Story of Your Lazy Function's Life A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs

Story of Your Lazy Function's Life

A Bidirectional Demand Semantics for Mechanized Cost Analysis of Lazy Programs

LI-YAO XIA, unaffiliated, France
LAURA ISRAEL, Portland State University, USA
MAITE KRAMARZ, University of Toronto, 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 extrinsic 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 implicit queue. We formally prove that the banker's queue and 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.

CCS Concepts: • **Software and its engineering → Functional languages**; **Software performance**; • **Theory of computation → Denotational semantics**; **Program specifications**; **Program verification**.

Additional Key Words and Phrases: formal verification, computation cost, lazy evaluation, amortized analysis

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, Article 237 (August 2024), 34 pages. https://doi.org/10.1145/3674626

1 Introduction

The power of laziness is great, but formal reasoning about its costs is notoriously elusive. After all, lazy evaluation is stateful and produces interleaved computation, with the cost of functions depending 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 and extrinsically

Authors' Contact Information: Li-yao Xia, lyxia@poisson.chat, unaffiliated, France; Laura Israel, laisrael@pdx.edu, Portland State University, Portland, OR, USA; Maite Kramarz, maite.kramarz@mail.utoronto.ca, University of Toronto, Toronto, Ontario, Canada; Nicholas Coltharp, coltharp@pdx.edu, Portland State University, Portland, OR, USA; Koen Claessen, koen@chalmers.se, Chalmers University of Technology, Gothenburg, Sweden; Stephanie Weirich, sweirich@cis.upenn.edu, University of Pennsylvania, Philadelphia, PA, USA; Yao Li, liyao@pdx.edu, Portland State University, Portland, OR, USA.

ACM 2475-1421/2024/8-ART237
https://doi.org/10.1145/3674626

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 $f : A \to B$, we can use the bidirectional demand semantics to 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 (A) to function f and the demand on its output (B^D), the *demand function* f^D calculates the minimal demand on the input (A^D), as well as the computation cost required to obtain the demanded output ($\mathbb{N}$). 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 input and 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 logic for reasoning.

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 artifact, which is publicly available [Xia et al. 2024].

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 5.2–5.3).

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.

```
1  Fixpoint insert (x : nat) (xs : list nat) : list nat :=
2    match xs with
3    | [] => x :: []
4    | y :: ys => if y <=? x then
5                   let zs := insert x ys in y :: zs
6                 else x :: y :: ys
7    end.
8
9  Fixpoint insertion_sort (xs : list nat) : list nat :=
10   match xs with
11   | nil => nil
12   | y :: ys =>
13       let zs := insertion_sort ys in insert y zs
14   end.
```

Fig. 1. The Gallina implementation of `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`). The infix operator `<=?` shown in `insert` is Gallina's "less than or equal" operator on natural numbers, which returns a boolean.

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.

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]).

Representing demand. 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 $\bot$) 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 ordering. 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.
```

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

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

```
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 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. As we saw in the introduction, 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.

For a concrete example, 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).

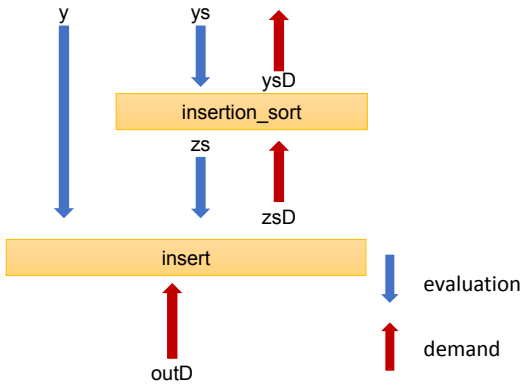


Fig. 3. An illustration of how the input demand is computed in `insertion_sortD`.

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 5).

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 6). Like in `insert`, we then proceed to check whether `y <=? x` (line 7). If `y <=? x`, we make a recursive call to `insertD` to get the input demand of the recursive call to `insert` (lines 8–9). Otherwise, we return `zsD` (line 10). The `let+` notation on line 8 is a custom notation for bind in the `Tick` monad. The `thunkD` combinator on line 8 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 `insertion_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`. Starting from the top, we first call `insertion_sort` (line 22, Fig. 2), then we call `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 : nat -> list A -> list A` as an example: the function call `take n xs` takes the first `n` cells from list `xs`. 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`, the first parameter of `take`. When `n` is a constant, the cost 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 that our approach is compositional. 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 `take` with a different sorting algorithm, *e.g.,* `selection_sort`. We demonstrate how we use demand semantics to reason about all these interactions in Section 4.

2.2 Amortized Analysis for Persistent Data Structures

Besides the usual benefits of lazy evaluation such as compositionality [Hughes 1989], lazy evaluation also enables the combination of *persistence* (old values can be reused) and *amortization* (we average the cost of operations over multiple calls) [Okasaki 1999]. For example, a call to `pop` on a banker's queue may have a high initial cost to evaluate many thunks, so that subsequent calls to `pop` again on the same old queue will be cheap. That is not possible with eager evaluation, where all repeated calls on the same value have the same cost.

In the Rocq Prover, we formally verified amortized cost bounds for both the banker's queue and the implicit queue under persistence. Our final theorems show that the cost of executing a program trace of either the banker's queue or the implicit queue is always linear with respect to the number of operations.

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

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

Fig. 4. Syntax

this paper. We provide more details about the verification of the banker's queue and implicit queue in Section 5.

3 Bidirectional Demand Semantics

In this section, we show the definition of the 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 with suspended computation and force triggers the suspended computation in a thunk (and does nothing if it's a value).

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 depends only on the denotation of its immediate subterms. It can be viewed as a translation from our calculus to a metalanguage able to express our semantics, which mainly consists of pattern-matching and recursion on the list approximation type. The denotational semantics consists of two denotation functions on well-typed terms $\Gamma \vdash M : A$, a pure forward interpretation $[\![M]\!]_{\mathrm{eval}}$ and a demand interpretation $[\![M]\!]_{\mathrm{dem}}$.

Forward evaluation. The "forward evaluation" $[\![M]\!]_{\mathrm{eval}} : [\![\Gamma]\!]_{\mathrm{eval}} \to [\![A]\!]_{\mathrm{eval}}$ 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 $[\![T\, A]\!]_{\mathrm{eval}}$ is the same as the unlifted $[\![A]\!]_{\mathrm{eval}}$. The demand

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

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

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

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

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

Fig. 5. Typing rules.

$$\begin{aligned}
[\![A]\!]_{\mathrm{eval}} &: \mathrm{Set} \\
[\![\mathrm{bool}]\!]_{\mathrm{eval}} &= \{0, 1\} \\
[\![\mathrm{T}\, A]\!]_{\mathrm{eval}} &= [\![A]\!]_{\mathrm{eval}} \\
[\![\mathrm{list}\, A]\!]_{\mathrm{eval}} &= \{\mathrm{nil}\} \uplus \{\mathrm{cons}\, a\, b \mid a \in [\![A]\!]_{\mathrm{eval}}, b \in [\![\mathrm{list}\, A]\!]_{\mathrm{eval}}\}
\end{aligned}$$

$$\begin{aligned}
[\![A]\!]_{\mathrm{approx}} &: \mathrm{Set} \\
[\![\mathrm{bool}]\!]_{\mathrm{approx}} &= \{0, 1\} \\
[\![\mathrm{T}\, A]\!]_{\mathrm{approx}} &= \mathrm{T}\, [\![A]\!]_{\mathrm{approx}} \stackrel{\mathrm{def}}{=} \{\bot\} \uplus \{\mathrm{thunk}\, a \mid a \in [\![A]\!]_{\mathrm{approx}}\} \\
[\![\mathrm{list}\, A]\!]_{\mathrm{approx}} &= \{\mathrm{nil}\} \uplus \{\mathrm{cons}\, a\, b \mid a \in \mathrm{T}\, [\![A]\!]_{\mathrm{approx}}, b \in \mathrm{T}\, [\![\mathrm{list}\, A]\!]_{\mathrm{approx}}\}
\end{aligned}$$

Fig. 6. Sets of total values $[\![A]\!]_{\mathrm{eval}}$ and sets of approximations $[\![A]\!]_{\mathrm{approx}}$

semantics [Bjerner and Holmström 1989] is defined by "backwards evaluation": $[\![M]\!]_{\mathrm{dem}} : [\![\Gamma]\!]_{\mathrm{eval}} \times [\![A]\!]_{\mathrm{approx}} \rightharpoonup \mathbb{N} \times [\![\Gamma]\!]_{\mathrm{approx}}$.

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 $[\![A]\!]_{\mathrm{approx}}$ consists of values with the same shape as in $[\![A]\!]_{\mathrm{eval}}$, possibly with some subterms replaced with a special value $\bot$ representing an unneeded thunk. $[\![A]\!]_{\mathrm{approx}}$ 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 $[\![A]\!]_{\mathrm{eval}}$ are *total values*.

An approximation value $a' : [\![A]\!]_{\mathrm{approx}}$ is an approximation of a total value $a : [\![A]\!]_{\mathrm{eval}}$, a relation denoted $a' \prec a$, when informally "a' has the same shape as a", ignoring **thunk**s, and some subterms

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

Fig. 7. Forward evaluation.

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

Fig. 8. Demand semantics: backward evaluation.

of a may have been replaced with $\bot$ in a'. The set of approximations of a is defined by

$$[\![A]\!]^{\prec a}_{\text{approx}} \stackrel{\text{def}}{=} \{a' \in [\![A]\!]_{\text{approx}} \mid a' \prec a\}$$

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

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

The set $[\![A]\!]^{\prec a}_{\text{approx}}$ 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

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

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 $a \prec b$

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

Fig. 12. Least approximation

of a more defined than both a_1 and a_2. The join is defined formally in Figure 13; it is also extended to environments $\{x_i \mapsto a_i\}_i : [\![\Gamma]\!]_{\mathrm{approx}}$. 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) $a_1 \sqcup a_2 \prec a$
(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 $a : [\![A]\!]_{\mathrm{eval}}$ has a *least approximation* $\bot_a : [\![A]\!]_{\mathrm{approx}}$. This is simply $\bot$ when A is of the form TB, but otherwise $\bot$ is not an element of $[\![A]\!]_{\mathrm{approx}}$, 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 12, also extended to environments $\{x_i \mapsto a_i\}_i : [\![\Gamma]\!]_{\mathrm{approx}}$. As its name implies, it is smaller than all other approximations of a.

Lemma 3.3 (bottom). *If $a' \prec a$, then $\bot_a \le a'$.*

Backwards evaluation. Given an input $g : [\![\Gamma]\!]_{\mathrm{eval}}$ and an approximation a of the output ($a \prec [\![M]\!]_{\mathrm{eval}}(g)$), the demand semantics $[\![M]\!]_{\mathrm{dem}}(g, a) : \mathbb{N} \times [\![\Gamma]\!]_{\mathrm{approx}}^{\prec g}$ 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

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

Fig. 13. Joining approximations (NB: cases with mismatched constructors cannot happen)

output demand a. The demand semantics is defined in Figure 8. Let us remark that the equation of $[\![]\!]_{\text{dem}}$ for $\text{let}\, x = M \,\text{in}\, N$ contains occurrences of both $[\![M]\!]_{\text{eval}}$ and $[\![M]\!]_{\text{dem}}$ (and similarly for foldr). If we view this semantics as a translation, the size of the demand function may thus grow quadratically with respect to the size of the original function. This effect is mitigated in practice because functions bodies are usually small.

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

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

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

$$\exists (n, g') = [\![M]\!]_{\text{dem}}(g, a') \wedge g' \prec g$$

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

$$[\![\Gamma \vdash M : A]\!]_{\text{dem}} : (g : [\![\Gamma]\!]_{\text{eval}}) \times [\![A]\!]^{\prec [\![M]\!]_{\text{eval}}(g)}_{\text{approx}} \to \mathbb{N} \times [\![\Gamma]\!]^{\prec g}_{\text{approx}}$$

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.

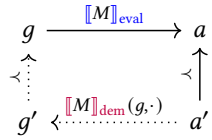
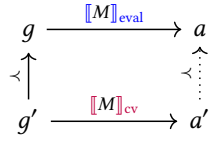
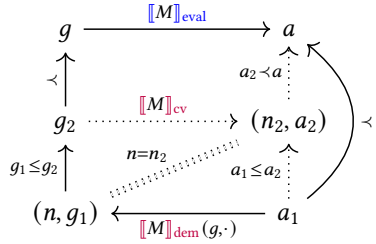


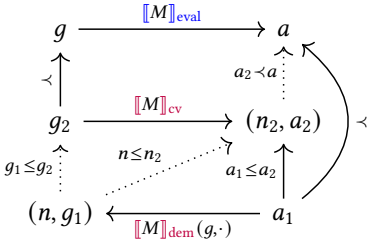
Fig. 14. Diagram of Lemma 3.4 (totality). Full arrows are quantified universally. Dotted arrows are quantified existentially.



(a) Theorem 3.7 (functional correctness)



(b) Theorem 3.8 (cost existence)



(c) Theorem 3.9 (cost minimality)

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

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. For example, when $a_1 = a_2$ and $n_1 = n_2$, we have $n_1 < n_1 + n_1$ when $n_1 \neq 0$.

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 and Hutton 2019; Li et al. 2021], a forward and nondeterministic semantics that 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 $[\![\Gamma \vdash M : A]\!]_{\mathrm{cv}} : [\![\Gamma]\!]_{\mathrm{approx}} \to \mathcal{P}(\mathbb{N} \times [\![A]\!]_{\mathrm{approx}})$. 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 $[\![M]\!]_{\mathrm{cv}}$. 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 $[\![M]\!]_{\mathrm{cv}}$ approximates the pure function $[\![M]\!]_{\mathrm{eval}}$. This theorem is pictured as a commutative diagram in Figure 15a. Let g be an environment, let $a = [\![M]\!]_{\mathrm{eval}}(g)$ (diagrammatically: $g \xrightarrow{[\![M]\!]_{\mathrm{eval}}} a$), let $g' \prec g$, and let a' be a nondeterministic output of $[\![M]\!]_{\mathrm{cv}}(g')$ (diagrammatically: $g' \xrightarrow{[\![M]\!]_{\mathrm{cv}}} a'$; formally: $(n, a') \in [\![M]\!]_{\mathrm{cv}}(g')$ for some n). Then a' approximates a.

Theorem 3.7 (Functional correctness). *Let $g \in [\![\Gamma]\!]_{\mathrm{eval}}$, and $g' \prec g$.*

$$\forall (n, a') \in [\![M]\!]_{\mathrm{cv}}(g'),\ a' \prec [\![M]\!]_{\mathrm{eval}}(g)$$

The demand semantics $[\![M]\!]_{\mathrm{dem}}(g, a_1)$ 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 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 $g \in [\![\Gamma]\!]_{\mathrm{eval}}$, $a_1 \prec [\![M]\!]_{\mathrm{eval}}(g)$, let $(n, g_1) = [\![M]\!]_{\mathrm{dem}}(g, a_1)$, and let g_2 such that $g_1 \le g_2 \prec g$.*

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

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

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

3.4 Deriving the Definition of $[\![M]\!]_{\mathrm{dem}}$

We can use the properties above to derive the definition of $[\![M]\!]_{\mathrm{dem}}$ by inequational reasoning. For example, to find the value of $[\![\mathrm{force}\, M]\!]_{\mathrm{dem}}$ as a function of $[\![M]\!]_{\mathrm{dem}}$, totality requires $[\![\mathrm{force}\, M]\!]_{\mathrm{dem}}(g, a') \prec g$, given $a' \prec_A [\![\mathrm{force}\, M]\!]_{\mathrm{eval}}(g) = [\![M]\!]_{\mathrm{eval}}(g)$, and given the totality property for M, $[\![M]\!]_{\mathrm{dem}}(g, a'') \le g$ for all $a'' \prec_{TA} a$. With $a'' = \mathbf{thunk}\, a'$, we have $[\![M]\!]_{\mathrm{dem}}(g, \mathbf{thunk}\, a') \le g$. This suggests the definition $[\![\mathrm{force}\, M]\!]_{\mathrm{dem}}(g, a') = [\![M]\!]_{\mathrm{dem}}(g, \mathbf{thunk}\, a')$. In a similar way, we can derive the tricky-looking definition of $[\![\mathrm{foldr}\, M_1\, M_2\, N]\!]_{\mathrm{dem}}$ by inequational reasoning.

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

3.5 Limitations of This Work

All of our examples of demand functions were translated manually. Although the demand semantics (Section 3) could be used to systematically translate from a pure function to a demand function, further efforts are necessary to simplify the generated code into a readable result conducive to mechanical reasoning. An automatic translation would significantly improve the usability of our framework.

Nevertheless, as manual translations can be error-prone, we have developed a method to ensure our translation is correct by cross-validating it with the clairvoyant semantics [Hackett and Hutton 2019; Li et al. 2021]. We will explain this framework in more detail in Section 4 and Section 5.

The bidirectional demand semantics presented here does not support general recursive and higher-order functions, which limits the expressiveness of the demand semantics. However, our case studies show that even with those restrictions, the 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 lack of general recursion is not restrictive in the context of time complexity analysis. We can simulate general recursion using a fuel parameter which decreases with every recursive call. This does not change the asymptotic complexity of programs, and time complexity bounds will determine how much fuel is enough.

Many classical data structures and algorithms in complexity analysis only involve first-order functions such as `push` and `pop` for queues. That is what our present work focuses on. It would be useful to support higher-order functions to reason about operations such as maps, traversals, filters, and folds. The bidirectional demand semantics originally presented by [Bjerner and Holmström 1989] does not feature higher-order functions either. Extending those semantics with higher-order functions would complicate the results of this section. Currently, we use the same denotation of types ($[\![A]\!]_{\text{approx}}$) for both the demand semantics and the clairvoyant semantics, allowing us to easily compare values between the two semantics. In the clairvoyant semantics, the interpretation of function types $A \to B$ as monadic functions $[\![A]\!]_{\text{approx}} \to \mathcal{P}(\mathbb{N} \times [\![B]\!]_{\text{approx}})$ is specific to clairvoyant semantics. It does not seem suitable to represent the demand on a function. Intuitively, functions propagate demand in two ways. When a function is applied, the demand on its result is mapped to the demand on its arguments and the function itself. When a function is defined, the demand on the function is mapped to a demand on the context of the lambda abstraction. It would be interesting to find a representation of demand for functions that meets those needs.

Demand semantics let us reason about total time cost, but not real-time (*i.e.,* non-amortized) cost, or space cost. This is because demand functions only calculate *if* a thunk is evaluated, not *when* a thunk is evaluated.

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 $n = |\texttt{xs}|$. 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 artifact [Xia et al. 2024].

4.1 The `take` Function

We show the definitions of `take` and `takeD` in Fig. 16. Surprisingly, `take` is not expressible in the calculus defined in the previous section: indeed, to define `take` using `foldr`, we also need first-class functions. We were not able to remove this unfortunate limitation without significantly increasing the complexity of our calculus. In practice, we are still able to manually define a demand function for `take` and many similar functions. As a safeguard, we cross-validate this demand function `takeD` with the clairvoyant semantics, which is simply a monadic translation `takeA`. We prove a correspondence between `takeD` and `takeA` in the form of the theorems of Section 3.3.

```
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} (k : nat) (xs : list A)
11                   (outD : listA A) : Tick (T (listA A)) :=
12   tick >> match k, xs, outD with
13   | O, _, _ => ret Undefined
14   | _, nil, _ => ret (Thunk NilA)
15   | S k', x :: xs', ConsA zD zsD =>
16     let+ ysD := thunkD (takeD k' xs') zsD in
17     ret (Thunk (ConsA (Thunk x) ysD))
18   | _, _, _ => bottom (* absurdity case *)
19   end.
```

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

To define `takeD`, we apply $[\![\cdot]\!]_{\text{dem}}(g, d)$ (Fig. 8) to the definition of `take` to obtain the following:

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

Because t is an argument of function `takeD` so we use the rule for evaluating variables to get $[\![t]\!]_{\text{dem}}(g, n') = (0, \{t \mapsto n'\})$. However, before we move on, we would like to treat specially the numbers contained in argument t. 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 make a simplification in `takeD` such that we only return the demand of the argument `xs`. For this reason, we will ignore the numbers in t in the rest of the translation process.

The $\mathbf{foldr}_{\text{dem}}$ 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 $t = \mathbf{nil}$, we return $\mathbf{thunk}\ \mathbf{nil}$ (line 14, Fig. 16). In the case that $t = \mathbf{cons}\ (n, y)\ ts$, there are two steps. First, we need to run the following:

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

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:

$$\begin{aligned}(c_2, g_2, a'_2) =& \mathbf{foldr}_{\mathrm{dem}}(g, M_1, M_2, a_2, b'_2)\\ =& \mathbf{foldr}_{\mathrm{dem}}(g, \mathbf{cons}\, y\, zs, \mathbf{nil}, zs, zsD)\end{aligned}$$

This is equivalent to a recursive call applied to zsD (line 16). According to the definition of $\mathbf{foldr}_{\mathrm{dem}}$, 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).

After these steps, we need to add back the `nat` argument `k` to the definition. In the case that `k = 0`, the original `take` function returns `nil`. By the rule for `nil` in Fig. 8, we know that we should return `Undefined` ($\bot$) in this case (line 13). We also handle the case in which the output demand is $\bot$ and the case in which we apply $[\![\mathbf{cons}\, y\, zs]\!]_{\mathrm{dem}}$ 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).

Finally, we manually add a `tick` in the beginning of the function (line 12) so that we can count the number of function calls.

Cost. Now that we have defined `takeD`, we can use it to reason about the cost of `take`. We can state and prove the following cost 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 function `Tick.cost` projects the computation cost from the `Tick` monad. 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 $\max(1, |\texttt{outD}|)$.

Functional correctness. Since our translation is manual, we additionally prove the following functional correctness theorem for `takeD`:

```
Lemma takeD_approx (n : nat) (xs : list nat) outD :
  outD `is_approx` take n xs ->
  Tick.val (takeD n xs outD) `is_approx` xs.
```

The function `Tick.val` projects the value from the `Tick` monad. We prove a similar theorem for every demand function that we translate so that we can be more confident about our translation.

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 `let`-expressions with function calls. According to the `let`-rule of the demand semantics (Fig. 8), we first need to run `insert` 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.

```
(* Computation cost. *)
Theorem insertD_cost x (xs : list nat)  (outD : listA nat) :
  Tick.cost (insertD x xs outD) <= leb_count x xs + 1.
Theorem insertD_cost' x (xs : list nat) (outD : listA nat) :
  Tick.cost (insertD x xs outD) <= sizeX' 1 outD.
Theorem 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).
Theorem selectD_cost (x : nat) (xs : list nat) (yD : nat) (ysD : listA nat) :
  Tick.cost (selectD x xs (pairA yD ysD)) <= length xs + 1.
Theorem 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).

(* Functional correctness. *)
Theorem insertD_approx (x : nat) (xs : list nat) (outD : listA nat) :
  outD `is_approx` insert x xs ->
  Tick.val (insertD x xs outD) `is_approx` xs.
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 selectD_approx (x : nat) (xs : list nat) (outD : prodA nat (listA nat)):
  outD `is_approx` select x xs ->
  Tick.val (selectD x xs outD) `is_approx` xs.
Theorem selection_sortD_approx (xs : list nat) (n : nat) (outD : listA nat) :
  outD `is_approx` selection_sort xs n ->
  Tick.val (selection_sortD xs n outD) `is_approx` xs.

(* 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.
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. 17. Main theorems we have proven for both insertion sort and selection sort.

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]. We also defined our custom own tactics to help solve `is_approx` relations.

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 $n = |\texttt{xs}|$.

```
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.

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

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

Compared to `insertion_sort`, `selection_sort` is more challenging because it returns a product type. Accordingly, we need to use `prodA 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`, as long as the demand on the output number (the smallest number in the list) is not `Undefined`, `select` still must traverse the entire input list to select such a number.

We show the main theorems we have proven for selection sort in Fig. 17. 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`. We also additionally prove the functional correctness theorems of `selectD` and `selection_sortD` to validate our manual translation. In the end, we can compose `takeD_cost` with `selection_sortD_cost` to prove the bounds stated in `take_selection_sortD_cost`.[3]

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].

[2]It is possible to manually prove that `selection_sort` terminates without using this "fuel" construct. 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`.

```
let q0 = empty in
let q1 = push q0 a in
let q2 = push q1 b in    (* D q2@2 = (a:nil,b:bot)                        *)
let q3 = push q2 c in    (* D q2@3 = (a:nil,b:bot)                        *)
let q4 = push q3 d in    (* D q2@4 = (bot, bot)    D q4@4 = (a:b:bot,bot) *)
(_, q5) <- pop q4 ;;     (* D q2@5 = (bot, bot)    D q4@5 = (a:b:bot,bot) *)
(_, q6) <- pop q5 ;;     (* D q2@6 = (bot, bot)    D q4@6 = (a:bot,bot)   *)
(_, q7) <- pop q4 ;;     (* D q2@7 = (bot, bot)    D q4@7 = (bot,bot)     *)
(* ... *)
```

Fig. 19. A program that uses the banker's queue, with demands of q2 and q4 labeled at each step.

5.1 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 $q^D = \{nf, f^D, nb, b^D\}$ as:

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

We use f^D to represent the demand of the front list and nb to represent the length of the back list. We overload the $|\cdot|$ operator to represent the length of a demand of a list, so $|f^D|$ represents the length of f^D. We use nb instead of $|b^D|$ in the potential function because moving a `back` list to the `front` list requires reversing the entire `back` list, regardless of the length of b^D. Note that $|f^D| \leq nf$ and $|b^D| \leq nb$.

For each operation of the a queue (*i.e.,* `push` and `pop`), we show that the following inequality holds:

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

$\Phi(q^D_{in})$ 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*).

To show that a queue is amortized and persistent over arbitrary program traces, we consider the demands for all versions of the queue generated in program points. We use $q^D_{i@j}$ to denote the demand for i-th queue at the j-th operation (both indices start from 0). We constrain j to be at least as large as i for any $q^D_{i@j}$. Note that we do not put any constraints on the program trace—one version of a queue can be reused for any number of times in a program trace.

Taking the program shown in Fig. 19 as an example. We start from $q^D_{i@j} = (\bot, \bot)$ for any $j \geq 8$, because there is no more demand after the program has finished. Using the initial list of $q^D_{i@8}$ for all is as the output demand, we can compute the minimal input demands for every queue at each step

```
1  Definition Physicist'sArgumentD : Prop :=
2    forall (o : op) (vs : list value), well_formed vs ->
3    forall output : stackA, output `is_approx` eval o vs ->
4    forall input cost, Tick.MkTick cost input = demand o vs output ->
5    sumof potential input + cost <= budget o vs + sumof potential output.
6
7  Definition AmortizedCostSpec : Prop :=
8    forall os : trace, (cost_of (exec_trace os) <= budget_trace os).
```

Fig. 20. Definitions of the reverse physicist's method (`Physicist'sArgumentD`) as well as amortization and persistence (`AmortizedCostSpec`) formalized in the Rocq Prover.

using our demand semantics (Section 3). The demands of `q2` and `q4` at each operation are shown as comments in Fig. 19.

This notion allows us to lift inequality over a single operation, *i.e.,* Inequality (2), to the following inequality over parts of a program trace:

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

We use $cost_{[i,j]}$ to represent 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. In addition, 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 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 $\sum_{i=0}^{t-1} \Phi(q^D_{i@t}) = 0$. Therefore, if we can show that Inequality (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.

We show our Rocq Prover formalization of Inequality (3) in lines 1–5 of Fig. 20. The definition `Physicist'sArgumentD` is defined generally so that we can use it on both the banker's queue and the implicit queue. The definition states that for any list of values `vs` (line 2), if we have a "stack" of approximations `output` of running `vs` using the pure function of operation `o`, where `o` can be either a `push` or a `pop` (line 3), and if we can run the demand function of operation `o` on `vs` and `output` to obtain a computation cost `cost` and an input demand `input` (line 4), then the sum of all `input`'s potential plus `cost` is smaller than or equal to a budget assigned to operation `o` over value `vs` plus the sum of all `output`'s potential (line 5).

The final theorem that states a queue is both amortized and persistent is shown in lines 7–8 in Fig. 20. We show that for any queue `q`, if we can show that a queue and all its operations satisfy `Physicist'sArgumentD`, it implies that the queue also satisfies `AmortizedCostSpec`. In this way, we only need to show that the banker's queue and the implicit queue both satisfy `Physicist'sArgumentD`.

5.2 Banker's Queue

We show a Gallina implementation of the banker's queue in Fig. 21. 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 1–2). When `push`ing an element to the queue, we add it to the head of the `back` list (lines 9–10). When

```
1  Record Queue (A : Type) : Type := MkQueue
2    { nfront : nat; front : list A; nback : nat; back : list A }.
3
4  Definition mkQueue {A : Type}
5    (nf : nat) (f : list A) (nb : nat) (b : list A) : Queue A :=
6    if nf <? nb then MkQueue (nf + nb) (append f (rev b)) 0 []
7    else MkQueue nf f nb b.
8
9  Definition push {A : Type} (q : Queue A) (x : A) : Queue A :=
10   mkQueue (nfront q) (front q) (nback q + 1) (x :: back q).
11
12 Definition pop {A : Type} (q : Queue A) : option (A * Queue A) :=
13   match front q with
14   | x :: f => Some (x, mkQueue (pred (nfront q)) f (nback q) (back q))
15   | [] => None
16   end.
```

Fig. 21. The banker's queue implemented in Gallina.

```
1  Record QueueA (A : Type) : Type := MkQueueA
2    { nfrontA : nat; frontA : T (listA A); nbackA : nat; backA : T (listA A) }.
3
4  Definition pushD {A} (q : Queue a) (x : A)
5                       (outD : QueueA A) : Tick (T (QueueA A) * T A) :=
6    tick >> let+ (frontD, backD) := mkQueueD (nfront q) (front q)
7                                     (S (nback q)) (x :: back q) outD in
8    ret (Thunk (MkQueueA (nfront q) frontD (nback q) (tailX backD)), Thunk x).
9
10 Definition popD {A} (q : Queue A)
11                    (outD : option (T A * T (QueueA A))) : Tick (T (QueueA A)) :=
12   tick >>
13   match front q, outD with
14   | [], _ => Tick.ret (exact q)
15   | x :: f, Some (xA, pop_qA) =>
16     let+ (fD, bD) := thunkD (mkQueueD (pred (nfront q)) f
17                                       (nback q) (back q)) pop_qA in
18     ret (Thunk (MkQueueA (nfront q) (Thunk (ConsA xA fD)) (nback q) bD))
19   | _, _ => bottom
20   end.
```

Fig. 22. Demand functions of the banker's queue.

poping an element from the queue, we remove it from the head of the `front` list (lines 12–16). Both functions 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 shorter than `back` (lines 4–7).

```
1  Definition potential (q: QueueA A) : nat :=
2    2 * (sizeX 0 (frontD q) - nbackD q).
3
4  Definition const : nat := 7.
5
6  Theorem pushD_cost {A} (q : Queue A) (x : A) (OutD : QueueA A) :
7    well_formed q ->
8    qOutD `is_approx` push q x ->
9    let (cost, qInD) := pushD q x qOutD in
10   potential qInD + cost <= potential qOutD + const.
11
12 Lemma popD_cost {A} (q : Queue A) (outD : option (T A * T (QueueA A))) :
13   well_formed q ->
14   outD `is_approx` pop q ->
15   let qA := Tick.val (popD q outD) in
16   let cost := Tick.cost (popD q outD) in
17   potential qA + cost <= const + potential outD.
18
19 Lemma pushD_spec {A} (q : Queue A) (x : A) (outD : QueueA A) :
20   outD `is_approx` push q x ->
21   forall qD xD, (qD, xD) = Tick.val (pushD q x outD) ->
22   let dcost := Tick.cost (pushD q x outD) in
23   pushA qD xD [[ fun out cost => outD `less_defined` out /\ cost <= dcost ]].
24
25 Lemma popD_spec {A} (q : Queue A) (outD : option (T A * T (QueueA A))) :
26   outD `is_approx` pop q ->
27   forall qD, qD = Tick.val (popD q outD) ->
28   let dcost := Tick.cost (popD q outD) in
29   popA qD [[ fun out cost => outD `less_defined` out /\ cost <= dcost ]].
```

Fig. 23. Cost specification for the banker's queue.

We manually derive all the demand functions of the banker's queue, shown in Fig. 22. We show the cost theorems we have proven for the banker's queue in Fig. 23. 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 Equality (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 6–10). 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. The theorem additionally requires the queue `q` to be well formed (`well_formed`, line 7), which is the invariant that all the banker's queue's operations maintain: (1) the `back` list is not longer than the

front list, and (2) the nfront and nback fields correctly represent the lengths of the front list and the back list, respectively. Similarly, we prove a cost theorem for popD (lines 12–17).

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 its translation from pure functions is more mechanical: it can be done by adding 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 in lines 19–29 in Fig. 23. Theorems pushD_spec and popD_spec state the equivalence relation between the demand functions pushD and popD with the clairvoyant functions pushA and popA. The [[...]] notation is called an *optimistic specification* [Li et al. 2021], which is a form of incorrectness logic [O'Hearn 2020] that shows the *existence* of an output approximation and computation cost that satisfies the property specified in [[...]].

5.3 Implicit Queue

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

An implicit queue of type A is an inductive data type that is either an empty queue Nil (line 2) or a "deep" structure (lines 3) that contains: (1) a Front, which contains one or two As, (2) a Rear, which contains zero or one A, and (3) an inner queue of a product A * A.

When pushing a new element to the queue, we first perform a pattern match on the input queue q (line 7). If q is empty, we add the new element to Front directly (line 8). If q is a Deep structure (line 9), we pattern match on its Rear (line 11). If its Rear contains zero elements, we put the new element in its Rear (line 12). If q is a Deep structure that already has an element y in the Rear (the maximal number of element in Rear), we make a *polymorphic recursive call* to push the product of y and the new element x to the inner queue (line 13). The use of polymorphic recursion is important for the efficiency of the implicit queue—the technique is known as *recursive slowdown*.

When poping an element from the queue, we again first perform a pattern matching on q (line 20). When q is Nil, there is simply nothing to pop so we return None (line 21). When q is a Deep structure, we first check its Front. If the Front contains only one element (the minimal number of elements in Front), we pop two elements from the inner queue to make the new Front if the inner queue still contains some elements (lines 27–30). If there is no more element from the inner queue (line 31), we move the Rear element to Front if there is one (line 33) or Nil otherwise (line 34). However, if the Front contains two elements, we just need to retrieve the first one (line 37).

We carefully defined the push and pop functions for the implicit queue in Fig. 24 so that they can take advantage of lazy let-bindings in lazy languages. For example, the push function can return a Deep structure without evaluating the input queue. Similarly, the push function can evaluate the Front elements in a Deep structure without evaluating the Rear of the input queue.

We define demand functions pushD and popD of the implicit queue. One challenge with these demand functions is polymorphic recursion. For example, the push function over a polymorphic type A runs a recursive call on the type A * A (line 13). In a lazy setting, the type A * A can be half evaluated—it might have an evaluated value as its first element while its second element is unevaluated. This requires the demand functions to be even "more polymorphic" because it needs to keep track of both pure values and demands. In practice, we define two extra demand functions pushD' and popD' that are the result of demand translation of push and pop but with one more polymorphic type as their parameters. We then define pushD and popD in terms of pushD' and popD'. We show the approximation of the implicit queue's Queue datatype, the type signatures of pushD'

```
Inductive Queue (A : Type) : Type :=
| Nil : Queue A
| Deep : Front A -> Queue (A * A) -> Rear A -> Queue A.

Fixpoint push (A : Type) (q : Queue A) (x : A) : Queue A :=
  let '(f, m, r) :=
    match q with
    | Nil => (FOne x, Nil, RZero)
    | Deep f m r =>
        let (m, r) :=
          match r with
          | RZero => (m, ROne x)
          | ROne y => (push m (y, x), RZero)
          end in
        (f, m, r)
    end in
  Deep f m r.

Fixpoint pop (A : Type) (q : Queue A) : option (A * Queue A) :=
  match q with
  | Nil => None
  | Deep f m r =>
      let (x, q) :=
        match f with
        | FOne x =>
            let q :=
              match (pop m) with
              | Some yzm' =>
                  let '((y, z), m') := yzm' in
                  Deep (FTwo y z) m' r
              | None =>
                  match r with
                  | ROne y => Deep (FOne y) Nil RZero
                  | RZero => Nil
                  end
              end in (x, q)
        | FTwo x y => (x, Deep (FOne y) m r)
        end in
      Some (x, q)
  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. We also simplify the code for readability. The actual code in our artifact is written in ANF for an easier translation.

```
Inductive QueueA (A : Type) : Type :=
| NilA : QueueA A
| DeepA : T (FrontA A) -> T (QueueA (prodA A A)) -> T (RearA A) -> QueueA A.

Definition pushD' {A B : Type} `{Exact A B}
  (q : Queue A) (x : A) (outD : QueueA B) : Tick (prodA (QueueA B) B).

Definition pushD {A : Type} :
  Queue A -> A -> QueueA A -> Tick (prodA (QueueA A) A) :=
  pushD'.

Definition popD' {A B : Type}
  (q : Queue A) (outD : option (T (prodA B (QueueA B)))) : Tick (T (QueueA B)).

Definition popD {A : Type} :
  Queue A -> option (T (prodA A (QueueA A))) -> Tick (T (QueueA A)) :=
  popD'.
```

Fig. 25. The approximation of the implicit queue's Queue datatype, the type signatures of pushD' and popD', and the definitions of pushD and popD of the implicit queue.

and popD', and the definitions of pushD and popD in Fig. 25. Interested readers can find detailed definitions of these functions, which are too long to be shown in this paper, in Appendix A.1 and in our artifact.

We also proved all the cost theorems for these demand functions similar to the banker's queue, which include that the demand functions agree with a clairvoyant translation of push and pop on computation cost. The cost theorems that we have proven can be found in Appendix A.2 and in our artifact. At last, we proved that the implicit queue satisfy Physicist'sArgumentD and AmortizedCostSpec (Fig. 20), which means that the implicit queue is both amortized and persistent.

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]. However, what they expose to the users are abstract and in the style of Okasaki's debit-based reasoning with no mutation or reasoning about ownership involved.

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.

Cost analysis or amortized analysis for non-lazy semantics. Cost analysis or amortized cost analysis in the context of strict semantics has been the subject of much research as well [Cutler et al. 2020; Danner et al. 2013; Hoffmann et al. 2012; McCarthy et al. 2018; 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]

Improvement Theory. Improvement theory studies if one program *improves* another program in terms of reduction steps under all program contexts [Sands 1996, 1997]. Moran and Sands [1999] showed that the improvement theory can be applied in a lazy setting as well, based on prior work on the call-by-need λ-calculus [Ariola et al. 1995]. Gustavsson and Sands [2001] also studied the space complexity under lazy evaluation using the improvement theory. Our work focuses on computation cost and amortized cost analysis based on proof assistants, rather than improvement relations between programs.

Machine-checked complexity theory. Forster and Smolka [2017] formalize 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 functions 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 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 in mainstream languages such as Haskell.

Acknowledgments

We thank all the ICFP 2024 reviewers and PLDI 2024 reviewers whose suggestions helped significantly improve this paper. We thank Joseph W. Cutler and Cassia Torczon for their involvement during the early stage of this project. Katie Casamento, James Hook, Mark P. Jones, Allison Naaktgeboren, Andrew Tolmach, and Grant VanDomelen have provided valuable suggestions during the authors' discussions with them. We also appreciate the feedback from participants at UCSC LSD Seminar and PNW PLSE Workshop 2024. This work was partially supported by the National Science Foundation under Grant Nos. CCF-2006535 and CNS-2244494.

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

A Appendix: Rocq Prover Formalization of the Implicit Queue

A.1 Demand Functions

```
Inductive QueueA (A : Type) : Type :=
| NilA : QueueA A
| DeepA : T (FrontA A) -> T (QueueA (prodA A A)) -> T (RearA A) -> QueueA A.

Fixpoint pushD' (A B : Type) `{Exact A B} (q : Queue A) (x : A) (outD : QueueA B) :
  Tick (prodA (QueueA B) B) :=
  let+ qD :=
    Tick.tick >>
      match outD with
      | DeepA fD mD rD =>
          match q with
          | Nil => Tick.ret (Thunk NilA)
          | Deep f m r =>
              match r with
              | RZero => Tick.ret (Thunk (DeepA fD mD (Thunk RZeroA)))
              | ROne y =>
                  let+ uD := thunkD (pushD' m (y, x)) mD in
                  let '(pairA mD pD) := uD in
                  let (yD, xD) :=
                    match pD with
                    | Thunk (pairA yD xD) => (yD, xD)
                    | _ => bottom
                    end in
                  Tick.ret (Thunk (DeepA fD mD (Thunk (ROneA yD))))
              end
          end
      | _ => bottom
      end in
    Tick.ret (pairA qD (exact x)).

Definition pushD (A : Type) : Queue A -> A -> QueueA A -> Tick (prodA (QueueA A) A) :=
  pushD'.

Fixpoint popD' (A B : Type) (q : Queue A) (outD : option (T (prodA B (QueueA B)))) :
  Tick (T (QueueA B)) :=
  Tick.tick >>
    match q with
    | Nil => Tick.ret (Thunk NilA)
    | Deep f m r =>
        let+ (fD, mD, rD) :=
          let (xD, qD) :=
            match outD with
            | Some (Thunk (pairA xD qD)) => (xD, qD)
            | _ => bottom
            end in
```

```
      match f with
      | FOne x =>
          let p := pop m in
          let (pD, rD) :=
            match p with
            | Some (yz, m') =>
                match qD with
                | Thunk (DeepA fD mD' rD) =>
                    let yzD :=
                      Thunk (match fD with
                             | Thunk (FTwoA yD zD) => pairA yD zD
                             | _ => bottom
                             end) in
                    (Thunk (Some (Thunk (pairA yzD mD'))), rD)
                | _ => bottom
                end
            | None =>
                let rD :=
                  match r with
                  | RZero => Thunk RZeroA
                  | ROne y =>
                      let yD :=
                        match qD with
                        | Thunk (DeepA (Thunk (FOneA yD)) _ _) => yD
                        | _ => bottom
                        end in
                      Thunk (ROneA yD)
                  end in
                (Thunk None, rD)
            end in
          let+ mD := thunkD (popD' m) pD in
          Tick.ret (Thunk (FOneA xD), mD, rD)
      | FTwo x y =>
          let '(yD, mD, rD) :=
            match qD with
            | Thunk (DeepA fD' mD rD) =>
                let yD :=
                  match fD' with
                  | Thunk (FOneA yD) => yD
                  | _ => bottom
                  end in
                (yD, mD, rD)
            | _ => bottom
            end in
          Tick.ret (Thunk (FTwoA xD yD), mD, rD)
      end in
    Tick.ret (Thunk (DeepA fD mD rD))
end.
```

```
Definition popD (A : Type) (q : Queue A) (outD : option (T (prodA A (QueueA A)))) :
  Tick (T (QueueA A)) :=
  popD' q outD.
```

A.2 Potential Functions and Cost Theorems

```
Definition size_FrontA (A : Type) (fA : FrontA A) : nat :=
  match fA with
  | FOneA _ => 1
  | FTwoA _ _ => 2
  end.

Definition size_RearA (A : Type) (rA : RearA A) : nat :=
  match rA with
  | RZeroA => 0
  | ROneA _ => 1
  end.

Instance Potential_QueueA : forall (A : Type), Potential (QueueA A) :=
  fix potential_QueueA (A : Type) (qA : QueueA A) :=
    match qA with
    | NilA => 0
    | DeepA fD mD rD =>
        let c := T_rect _ size_FrontA 2 fD - T_rect _ size_RearA 0 rD
        in c + @Potential_T _ (potential_QueueA _) mD
    end.

Lemma pushD'_cost : forall (A B : Type) `{LessDefined B, Exact A B}
  (q : Queue A) (x : A) (outD : QueueA B),
    outD `is_approx` push q x ->
    let inM := pushD' q x outD in
    let cost := Tick.cost inM in
    let (qD, _) := Tick.val inM in
    potential qD + cost <= 2 + potential outD.

Corollary pushD_cost : forall (A : Type) `{LessDefined A}
  (q : Queue A) (x : A) (outD : QueueA A),
    outD `is_approx` push q x ->
    let inM := pushD q x outD in
    let cost := Tick.cost inM in
    let (qD, _) := Tick.val inM in
    potential qD + cost <= 2 + potential outD.

Lemma pushD'_spec (A B : Type) :
  forall `{LDB : LessDefined B, !Reflexive LDB, Exact A B}
    (q : Queue A) (x : A) (outD : QueueA B),
    outD `is_approx` push q x ->
```

```
    forall qD xD, pairA qD xD = Tick.val (pushD' q x outD) ->
             let dcost := Tick.cost (pushD' q x outD) in
             pushA qD xD [[ fun out cost =>
                              outD `less_defined` out /\ cost <= dcost ]].

Corollary pushD_spec (A : Type) :
  forall `{LDA : LessDefined A, !Reflexive LDA}
    (q : Queue A) (x : A) (outD : QueueA A),
    outD `is_approx` push q x ->
    forall qD xD, pairA qD xD = Tick.val (pushD' q x outD) ->
             let dcost := Tick.cost (pushD' q x outD) in
             pushA qD xD [[ fun out cost =>
                              outD `less_defined` out /\ cost <= dcost ]].

Lemma popD'_cost : forall (A B : Type)
                    `{LessDefined B, Exact A B}
                    (q : Queue A) (outD : option (T (prodA B (QueueA B)))),
    outD `is_approx` pop q ->
    let d := match outD with
             | Some (Thunk (pairA _ qD)) => potential qD
             | _ => 0
             end in
    let inM := popD' q outD in
    let cost := Tick.cost inM in
    let inD := Tick.val inM in
    potential inD + cost <= 3 + d.

Corollary popD_cost :
  forall (A : Type) `{LessDefined A}
         (q : Queue A) (outD : option (T (prodA A (QueueA A)))),
    outD `is_approx` pop q ->
    let d := match outD with
             | Some (Thunk (pairA _ qD)) => potential qD
             | _ => 0
             end in
    let inM := popD' q outD in
    let cost := Tick.cost inM in
    let inD := Tick.val inM in
    potential inD + cost <= 3 + d.

Lemma popD'_spec :
  forall (A B : Type) `{LDB : LessDefined B, !Reflexive LDB, Exact A B}
    (q : Queue A) (outD : option (T (prodA B (QueueA B)))),
    outD `is_approx` pop q ->
    forall qD, qD = Tick.val (popD' q outD) ->
    let dcost := Tick.cost (popD' q outD) in
    popA qD [[ fun out cost => outD `less_defined` out /\ cost <= dcost ]].
```

```
Corollary popD_spec :
  forall (A : Type) `{LDA : LessDefined A, !Reflexive LDA}
    (q : Queue A) (outD : option (T (prodA A (QueueA A)))),
    outD `is_approx` pop q ->
    forall qD, qD = Tick.val (popD' q outD) ->
    let dcost := Tick.cost (popD' q outD) in
    popA qD [[ fun out cost => outD `less_defined` out /\ cost <= dcost ]].
```

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.

Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. 1995. The Call-by-Need Lambda Calculus. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 233–246. https://doi.org/10.1145/199448.199507

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-coqm` 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

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

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

Jörgen Gustavsson and David Sands. 2001. Possibilities and Limitations of Call-by-Need Space Improvement. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 265–276. https://doi.org/10.1145/507635.507667

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

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

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

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time Credits and Time Receipts in Iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 3–29. https://doi.org/10.1007/978-3-030-17184-1_1

Andrew Moran and David Sands. 1999. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 43–56. https://doi.org/10.1145/292540.292547

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

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

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

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

David Sands. 1996. Total Correctness by Local Improvement in the Transformation of Functional Programs. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 175–234. https://doi.org/10.1145/227699.227716

David Sands. 1997. From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 428–441. https://doi.org/10.1145/263699.263760

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

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 (Artifact). https://doi.org/10.5281/zenodo.11493754