \arrow[r, "\llbracket M \rrbracket_{\mathrm{eval}}"] & a \\
g_2 \arrow[u, "\prec"] \arrow[r, dotted, "\llbracket M \rrbracket_{\mathrm{cv}}"] & (n_2, a_2) \arrow[u, dotted, "a_2 \prec a"] \\
(n, g_1) \arrow[u, "g_1 \leq g_2"] \arrow[ur, dash, dotted, "n = n_2"] & a_1 \arrow[u, dotted, "a_1 \leq a_2"] \arrow[l, "\llbracket M \rrbracket_{\mathrm{dem}}(g, \cdot)"] \arrow[uu, bend right=60, "\prec"']
\end{tikzcd}$$

(b) Theorem 3.8 (cost existence)

$$\begin{tikzcd}
g \arrow[r, "\llbracket M \rrbracket_{\mathrm{eval}}"] & a \\
g_2 \arrow[u, "\prec"] \arrow[r, "\llbracket M \rrbracket_{\mathrm{cv}}"] & (n_2, a_2) \arrow[u, dotted, "a_2 \prec a"] \\
(n, g_1) \arrow[u, dotted, "g_1 \leq g_2"] \arrow[ur, dotted, "n \leq n_2"] & a_1 \arrow[u, "a_1 \leq a_2"] \arrow[l, "\llbracket M \rrbracket_{\mathrm{dem}}(g, \cdot)"] \arrow[uu, bend right=60, "\prec"']
\end{tikzcd}$$

(c) Theorem 3.9 (cost minimality)

Fig. 15. Diagrams of correctness theorems between demand semantics and clairvoyant semantics

and Hutton 2019; Li et al. 2021], which was previously related to the (operational) *natural semantics* of laziness [Launchbury 1993] by Hackett and Hutton.

The monadic clairvoyant semantics of Li et al. [2021] is denoted $\llbracket \Gamma \vdash M : A \rrbracket_{\mathrm{cv}} : \llbracket \Gamma \rrbracket_{\mathrm{approx}} \to \mathcal{P}(\mathbb{N} \times \llbracket A \rrbracket_{\mathrm{approx}})$. Note a minor difference from our calculus to the one in Li et al. [2021]: the present calculus is essentially a call-by-value calculus with an explicit thunk type, whereas Li et al. [2021] defines a calculus with laziness by default, introducing thunks in the denotation of types. Intuitively, the clairvoyant semantics of Li et al. [2021] can be decomposed into an elaboration to the present calculus, followed by its clairvoyant semantics $\llbracket M \rrbracket_{\mathrm{cv}}$. The calculus presented here corresponds more closely to the core combinators of the clairvoyance monad in Li et al. [2021].

Functional correctness says that the clairvoyant semantics $\llbracket M \rrbracket_{\mathrm{cv}}$ approximates the pure function $\llbracket M \rrbracket_{\mathrm{eval}}$. This theorem is pictured as a commutative diagram in Figure 15a. Let g be an environment, let $a = \llbracket M \rrbracket_{\mathrm{eval}}(g)$ (diagrammatically: $g \xrightarrow{\llbracket M \rrbracket_{\mathrm{eval}}} a$), let $g' \prec g$, and let a' be a nondeterministic output of $\llbracket M \rrbracket_{\mathrm{cv}}(g')$ (diagrammatically: $g' \xrightarrow{\llbracket M \rrbracket_{\mathrm{cv}}} a'$; formally: $(n, a') \in \llbracket M \rrbracket_{\mathrm{cv}}(g')$ for some n). Then a' approximates a.

Theorem 3.7 (Functional correctness). *Let $g \in \llbracket \Gamma \rrbracket_{\mathrm{eval}}$, and $g' \prec g$.*

$$\forall (n, a') \in \llbracket M \rrbracket_{\mathrm{cv}}(g'),\ a' \prec \llbracket M \rrbracket_{\mathrm{eval}}(g)$$

The demand semantics $\llbracket M \rrbracket_{\mathrm{dem}}(g, a_1)$ finds a minimal pair (n, g') such that to produce a result at least as defined as the output demand a_1, the clairvoyant semantics must be applied to an input at least as defined as g' and the associated cost will be at least n. The minimality of (n, g') can be formalized as the conjunction of an existence property (the minimal cost is achievable) and a universality property (all other candidate executions have a higher cost). Those theorems are illustrated diagrammatically in Figure 15b and Figure 15c.

Theorem 3.8 (Cost existence). *Let $g \in \llbracket \Gamma \rrbracket_{\mathrm{eval}}$, $a_1 \prec \llbracket M \rrbracket_{\mathrm{eval}}(g)$, let $(n, g_1) = \llbracket M \rrbracket_{\mathrm{dem}}(g, a_1)$, and let g_2 such that $g_1 \leq g_2 \prec g$.*

$$g_1 \leq g_2 \implies \exists (n_2, a_2) \in \llbracket M \rrbracket_{\mathrm{cv}}(g_2),\ n = n_2 \wedge a_1 \leq a_2$$

Theorem 3.9 (Cost minimality). *Let* $g \in [\![\Gamma]\!]_{\text{eval}}$, $a_1 \prec [\![M]\!]_{\text{eval}}(g)$, *let* $(n, g_1) = [\![M]\!]_{\text{dem}}(g, a_1)$, *and let* $g_2 \prec g$.

$$\forall (n_2, a_2) \in [\![M]\!]_{\text{cv}}(g_2),\ a_1 \le a_2 \implies n \le n_2 \wedge g_1 \le g_2$$

3.4 Deriving the definition of $[\![M]\!]_{\text{dem}}$

We can use the properties above to derive the definition of $[\![M]\!]_{\text{dem}}$ by inequational reasoning. For example, to find the value of $[\![\text{force}\, M]\!]_{\text{dem}}$ as a function of $[\![M]\!]_{\text{dem}}$, totality requires $[\![\text{force}\, M]\!]_{\text{dem}}(g, a') \prec g$, given $a' \prec_A [\![\text{force}\, M]\!]_{\text{eval}}(g) = [\![M]\!]_{\text{eval}}(g)$, and given the totality property for M, $[\![M]\!]_{\text{dem}}(g, a'') \le g$ for all $a'' \prec_{T\,A} a$. With $a'' = \textbf{thunk}\, a'$, we have $[\![M]\!]_{\text{dem}}(g, \textbf{thunk}\, a') \le g$. This suggests the definition $[\![\text{force}\, M]\!]_{\text{dem}}(g, a') = [\![M]\!]_{\text{dem}}(g, \textbf{thunk}\, a')$.

In a similar way, we can derive the tricky-looking definition of $[\![\text{foldr}\, M_1\, M_2\, N]\!]_{\text{dem}}$ by inequational reasoning.

Note that the minimality property forbids the trivial definition $[\![M]\!]_{\text{dem}}(g, a) = (0, \perp_g)$.

4 CASE STUDIES: SORTING ALGORITHMS

In this section, we analyze insertion sort and selection sort using our bidirectional demand semantics to demonstrate how our model can be used in practice. These algorithms are known to exhibit $O(n^2)$ time complexity under eager evaluation. However, we can achieve $O(k \cdot n)$ complexity under lazy evaluation if we only need the smallest k elements of a list. More formally, given the following functions:

```
Definition p1 (k : nat) (xs : list nat) := take k (insertion_sort xs).
Definition p2 (k : nat) (xs : list nat) := take k (selection_sort xs (length xs)).
```

We will prove that the computation cost of both `p1 k xs` and `p2 k xs` are bounded by $O(k \cdot n)$ where $n = |\texttt{xs}|$. To do this, we use demand semantics.

All the lemmas and theorems we show in this section have been formally proven in the Rocq Prover. This code can be found in our supplementary materials.

4.1 The take Function

We show the definitions of `take` and `takeD` in Fig. 16. The `take` function is defined recursively, which our calculus does not directly support. However, we can apply the universal property of fold [Hutton 1999] to see that `take` is actually an instance of foldr over the list t = `zip [k; k - 1; k - 2; ...; 1] xs` (note the lack of 0 in the first list). We can then treat the `take` function as foldr $(\lambda\, (n, y)\, zs.\, \textbf{cons}\, y\, zs)\, \textbf{nil}\, t$. If we apply $[\![\,\cdot\,]\!]_{\text{dem}}(g, d)$ (Fig. 8) to it, we obtain t as well as the output demand d (Fig. 9).

$$[\![\text{foldr}\ (\lambda(n, y)\, zs.\, \textbf{cons}\, y\, zs)\ \textbf{nil}\ t]\!]_{\text{dem}}(g, d) = (c, g') \uplus [\![t]\!]_{\text{dem}}(g, n')$$
$$\text{where } (c, g', n') = \textbf{foldr}_{\text{dem}}(g, \textbf{cons}\, y\, zs, \textbf{nil}, \textbf{thunk}\ ([\![t]\!]_{\text{eval}}(g)), \textbf{thunk}\, d)$$

Because we are trying to define `takeD`, we would like to treat t as its arguments so that we only need to use the rule for evaluating variables to evaluate $[\![t]\!]_{\text{dem}}(g, n') = (0, \{t \mapsto n'\})$. However, even though we use the inductive `nat` data type in Gallina to represent numbers for simplicity, numbers are primitive data types in most programming languages. As such, we are not interested in an approximation of a `nat`, so we we make a simplification in `takeD` such that we only return the demand the argument `xs`. For this reason, we will ignore the numbers in t in the rest of the translation process.

The $\textbf{foldr}_{\text{dem}}$ function is a recursive function that supports pattern matching (Fig. 9). It lives in the universe of denotations that we would like to replace with Gallina definitions—in fact, this will be our definition of `takeD`. In the case that $t = \textbf{nil}$, we return $\textbf{thunk}\, \textbf{nil}$ (line 14, Fig. 16). In the case

```
1   Fixpoint take {a} (k : nat) (xs : list a) : list a :=
2     match k, xs with
3     | O, _ => nil
4     | S _, nil => nil
5     | S k', x :: xs' =>
6       let zs := take k' xs' in
7       x :: zs
8     end.
9
10  Fixpoint takeD {a} (n : nat) (xs : list a)
11                     (outD : listA a) : Tick (T (listA a)) :=
12    tick >> match n, xs, outD with
13    | 0, _, _ => ret Undefined
14    | _, [], _ => ret (Thunk NilA)
15    | S m, y :: ys, ConsA zD zsD =>
16      let+ ysD := thunkD (takeD m ys) zsD in
17      ret (Thunk (ConsA (Thunk y) ysD))
18    | _, _, _ => bottom (* absurdity case *)
19    end.
```

Fig. 16. The Gallina implementation of take and takeD.

that $t = \mathbf{cons}\,(n, y)\; ts$, there are two steps. First, we need to run the following:

$$\begin{aligned}(c_1, \{g_1, y \mapsto a_1', zs \mapsto b_2'\}) =& [\![M_1]\!]_{\text{dem}}(\{g, y \mapsto a_1, zs \mapsto \mathbf{foldr}_{\text{eval}}(g, M_1, M_2, a_2)\}, d)\\ =& [\![\mathbf{cons}\; y\; zs]\!]_{\text{dem}}(\{g, y \mapsto y,\\ & \quad zs \mapsto \mathbf{foldr}_{\text{eval}}(g, \mathbf{cons}\; y\; zs, \mathbf{thunk}\;\mathbf{nil}, zs)\}, d)\end{aligned}$$

However, this expression only makes sense when $d = \mathbf{cons}\; zD\; zsD$ according to Fig. 8, so we perform a pattern match on d as well (line 15). We will handle $d = \mathbf{nil}$ as one of the absurdity cases (line 18). If we continue running the demand semantics, we will see that this part evaluates to $(c_1, \{g_1, y \mapsto a_1', zs \mapsto b_2'\}) = (0, \{y \mapsto zD, zs \mapsto zsD\})$. After that, we need to run:

$$\begin{aligned}(c_2, g_2, a_2') =& \mathbf{foldr}_{\text{dem}}(g, M_1, M_2, a_2, b_2')\\ =& \mathbf{foldr}_{\text{dem}}(g, \mathbf{cons}\; y\; zs, \mathbf{nil}, zs, zsD)\end{aligned}$$

This is equivalent to a recursive call applied to zsD (line 16). According to the definition of $\mathbf{foldr}_{\text{dem}}$, we obtain $(c_1 + c_2, g_1 \sqcup g_2, \mathbf{thunk}\;(\mathbf{cons}\; a_1'\; a_2'))$. Finally, combining this result with the denotational semantics, we obtain the cost $c_1 + c_2$ and the input demand $\mathbf{thunk}\;(\mathbf{cons}\; a_1'\; a_2')$. As explained previously, we use the `Tick` data type to represent the tuple of computation cost and input demand. It is defined as a writer monad such that costs are added together in a monadic bind (the `let+` notation on line 16). Thanks to the use of monads, we only need to `ret` the input demand (line 17).

In addition, we manually add a `tick` in the beginning of the function (line 12) so that we can count the number of function calls. We also handle the case in which the output demand is $\bot$ and the case in which we apply $[\![\mathbf{cons}\; y\; zs]\!]_{\text{dem}}$ to a d that is not a **cons**. In these cases, we return `bottom`, which represents a cost of 0 and the minimal input demand, *i.e.,* `Undefined` in this case (line 18). Theoretically, we should also return `bottom` when `k = 0`, but we write `ret Undefined` instead (line 13) to make the function more structurally similar to `take`—note that `bottom` and `ret Undefined` mean the same thing in this function.

Now that we have defined `takeD`, we can use it to reason about the cost of `take`. We can state and prove the following the theorems:

```
Theorem takeD_cost : forall {A : Type} (n : nat) (xs : list A) outD,
    Tick.cost (takeD n xs outD) <= 1 + n.

Theorem takeD_cost' : forall {A : Type} (n : nat) (xs : list A) outD,
    Tick.cost (takeD n xs outD) <= sizeX' 1 outD.
```

Both theorems define aspects of the cost of a lazy `take`. The first theorem states that the cost is bounded by its parameter `n + 1`. The second theorem states that the cost is also bounded by the size of the output demand. The function `sizeX'` is defined such that `sizeX' 1 outD` is equivalent to $\max(1, |\texttt{outD}|)$.

4.2 Insertion Sort

We have shown the pure implementation of the insertion sort and its corresponding demand functions in Fig. 1 and Fig. 2, respectively. The process of translating `insert` is similar to translating `take` except for the need to translate an `if`-expression. Translating `insertion_sort`, however, is more challenging. Doing so involves translating a `let`-expression that contains two function calls. To translate the `let`-expression, we apply the following rule from our denotational semantics (Fig. 8):

$$\begin{aligned}
[\![\text{let}\, x = M \,\text{in}\, N]\!]_{\text{dem}}(g, a) &= (c_N + c_M, d_N \sqcup d_M) \\
\text{where}\, (c_N, \{d_N, x \mapsto b\}) &= [\![N]\!]_{\text{dem}}(\{g, x \mapsto [\![M]\!]_{\text{eval}}(g)\}, a) \\
\text{and}\, (c_M, d_M) &= [\![M]\!]_{\text{dem}}(g, b)
\end{aligned}$$

According to this, first we need to run `select` in the forward direction, then use its result to run `insertion_sort` in the backward direction (*i.e.,* `insertion_sortD`), and finally use the input demand from `insertion_sortD` to run `insert` in the backward direction (*i.e.,* `insertD`). This process corresponds to our illustration in Fig. 3, and it is also how we define `insertion_sortD` on lines 22–25 in Fig. 2.

We show the main theorems we have proven for insertion sort in Fig. 17. Proofs for these theorems are relatively straightforward. Theorems regarding `insertD` and `insertion_sortD` can all be proven by an induction over the list arguments `xs`. All the inequalities involved in these theorems can be solved by Rocq Prover's built-in tactics `lia` and `nia` [Besson and Makarov 2023].

In addition, we combine the theorems `takeD_cost` and `insertion_sortD_cost` to prove the theorem `take_insertion_sortD_cost` (Fig. 17). The function `take_insertion_sortD` is the demand function of `take_insertion_sort`, which composes `take` and `insertion_sort`. By proving this theorem, we formally prove that the cost of `take_insertion_sortD k xs outD` is bounded by $O(k \cdot n)$ where $n = |\texttt{xs}|$.

Theoretically, we can show a tighter bound in `take_insertion_sortD_cost`, as the list argument for `insertion_sort` decreases after each recursive call. However, the bound we show here is sufficient to show that the function has $O(k \cdot n)$ time complexity. Proving this tighter bound would not change the asymptotic cost.

4.3 Selection Sort

We have proven that the computation cost of `selection_sort` is bounded by the same time complexity as `insertion_sort`. We take the implementation of `select` and `selection_sort` from *Verified Functional Algorithms* [Appel et al. 2023], then manually translate them into demand functions.

```
(* Computation cost. *)
Lemma insertD_cost x (xs : list nat)  (outD : listA nat) :
  Tick.cost (insertD x xs outD) <= leb_count x xs + 1.
Lemma insertD_cost' x (xs : list nat) (outD : listA nat) :
  Tick.cost (insertD x xs outD) <= sizeX' 1 outD.
Lemma insertD_cost'' x (xs : list nat) (outD : listA nat) :
  Tick.cost (insertD x xs outD) <= length xs + 1.
Theorem insertion_sortD_cost (xs : list nat) (outD : listA nat) :
  Tick.cost (insertion_sortD xs outD) <= (sizeX' 1 outD + 1) * (length xs + 1).

(* Composition. *)
Theorem take_insertion_sortD_cost (n : nat) (xs : list nat) (outD : listA nat) :
  Tick.cost (take_insertion_sortD n xs outD) <= (n + 1) * (length xs + 2) + 1.
```

Fig. 17. Main theorems we have proven for insertion sort.

.

```
Definition select (x: nat) (l: list nat) : nat * list nat.
Definition selection_sort (l : list nat) (n : nat) : list nat.
Definition selectD (x : nat) (l : list nat)
  (outD : prodA nat (listA nat)) : Tick (T (listA nat)).
Definition selection_sortD (l : list nat) (n : nat)
  (outD : listA nat) : Tick (T (listA nat)).
```

Fig. 18. Type signatures of all definitions related to the selection sort.

For the sake of space, we only show the types of all relevant definitions in Fig. 18. In addition to the list argument `xs`, the `selection_sort` function takes an additional argument `n : nat`, which is our “fuel” for running selection sort. We use this fuel to convince the Gallina termination checker that `selection_sort` will terminate.[2] In practice, we always want to use a fuel size that is at least as large as the length of `xs`. The complete definitions and proofs for selection sort are included in our supplementary materials.

Compared to `insertion_sort`, `selection_sort` is more challenging because it returns a product type. Accordingly, we need to use `prodD A B = T A * T B`, an approximation of the product type, as the type of `outD` in `selectD`. This typing is crucial as, unlike `insert`, the cost of `select` is not bounded by the demand on the output list. In fact, even when the demand on the output list is `Undefined`, `select` still must traverse the entire input list to select the smallest element.

We show the main cost theorems we have proven for selection sort in Fig. 19. Compared to `insertD` in insertion sort, `selectD` is only bounded by `length xs + 1`, not `sizeX' 1 outD`, because the function always traverses the entire input argument `xs`. Nevertheless, we show a similar bound for `selection_sortD` as long as the fuel we use is greater than or equal to `length xs`. In the end, we can compose `takeD_cost` with `selection_sortD_cost` to prove the bounds stated in `take_selection_sortD_cost`.[3]

[2] It is possible to manually prove that `selection_sort` terminates without using this “fuel” construct. Instead, we demonstrate the fuel version here for simplicity, as it requires fewer proofs.
[3] The `take_selection_sort` function runs `selection_sort` with a fuel equal to `length xs`.

```
(* Computation cost. *)
Lemma selectD_cost : forall x xs yD ysD,
    Tick.cost (selectD x xs (pairA yD ysD)) <= length xs + 1.
Lemma selection_sortD_cost (xs : list nat) (n : nat) (outD : listA nat) :
  n >= length xs ->
  Tick.cost (selection_sortD xs n outD) <= (sizeX' 1 outD) * (length xs + 1).

(* Composition. *)
Theorem take_selection_sortD_cost (n : nat) (xs : list nat) (outD : listA nat) :
  Tick.cost (take_selection_sortD n xs outD) <= n * (length xs + 2) + 1.
```

Fig. 19. Main cost theorems we have proven for selection sort.

.

5 AMORTIZED AND PERSISTENT DATA STRUCTURES

In this work, we apply our method to mechanically reason about two *lazy*, *amortized*, and *persistent* data structures, Okasaki's banker's queue and implicit queue [Okasaki 1999]. Both data structures implement first-in-first-out (FIFO) queues with amortized constant time operations. The banker's queue achieves amortization and persistence by maintaining a balancing invariant on two lists. The implicit queue achieves both properties using a technique called "implicit recursive slowdown" [Kaplan and Tarjan 1995; Okasaki 1999].

Our amortized and persistent analysis is based on a novel *reverse physicist's method* that is designed for our demand semantics (Section 5.2). We manually derive the demand functions and prove that both the banker's queue and the push function of the implicit queue are amortized and persistent using the Rocq Prover (Section 5.3–5.4). The demand functions for these data structures (especially the implicit queue) are more complex than what was shown in Section 4, so in our mechanized proofs, we do *not* trust our demand functions. Instead, we validate that our demand functions are correct by corresponding them with an alternative and more mechanical semantics called the clairvoyant semantics [Li et al. 2021]. All the definitions and proofs can be found in our supplementary materials.

5.1 Banker's Queue

We show a Gallina implementation of the banker's queue in Fig. 20. We first declare a record Queue whose internal representation is two lists and two numbers: a front list and a back list, with two numbers nfront and nback that keep track of the lengths of these two lists, respectively (lines 1–8). When pushing an element to the queue, we add it to the head of the back list (lines 15–16). For the sake of space, we omit the definitions of empty and pop. We then use a smart constructor mkQueue to maintain the "balance" of the queue, by reversing and moving the back list to the end of the front list when front is longer than back (lines 10–13). All functions of the banker's queue maintain the invariant that nfront and nback represent the lengths of the front and back lists.

Amortization and persistence: an informal analysis. When we update a data structure in a functional program, we produce a new version of the data structure while keeping the original one, a feature otherwise known as "persistence". This results in multiple versions of a data structure coexisting at the same time. These versions must be utilized carefully, as using the same version multiple times can break amortization.

```
1   Variable Elt : Type.
2
3   Record Queue : Type := MkQueue
4     { nfront : nat
5     ; front : list Elt
6     ; nback : nat
7     ; back : list Elt
8     }.
9
10  Definition mkQueue
11    (nf : nat) (f : list Elt) (nb : nat) (b : list Elt) : Queue :=
12    if nf <? nb then MkQueue (nf + nb) (append f (rev b)) 0 []
13    else MkQueue nf f nb b.
14
15  Definition push (q : Queue) (x : Elt) : Queue :=
16    mkQueue (nfront q) (front q) (nback q + 1) (x :: back q).
```

Fig. 20. The banker's queue [Okasaki 1999], implemented in Gallina. The queue has an amortized cost of $O(1)$ for both push and pop operations.

To intuitively understand why the banker's queue is persistent, let's work through an example. Assume that we have a queue `q0` that has a front list of `[1, 2, 3]`, a back list of `[6, 5, 4]`, and therefore its `nfront` and `nback` are both `3`. For simplicity, let's assume that everything in `q0` has already been evaluated, *i.e.,* there is no unevaluated thunk. Consider the following program:

```
(q1, e1) <- pop q0 ;;     (* front q1 = [2,3] ++ rev [6,5,4], e1 = 1 *)
(q2, e2) <- pop q1 ;;     (* front q2 =   [3] ++ rev [6,5,4], e2 = 2 *)
(q3, e3) <- pop q2 ;;     (* front q3 =    [] ++ rev [6,5,4], e3 = 3 *)
(q4, e4) <- pop q3 ;;     (* front q4 = [5,6] ++ [],          e4 = 4 *)
(* ... more operations *)
```

In the first line, `1` is removed from `q0`'s front list. We use a custom notation that is similar to Haskell's **do**-notation to represent monadic binds on the `option` type. The `pop` operation on `q0` causes `nfront` to be less than `nback` in `q1` and triggers a reversal of the back list (line 10 in Fig. 20). Due to lazy evaluation, however, the reversal does not happen immediately—it is stored as a thunk instead. The thunk will only be forced at `q4`, and every element in the queue will be visited at most twice. So, the cost of the reversal is amortized across `pop`s.

Now suppose that we want to use a copy of the queue multiple times. Let's consider the "worst" case, *i.e.,* when the reversal is forced:

```
(_, r1) <- pop q3 ;;    (* r1 = 4 *)
(_, r2) <- pop q3 ;;    (* r2 = 4 *)
(_, r3) <- pop q3 ;;    (* r3 = 4 *)
(* ... more operations *)
```

In this program, we call `pop` on `q3` multiple times. Each time, we expect `4` from the queue's "original" back list to be popped. Even though we are accessing an element from the reverse of the back list, all these operations cost constant time because the thunk containing the reversal computation stored in `q3` was forced when we called `pop q3` the first time.

```
let q0 = empty in
let q1 = push q0 a in    (* D q2 = (a:nil,b:bot)  *)
let q2 = push q1 b in    (* D q2 = (a:nil,b:bot)  *)
let q3 = push q2 c in    (* D q2 = (bot, bot)     D q4 = (a:b:bot,bot) *)
let q4 = push q3 d in    (* D q2 = (bot, bot)     D q4 = (a:b:bot,bot) *)
(_, q5) <- pop q4 ;;     (* D q2 = (bot, bot)     D q4 = (a:bot,bot)   *)
(_, q6) <- pop q5 ;;     (* D q2 = (bot, bot)     D q4 = (a:bot,bot)   *)
(_, q7) <- pop q4 ;;     (* D q2 = (bot, bot)     D q4 = (bot,bot)     *)
()
```

Fig. 21. A program that uses the banker's queue, with demands of q2 and q4 labeled at each step.

5.2 The Reverse Physicist's Method

The banker's method and the physicist's method are the two classical methods for analyzing amortized computation costs [Tarjan 1985]. However, these methods only work for "forward" and strict semantics, where we *first* accumulate credits (in the banker's method) or potential (in the physicist's method) before making an "expensive" operation to spend the accumulation. Our demand semantics works differently: we compute a minimal input demand from an output demand, working "backwards". Therefore, we propose a new method, called the *reverse physicist's method*, to analyze amortized computation cost based on this semantics.

The key idea of the reverse physicist's method is to consider the demand semantics as an evaluation on approximations that happen "backward". Under this view, future operations are accumulating potential that is going to be used by expensive operations that happen earlier. Therefore, our solution is to assign *potential* to demands. Taking the banker's queue as an example, we assign a potential Φ to each demand of a queue q^D as:

$$\Phi(q^D) = \max(2 \times (|f^D| - |b|), 0) \tag{1}$$

f^D represents the demand of the front list (*i.e.*, an approximation of the front list) and b represents the back list. We overload the $|\cdot|$ operator to represent the length of either a list or a demand of a list, so $|f^D|$ and $|b|$ represent the lengths of f^D and b, respectively. Note that $|f^D| \leq |f|$.

To show that the banker's queue has amortized constant cost, it suffices to show that for push and pop, the cost of each operation satisfies the following inequality:

$$cost \leq \Phi(q^D_{out}) - \Phi(q^D_{in}) + const \tag{2}$$

$\Phi(q^D_{in})$ and $\Phi(q^D_{out})$ represent the potential for the input queue and the output queue of the operation, respectively. The *cost* is bounded by the difference between the input demand's potential and the output demand's potential plus a constant number (*const*).

Amortization. Suppose that we have a program with a banker's queue that begins empty, and then is acted on by push and pop operations. By doing an analysis with our demand semantics, we know the minimal input demand for each operation. We call the initial empty queue demand q^D_0 and the queue demand produced by the i-th operation q^D_i. By showing that the above inequality holds for every operation on the banker's queue, we show that:

$$\begin{aligned}
cost_1 &\leq \Phi(q_1^D) - \Phi(q_0^D) + const \\
cost_2 &\leq \Phi(q_2^D) - \Phi(q_1^D) + const \\
&\cdots \\
cost_n &\leq \Phi(q_n^D) - \Phi(q_{n-1}^D) + const
\end{aligned}$$

Combining all these inequalities, we get:

$$\sum_{i=1}^{n} cost_i \leq \Phi(q_n^D) - \Phi(q_0^D) + n \cdot const$$

Because we start with an empty queue and there will be no more demands on the queue after the program ends, we know that $\Phi(q_0^D) = \Phi(q_n^D) = 0$, which means:

$$\sum_{i=1}^{n} cost_i \leq n \cdot const$$

Therefore, our model shows that the banker's queue has an amortized constant time cost.

Persistence. To show that the banker's queue is persistent, we consider the demand for all queues that are generated in the program at certain point. We use $q_{i@j}^D$ to denote the demand of q_i^D at the j-th operation (both indices start from 0). We constrain j to be at least as large as i for any $q_{i@j}^D$.

Taking the program shown in Fig. 21 as an example. We start from $q_{i@j}^D = (\bot, \bot)$ for any $j \geq 8$, because there is no demand after the program has finished. Using the initial list of $q_{i@8}^D$ for all is as the output demand, we can compute the following minimal input demands for every queue at each step using our demand semantics (Section 3). We obtain the following demands:

$$q_{7@j}^D = (\bot, \bot) \qquad q_{6@j}^D = (\bot, \bot)$$

$$q_{5@j}^D = \begin{cases} (b : \bot, \bot) & \text{if } j \leq 6, \\ (\bot, \bot) & \text{otherwise} \end{cases} \qquad q_{4@j}^D = \begin{cases} (a : b : \bot, \bot) & \text{if } j \leq 5, \\ (a : \bot, \bot) & \text{if } 5 < j \leq 7, \\ (\bot, \bot) & \text{otherwise} \end{cases}$$

$$q_{3@j}^D = \begin{cases} (a : b : \bot, \bot) & \text{if } j \leq 4, \\ (\bot, \bot) & \text{otherwise} \end{cases} \qquad q_{2@j}^D = \begin{cases} (a : \mathbf{nil}, b : \bot) & \text{if } j \leq 3, \\ (\bot, \bot) & \text{otherwise} \end{cases}$$

$$q_{1@j}^D = \begin{cases} (a : \mathbf{nil}, \bot) & \text{if } j \leq 2 \\ (\bot, \bot) & \text{otherwise} \end{cases} \qquad q_{0@j}^D = (\mathbf{nil}, \mathbf{nil})$$

In addition, we generalize Inequality (2) as follows:

$$cost_{[i,j]} \leq \sum_{k=0}^{j} \Phi(q_{k@j}^D) - \sum_{k=0}^{i} \Phi(q_{k@i}^D) + (j - i) \cdot const \tag{3}$$

First, we generalize *cost* in Inequality (2) to $cost_{[i,j]}$, which represents the computation cost incurred from the i-th operation to the j-th operation. When $j = i+1$, $cost_{[i,j]}$ is the cost for a single operation. Second, instead of computing the difference between the potential of an input demand and that of an output demand, we compute the difference between *the sum* of all the potential of all queue demands at step j and that at step i.

If we suppose that our program has t operations in total, then the cost of the entire program is $cost_{[0,t]}$. At the beginning of a program, the empty queue is the only possible queue, and its

```
Record QueueA (a : Type) : Type := MkQueueA
  { nfrontA : nat
  ; frontA : T (listA a)
  ; nbackA : nat
  ; backA : T (listA a)
  }.

Definition pushD {a} (q : Queue a) (x : a)
                     (outD : QueueA a) : Tick (T (QueueA a) * T a) :=
  tick >> let+ (frontD, backD) := mkQueueD (nfront q) (front q)
                                   (S (nback q)) (x :: back q) outD in
  ret (Thunk (MkQueueA (nfront q) frontD (nback q) (tailX backD)), Thunk x).
```

Fig. 22. Monadic and demand translations of the banker's queue's push function.

demand is $q^D_{0@0} = \bot$ and $\Phi(q^D_{0@0}) = 0$. At the end of a program, there cannot be any more demands, so $q^D_{i@t} = \bot$ for all $0 \leq i < t$ and $\sum_{k=0}^{t-1} \Phi(q^D_{k@t}) = 0$. Therefore, if we can show that Eq. (3) is true for all i and j in one program trace, we can conclude that the cost of the entire program is bounded by constant cost multiplied by the number of queue operations.

To show that the banker's queue is *persistent*, it suffices to show that for any program trace that uses the banker's queue, the cost of the entire program is bounded by constant cost multiplied by the number of queue operations. Note that we do not put any constraints on the program trace—a queue can be reused for any number of times in a program trace.

5.3 Banker's Queue in the Rocq Prover

We manually derive all the demand functions of the banker's queue. For the sake of space, we only show the approximation data type `QueueA` and the demand function `pushD` in Fig. 22. The translation is mostly based on our demand semantics, but we made certain simplifications—we discuss how we validated our manual translation at the end of this section.

We show the cost theorems we have proven for the banker's queue in Fig. 23. These theorems are defined by following the reverse physicist's method (Section 5). We define the potential (lines 1–2) of the queue to be twice the length of the *demand* of the queue's front list's minus the length of its back list, as described in Eq. (1). The function `sizeX` measures the "length" of a demand list. It is parameterized by a natural number which represents how long we consider the length of `NilA` — *i.e.,* `sizeX 0` means that a demand list of `NilA` has a length of `0`. We do need a max function to make sure that the potential is at least 0, because the type of the `potential` function specifies that the potential must be a natural number.

We then define the cost specification of `pushD` as an inequality between the potential of its input demand (`qInD`) plus the cost (`cost`) and the potential of its output demand (`qOutD`) plus a constant (`const`) (lines 6–9). This is the same as Inequality (2) but we change the inequality to only include the `+` operation so that we only need to use natural numbers in the specification. We also prove a similar theorem for `popD`.

However, we do not wish to trust our demand semantics, as functions in the banker's queue are more complicated than insertion sort or selection sort. To show that our mechanized cost analysis is correct, we additionally translate the banker's queue using another model of laziness, namely the clairvoyant semantics [Hackett and Hutton 2019; Li et al. 2021]. The advantage of a clairvoyant semantics is that it translation from pure functions is more mechanical: it can be done by adding

```
Definition potential (q: QueueA) : nat :=
  2 * (sizeX 0 (frontD q) - nbackD q).

Definition const : nat := 7.

Theorem pushD_cost : forall q x qOutD,
  well_formed q ->
  qOutD `is_approx` push q x ->
  let (cost, qInD) := pushD q x qOutD in
  potential qInD + cost <= potential qOutD + const.
```

Fig. 23. Cost specification for push.

```
 1  Inductive Queue (A : Type) : Type :=
 2  | Nil : Queue A
 3  | Deep : Front A -> Queue (A * A) -> Rear A -> Queue A.
 4
 5  Fixpoint push (A : Type) (q : Queue A) (x : A) : Queue A :=
 6    match q with
 7    | Nil => Deep (FOne x) Nil RZero
 8    | Deep f m RZero => Deep f m (ROne x)
 9    | Deep f m (ROne y) => Deep f (push m (y, x)) RZero
10    end.
```

Fig. 24. The Gallina implementation of the implicit queue. Compared to the implementation presented by Okasaki [1999], we slightly simplify the base case to make it only represent an empty queue. This does not affect the computation cost of the implicit queue.

the right combinators to the pure function. We show that the demand functions of the banker's queue agree with the clairvoyant translation on computation cost.

Finally, to show that the reverse physicist's method implies amortization and persistence, we show that the cost of running an arbitrary trace that manipulates the banker's queue using given methods is bounded by a constant factor, by applying theorem `pushD_cost`.

5.4 Implicit Queue in the Rocq Prover

The implicit queue is another persistent data structure that exhibits amortized constant computation cost shown by Okasaki. We show some key Gallina definitions of the implicit queue in Fig. 24.

An implicit queue of type `A` is a inductive data type that is either an empty queue `Nil`(line 10) or a "deep" structure that contains: (1) a `Front`, which contains one or two `A`s (lines 1–3); (2) a `Rear`, which contains zero or one `A` (lines 5–7); (3) an inner queue of a product `A * A` (line 11).

When `push`ing a new element to the list, we first perform a pattern match on the input queue `q` (line 14). If `q` is empty, we add the new element to `Front` directly (line 15). If `q` is `Deep` structure whose `Rear` contains zero elements, we put the new element in the `Rear` (line 16). If `q` is a `Deep` structure that already has an element `y` in the `Rear`, we make a *polymorphic recursive call* to `push` the product of `y` and the new element to the inner queue (line 17). The use of polymorphic recursion is important for the efficiency of the implicit queue—the technique is known as *recursive slowdown*.

We prove similar cost theorems for `pushD` and `popD` of the implicit queue based on the reverse physicist's method, despite the differences between implicit queue and banker's queue. We also show that the demand functions `pushD` and `popD` agree with a clairvoyant translation of `push` and `pop` on computation cost.

6 RELATED WORK

Verification of lazy functional programs. Analyzing computation cost of lazy languages usually requires modeling and reasoning about mutable heaps [Launchbury 1993], which is challenging. Prior works on formalized cost analysis of lazy programs can be divided into three categories. The first approach uses a tick monad [Danielsson 2008] to model computation cost in a functional way. To model sharing, a `pay` combinator needs to be manually inserted at proper places to preemptively force the computation of the shared part of a data structure. Liquid Haskell [Handley et al. 2020; Vazou 2016] uses a similar approach.

Alternatively, one can reason about mutable heaps using separation logics. This approach is taken by Pottier et al. [2024] to verify an OCaml implementation of the lazy banker's queue, the physicist's queue, and implicit queues in Coq based on the Iris$^{\$}$ framework [Mével et al. 2019]. Pottier et al.'s method is based on an imperative style, as their code directly encodes thunks in OCaml and their reasoning is based on the Iris separation logic [Jung et al. 2018; Spies et al. 2022].

Instead of dealing with the complexity caused by mutable heaps directly, the third approach uses alternative semantics models instead of natural semantics [Launchbury 1993]. Prior works on *clairvoyance semantics* fall in this category [Hackett and Hutton 2019; Li et al. 2021]. The key idea of the clairvoyance semantics is simulating laziness using a nondeterministic *call-by-value* model. Our work is based on Bjerner and Holmström [1989]'s demand semantics, which is untyped and relies on partial functions, whereas our version is typed, allowing the semantics to be defined in terms of total functions, making the semantics simpler to formalize and use in a proof assistant based on type theory such as Coq. We have also formally proved a correspondence between the demand semantics and a preexisting model of laziness, namely the clairvoyant semantics, which prior work [Hackett and Hutton 2019; Li et al. 2021] has connected to the natural semantics of Launchbury [1993]. Compared with the clairvoyant semantics, demand semantics allows us to sidestep the need for both nondeterminism and the need for incorrectness logic [Li et al. 2021], by including output demands as parts of input for the demand functions.

One drawback of our method is that, due to changing "when" a computation happens in our model, our method cannot be used to analyze real-time computation cost, nor can it be used to analyze other types of resource that are not monotone, *e.g.*, memory usage. In comparison, Liquid Haskell can be used on measuring the usage of "any kind of resource whose usage is additive" [Handley et al. 2020]. Iris$^{\$}$ has been used on verifying the implicit queue that has real-time constant computation cost.

On the testing side, Foner et al. [2018] introduce a low level Haskell testing library `StrictCheck` using a similar idea of demand-driven analysis. They focus on testing for strictness bugs in lazy programs, utilizing persistent and non-persistent queues introduced by Okasaki as examples to verify their implementation, much as we have.

Demand-driven program analysis and symbolic execution. Demand-driven analysis is also a useful technique that has been used in data-flow analysis, control-flow analysis, and symbolic execution [Dubé and Feeley 2002; Facchinetti et al. 2019; Germane et al. 2019; Horwitz et al. 1995; Palmer et al. 2020]. The key idea is by starting from the target to be analyzed, unneeded analysis/evaluation can be avoided. Our demand semantics is based on similar idea, as it avoids nondeterminism present in clairvoyant semantics by having output demand as parts of demand function's input. However, our demand semantics focuses on reasoning about computation cost for lazy functional programs and lazy, amortized, and persistent data structures. Facchinetti et al. [2019]; Germane et al. [2019]; Palmer et al. [2020] also support program analysis and symbolic execution for *higher-order* functions, which our demand semantics does not currently support.

Demand analysis in compilers. Compilers like the Glasgow Haskell Compiler (GHC) employs demand analysis to perform compiler optimizations [Sergey et al. 2014, 2017]. However, demand analyses in compilers typically focus on finding one-shot lambdas, single-entry thunks, *etc.* to apply optimizations such as deforestation. Our demand semantics focuses on compositional mechanized reasoning for proving properties about computation cost.

Amortized analysis for non-lazy semantics. Amortized cost analysis in the context of strict semantics has been the subject of much research as well [Cutler et al. 2020; Hoffmann et al. 2012; Rajani et al. 2021]. Cutler et al. [2020] provide a formalized framework for reasoning about amortized analysis in an imperative setting. Rajani et al. [2021] introduce a type system which can embed call-by-value and call-by-name evaluation as well as accounting for cost savings via amortization. Hoffmann et al. [2012] improve this type of analysis by introducing arbitrary multivariate polynomial functions to their cost representations, including comparisons of theoretical bounds to real-world examples. All of these models allow for space-usage analysis, but none of them support reasoning about laziness.

Calf is a cost-aware logical framework for reasoning about resource usage in full-spectrum dependently-typed functional programs [Niu et al. 2022]. The language also supports effects via the call-by-push-value evaluation [Levy 2001; Pédrot and Tabareau 2020]. The framework has been utilized to reason about amortized cost via *coinduction* [Grodin and Harper 2023].

Compiler optimization for lazy functional languages. Compilers for lazy functional languages employ techniques such as strictness analysis to avoid creating unnecessary thunks [Jones and Partain 1993; Wadler and Hughes 1987]. Ennals and Jones [2003] experimented with a more aggressive optimization called "optimistic evaluation" that speculatively evaluates thunks but aborts if the compiler decides that it's a bad choice. Their results show significant improvement over GHC, but it was ultimately not incorporated in GHC due to its complexity.[4]

Machine-checked complexity theory. Forster and Smolka [2017] formlize a weak call-by-value λ-calculus in Coq called L. They have used L as a model of computation to formally prove the Cook-Levin Theorem [Cook 1971; Gäher and Kunze 2021].

7 CONCLUSION AND FUTURE WORK

In this paper, we present a demand semantics for lazy functional programs that, given any valid output demand, returns the minimal input demand required. We base our demand semantics on Bjerner and Holmström [1989], but we expand it to support higher-order function such as foldr. In addition, we formally prove that the demand semantics is equivalent to the natural semantics of laziness, by showing that it is equivalent to the clairvoyant semantics.

We demonstrate the effectiveness of our approach by applying our method to formally prove that Okasaki's banker's queue is amortized and persistent using the Coq theorem prover. In the process, we propose a novel reverse physicist's method that allows reasoning about amortization and persistence based on the demand semantics in a modular way.

In future work, we would like to apply this approach to larger lazy functional data structures such as finger trees [Claessen 2020; Hinze and Paterson 2006]. We would also like to develop a tool

[4] https://mail.haskell.org/pipermail/haskell/2006-August/018424.html

that automatically generates a function's demand function based on the demand semantics. By integrating this translator with tools like `hs-to-coq`, we can apply our method to more functional programs and data structures existed in mainstream languages such as Haskell.

REFERENCES

Andrew W. Appel, Andrew Tolmach, and Michael Clarkson. 2023. *Verified Functional Algorithms.* Electronic textbook. Version 1.5.4. http://www.cis.upenn.edu/~bcpierce/sf.

Frédéric Besson and Evgeny Makarov. 2023. Micromega: solvers for arithmetic goals over ordered rings. https://coq.inria.fr/doc/v8.17/refman/addendum/micromega.html

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 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 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-coq`m to real-world Haskell code. *J. Funct. Program.* 31 (2021), e5. https://doi.org/10.1017/S0956796820000283

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

Stephen A. Cook. 1971. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (Shaker Heights, Ohio, USA) *(STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158. https://doi.org/10.1145/800157.805047

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

Danny Dubé and Marc Feeley. 2002. A demand-driven adaptive type analysis. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 84–97. https://doi.org/10.1145/581478.581487

Robert Ennals and Simon Peyton Jones. 2003. Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden) *(ICFP '03)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/944705.944731

Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. 2019. Higher-order Demand-driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 3 (2019), 14:1–14:53. https://doi.org/10.1145/3310340

Kenneth Foner, Hengchu Zhang, and Leonidas Lampropoulos. 2018. Keep your laziness in check. *Proc. ACM Program. Lang.* 2, ICFP (2018), 102:1–102:30. https://doi.org/10.1145/3236797

Yannick Forster and Gert Smolka. 2017. Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq. In *Interactive Theorem Proving*. Springer International Publishing, 189–206. https://doi.org/10.1007/978-3-319-66107-0_13

Lennard Gäher and Fabian Kunze. 2021. Mechanising Complexity Theory: The Cook-Levin Theorem in Coq. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 20:1–20:18. https://doi.org/10.4230/LIPIcs.ITP.2021.20

Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. 2019. Demand Control-Flow Analysis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 226–246. https://doi.org/10.1007/978-3-030-11245-5_11

Harrison Grodin and Robert Harper. 2023. Amortized Analysis via Coinduction. *CoRR* abs/2303.16048 (2023). arXiv:2303.16048 https://arxiv.org/abs/2303.16048

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

Ralf Hinze and Ross Paterson. 2006. Finger Trees: A Simple General-Purpose Data Structure. *J. Funct. Program.* 16, 2 (2006), 197–217. https://doi.org/10.1017/S0956796805005769

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1995, Washington, DC, USA, October 10-13, 1995*, Gail E. Kaiser (Ed.). ACM, 104–115. https://doi.org/10.1145/222124.222146

John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

Graham Hutton. 1999. A Tutorial on the Universality and Expressiveness of Fold. *J. Funct. Program.* 9, 4 (1999), 355–372. http://journals.cambridge.org/action/displayAbstract?aid=44275

Simon Peyton Jones and Will Partain. 1993. Measuring the effectiveness of a simple strictness analyser. In *Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, July 5-7, 1993 (Workshops in Computing)*, John T. O'Donnell and Kevin Hammond (Eds.). Springer, 201–221. https://doi.org/10.1007/978-1-4471-3236-3_17

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Haim Kaplan and Robert Endre Tarjan. 1995. Persistent lists with catenation via recursive slow-down. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, Frank Thomson Leighton and Allan Borodin (Eds.). ACM, 93–102. https://doi.org/10.1145/225058.225090

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. 2001. *Call-by-push-value.* Ph. D. Dissertation. Queen Mary University of London, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233

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

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

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

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

Zachary Palmer, Theodore Park, Scott F. Smith, and Shiwei Weng. 2020. Higher-order demand-driven symbolic evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 102:1–102:28. https://doi.org/10.1145/3408984

Pierre-Marie Pédrot and Nicolas Tabareau. 2020. The fire triangle: how to mix substitution, dependent elimination, and effects. *Proc. ACM Program. Lang.* 4, POPL (2020), 58:1–58:28. https://doi.org/10.1145/3371126

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

Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-Passing Style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992.* ACM, 288–298. https://doi.org/10.1145/141471.141563

Ilya Sergey, Simon Peyton Jones, and Dimitrios Vytiniotis. 2014. Theory and practice of demand analysis in Haskell. https://www.microsoft.com/en-us/research/publication/theory-practice-demand-analysis-haskell/ Unpublished draft.

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.* 27 (2017), e11. https://doi.org/10.1017/S0956796817000016

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

Robert Endre Tarjan. 1985. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (1985), 306–318.

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

Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings (Lecture Notes in Computer Science, Vol. 274)*, Gilles Kahn (Ed.). Springer, 385–407. https://doi.org/10.1007/3-540-18317-5_21