Unifying Hindsight and Foresight

Lazy Cost Analysis as Functional Logic Programming

Nicholas Coltharp[1], Steven Libby[2], Laura Israel[3], and Yao Li[1⋆]

[1] Portland State University, USA
[2] University of Portland, USA
[3] University of Konstanz, Germany

Abstract. Clairvoyance semantics and demand semantics are both pure and time-cost equivalent to lazy evaluation, offering alternatives to the stateful natural semantics of lazy evaluation for analyzing computation cost. Unfortunately, clairvoyance semantics is simple but hard to execute, and demand semantics is executable but complicated. In this paper, we propose a novel approach for unifying these two semantics using functional logic programming. We propose (1) a logical clairvoyance translation, which translates lazy functional programs to functional logic programs where the computation cost is reified, and (2) a logical demand translation, which is a simple extension to the logical clairvoyance translation. We prove that these two translations correctly implement clairvoyance and demand semantics respectively using Agda. We also conduct case studies on examples including insertion sort and Okasaki's banker's queue using Curry/KiCS2 to show how these translations can be used in practice.

Keywords: Functional Logic Programming, Lazy Evaluation, Cost Analysis

1 Introduction

Lazy evaluation is demand-driven, performing only those operations required to compute a final result [15]. This gives programmers the luxury of writing highly compositional code that performs comparably to, or even better than, code written in an eagerly-evaluated language [13]. Sadly, as a "side-effect," it presents challenges for performance analysis.

For example, consider the Haskell functions `take` and `(++)`, which respectively take the length-n prefix of a list and concatenate two lists. Suppose that we want to evaluate the following code:

```
take 2 ([1,2,3] ++ [4,5])
```

In a lazy language like Haskell, such an expression is evaluated only insofar as its result *demands*. Suppose that we try to print just the first element of the result: we run `take` and `(++)` once to obtain `1`. Now suppose that we save the result in a

⋆ Corresponding author.

```
take :: Int -> [a] -> [a]
takeC :: Int -> [a]^A -> P([a]^A, ℕ)
takeD :: Int -> [a] -> [a]^A -> ([a]^A, ℕ)

(++) :: [a] -> [a] -> [a]
appendC :: [a]^A -> [a]^A -> P([a]^A, ℕ)
appendD :: [a] -> [a] -> [a]^A -> (([a]^A, [a]^A), ℕ)
```

Fig. 1: The Haskell type signatures of `take`, `(++)`, and their respective clairvoyance and demand translations. $\mathcal{P}$ is the power set operator. The superscript ^{A} denotes the *approximation* of a type.

variable and, in the future, try to print the whole thing: at that point in time, we evaluate `take` and `(++)` "one level deeper" to obtain the second element, `2`, and `take` again to obtain the end of the list, `[]`. These computations are evaluated *on demand* and *out of order*, and they are *interleaved*. Thus, even in a pure language like Haskell, the performance characteristics of the computation are inherently *stateful*: the amount of work required to evaluate a given expression depends on what has been evaluated previously. Together, these characteristics mean that one cannot generally analyze a piece of lazy code in isolation; instead, one must consider the whole program. This makes performance analysis difficult.

How can we bring locality, purity, order, and compositionality to lazy computation time costs when these properties are fundamentally missing? Some recent work proposes that, instead of working with the stateful natural semantics of lazy evaluation [15], we can use a simpler *alternative semantics* that is equivalent to lazy evaluation in terms of computation cost. Here, we focus on two such semantics: clairvoyance semantics [8, 17] and demand semantics [2, 30].

Clairvoyance semantics. Clairvoyance semantics, or clairvoyant call-by-value, builds on the observation that, for analyzing computation costs, it matters only *whether* a computation cost is incurred, not *when*. Clairvoyance semantics evaluates terms eagerly, but when there is a lazy operation, it makes a *nondeterministic* choice to either evaluate the computation or skip it.

To evaluate `take 2 ([1,2,3] ++ [4,5])` in clairvoyance semantics, we first evaluate `[1,2,3] ++ [4,5]`. Each recursive function application is then nondeterministically evaluated or skipped. The possible results of `[1,2,3] ++ [4,5]` are `1:`$\bot$, `1:2:`$\bot$, $\cdots$, `[1,2,3,4,5]`, where $\bot$ represents a value whose evaluation is skipped. Next, we run `take 2` over all the nondeterministic results of `[1,2,3] ++ [4,5]`. If a branch gets "stuck," *e.g.*, forcing the second element from `take 2 (1:`$\bot$`)`, we discard it. In the end, one computation branch will have succeeded in the same amount of time as a standard lazy evaluation.

Clairvoyance semantics can be used as a recipe for a mechanical translation from a pure program to a program with reified computation costs [17]. We show pseudo type signatures of translated `take` and `(++)` in Fig. 1 as `takeC` and

`appendC`, respectively. These translated functions use *approximations* of lists, denoted as $\texttt{[a]}^A$. They represent lists that are potentially "less evaluated", like $\texttt{1:}\bot$. We defer a formal definition of approximations to Section 3.2. A natural number ($\mathbb{N}$) in the result represents the associated computation cost. Because the evaluation is nondeterministic, each translated function returns a power set of result-cost pairs. We can use these translated functions to formally reason about the computation cost of lazy programs, as illustrated by Li *et al.* [17]

One major drawback of this approach is that power sets in return types make execution expensive. This makes it hard to use clairvoyance semantics in testing.

Demand semantics. Demand semantics builds on the observation that, given a function's input and a demand on its output, we can evaluate the minimal input demand required for such an output demand *deterministically*.

In our example of `take 2 ([1,2,3] ++ [4,5])`, we first consider a demand on the output, *e.g.*, $\texttt{1:}\bot$, where $\bot$ represents parts of the list that are not evaluated. We then run the `take` function "backward" to find that the minimal input demand required for `take 2` is also $\texttt{1:}\bot$. We then run the `(++)` function "backward" as well to find that the minimal input demands required for the two input lists are $\texttt{1:}\bot$ and $\bot$, respectively. Throughout this process, the associated computation cost is also evaluated and carried.

Demand semantics is also useful as a translation strategy [30]. We show pseudo type signatures of the translated functions `takeD` and `appendD` in Fig. 1. As in the clairvoyance translation, we use a natural number in the result to represent computation cost. The demand translation also uses approximations like $\texttt{1:}\bot$, but it uses them to represent demands. For example, the input of `takeD` includes the input of `take` (`Int` and `[a]`), a demand on `take`'s output ($\texttt{[a]}^A$). The return type of `takeD` is the minimal demand on its input ($\texttt{[a]}^A$) and the cost ($\mathbb{N}$). Note that the `Int` datatype is not lazy, so there is no demand on it.

Because these demand functions are deterministic, computation costs are easy to test. Unfortunately, the demand translation is more complicated than the clairvoyance translation, because the "backward" direction of a function can be complex. This complexity makes it challenging to develop tools to formally reason about code under demand semantics.

Our contributions. It is unsatisfying that, although we have two semantics that model lazy computation costs, they are very different and we have to choose between them depending on what we need. In this paper, we propose a novel approach of *unifying* these two semantics using *functional logic programming*. In this new approach, we translate a lazy functional program to a functional logic program that models clairvoyance semantics via a *logical clairvoyance translation*. This functional logic program can then be extended to obtain a new program that encodes demand semantics via a *logical demand translation* using a simpler wrapper. This simple wrapper shows that there is a simple connection between clairvoyance semantics and demand semantics. In other words, these two independently proposed semantics are two sides of the same coin. We show this connection both in theory and in practice.

```
data T a = Undefined | Thunk a

takeC :: Int -> T (List a) -> Tick (List a)
takeC n xsT = do
  tick
  fcase n `compare` 0 of
    EQ -> return Nil
    GT -> do
      xs <- force xsT
      fcase xs of
        Nil -> return Nil
        x :~ xsT' -> (x:~) <$> thunk (takeC (n - 1) xsT')

takeD :: Approx a => Int -> T (List a) -> List a -> Tick (T (List a))
takeD n xsT ysD
    | takeC n xsTD =:= Tick (ysD, c)
    = Tick (xsTD, c)
  where xsTD = approx xsT
        c free
```

Fig. 2: The logical clairvoyance translation (`takeC`) and logical demand translation (`takeD`) of the `take` function, implemented in Curry.

We show an example of translating the Haskell `take` function to Curry [10] in Fig. 2. Function `takeC` is the result of a logical clairvoyance translation, while `takeD` is the result of a logical demand translation.

The `takeC` function takes an `Int` and a `T (List a)`. The `T` datatype represents a value that could be $\bot$ (represented by `Undefined`). `List a` is the approximation of `[a]`: `Nil` is the empty list, and the `:~` operator is the cons operator on `List a`. The function returns a new `List a` inside a `Tick` monad, which encodes the computation cost associated with the returned `List a`. The function works similarly to `take` for the most part. The key to implementing clairvoyance semantics is the recursive call `thunk (takeC n' xsT')`, which unfolds to:

```
pure Undefined ? fmap Thunk (takeC n' xsT')
```

That is, we use Curry's nondeterministic choice operator `?` to either skip the evaluation and return `Undefined` (*i.e.*, $\bot$), or to evaluate the recursive call and wrap the result inside a `Thunk` constructor.

The `takeD` function is implemented in terms of `takeC`. The core of `takeD` is its guard condition: `takeC n xsTD =:= Tick (ysD, c)`, where `n` and `ysD` are `takeD`'s arguments, and `xsTD` and `c` are logic variables (also known as free variables). In this way, `takeD` computes the input `xsTD` and parts of the output `c` of `takeC`, given the rest of its output `ysD`. Put differently, `takeD` is an *inversion* of `takeC`. In addition, we require `xsTD` to be an approximation of `xsT`, another argument of `takeD`, so that it either is `xsT`, or is "less evaluated" than `xsT`.

Thanks to functional logic programming compilers, we now have a simple way of executing clairvoyance semantics. Our work also proposes a simpler way to encode demand semantics.

More concretely, we make the following contributions:

- We propose and formalize two methods that translate functional programs to functional logic programs that reify lazy computation costs: a logical clairvoyance translation that encodes clairvoyance semantics, and a logical demand translation that wraps the logical clairvoyance translation and encodes demand semantics (Section 2).
- We formally prove that the logical clairvoyance translation and the logical demand translation are correct with respect to clairvoyance and demand semantics, respectively. All theorems and proofs are mechanized in the proof assistant Agda [22] (Section 3).
- We conduct case studies by manually translating lazy Haskell programs, including insertion sort and Okasaki's banker's queue [23], to Curry [10] based on our translation strategies. We evaluate the performance of translated code using the KiCS2 Curry compiler [4], showing that logical demand translation trades off performance for simplicity, but the performance is usable for lazy cost analysis while allowing for simpler code (Section 4).

In addition, we discuss related work in Section 5, discuss limitations and conclude in Section 6. All the Agda formalizations and Curry code of this paper can be found in our publicly available artifact:

https://doi.org/10.5281/zenodo.18808172

2 Logical Clairvoyance Translation and Logical Demand Translation

In this section, we define two languages: a simple functional language **B** (Section 2.1) and a simple functional logic language **L** (Section 2.2). Next, we develop two translations from **B** to **L**: a *logical clairvoyance translation* which produces terms modeling clairvoyance semantics [8, 17], and a *logical demand translation*, which is a wrapper around the logical clairvoyance translation, which produces terms modeling demand semantics [2, 30] (Section 2.3).

2.1 The Base Language B

We show the syntax of the base language **B** in Fig. 3. Language **B** is a simply typed calculus with booleans, lists, and lazy thunks (*i.e.*, suspended computations). Its terms include variables, `let`-bindings, conditionals, and recursion based on `foldr`. In addition, it contains `lazy` and `force` operators, which wrap a computation in a lazy thunk and force a computation from a thunk, respectively. Finally, the `tick` operator incurs one unit of computation cost. These explicit operators decouple our analysis from any particular evaluation model.

$$
\begin{array}{lcl}
A & ::= & \texttt{Bool} \mid \texttt{List}\ A \mid \texttt{Thunk}\ A \\
t & ::= & x \mid \texttt{let}\ x\ \texttt{=}\ t\ \texttt{in}\ t \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ t\ \texttt{else}\ t \\
 & \mid & \texttt{[]} \mid t\ \texttt{::}\ t \mid \texttt{foldr}(\lambda x x.t,\ t,\ t) \mid \texttt{lazy}\ t \mid \texttt{force}\ t \mid \texttt{tick}\ t
\end{array}
$$

Typing

$$\frac{\Gamma \vdash^{\mathbf{B}} t_1 : A \qquad \Gamma \vdash^{\mathbf{B}} t_2 : \texttt{Thunk (List}\ A\texttt{)}}{\Gamma \vdash^{\mathbf{B}} t_1\ \texttt{::}\ t_2 : \texttt{List}\ A}\text{CONS}$$

$$\frac{\Gamma, x_1 : A, x_2 : \texttt{Thunk}\ B \vdash^{\mathbf{B}} t_1 : B \qquad \Gamma \vdash^{\mathbf{B}} t_2 : B \qquad \Gamma \vdash^{\mathbf{B}} t_3 : \texttt{List}\ A}{\Gamma \vdash^{\mathbf{B}} \texttt{foldr}(\lambda x_1 x_2.t_1,\ t_2,\ t_3) : B}\text{FOLDR}$$

$$\frac{\Gamma \vdash^{\mathbf{B}} t : A}{\Gamma \vdash^{\mathbf{B}} \texttt{lazy}\ t : \texttt{Thunk}\ A}\text{LAZY} \qquad \frac{\Gamma \vdash^{\mathbf{B}} t : \texttt{Thunk}\ A}{\Gamma \vdash^{\mathbf{B}} \texttt{force}\ t : A}\text{FORCE} \qquad \frac{\Gamma \vdash^{\mathbf{B}} t : A}{\Gamma \vdash^{\mathbf{B}} \texttt{tick}\ t : A}\text{TICK}$$

Fig. 3: The syntax of language **B** and typing rules related to lazy thunks and costs.

Note that language **B** does not have higher-order functions or general recursion because demand semantics does not support these features [2, 30].

Most of the typing rules for language **B** are standard, so we omit them. We show the typing rules for thunks and computation costs in Fig. 3. Notably, lists are lazy: the cons of lists t_1 `::` t_2 takes a term t_1 of type A and a term t_2 of type `Thunk (List` A`)`, so the tail of a list must always be wrapped in a `Thunk`. Similarly, the "function" argument to `foldr` always requires that its second parameter be wrapped in a `Thunk`.

We can give language **B** a denotational semantics $[\![t]\!](\gamma)$ for any term t in an environment γ by simply ignoring all operations related to lazy evaluation and computation costs:

$$[\![\texttt{lazy}\ t]\!](\gamma) = [\![\texttt{force}\ t]\!](\gamma) = [\![\texttt{tick}\ t]\!](\gamma) = [\![t]\!](\gamma)$$

2.2 The Logic Language L

We show the syntax of language **L** in Fig. 4. Compared to **B**, language **L** has more types: unit `Unit`, natural numbers `Nat`, and products A `*` B. We need natural numbers and products to represent computation costs associated with values, and we need products and the unit type to represent environments. Note that the types of language **L** are a superset of the types of **B**. As a consequence, any valid typing context for **B** is also a valid typing context for language **L**.

Language **L** has introduction and elimination forms for all of the new types. In particular, the `lazy` and `force` operators from **B** have been replaced by constructors `thunk` and `undefined`, and eliminator `case`. The `thunk` operator reduces its argument and wraps it in a **thunk** value; `undefined` reduces to the

$$
\begin{array}{lcl}
A & ::= & \texttt{Unit} \mid \texttt{Bool} \mid \texttt{Nat} \mid A \texttt{ * } A \mid \texttt{List } A \mid \texttt{Thunk } A \\
t & ::= & x \mid \texttt{let } x \texttt{ = } t \texttt{ in } t \mid \texttt{()} \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } t \texttt{ then } t \texttt{ else } t \\
& \mid & t \texttt{ == } t \mid t \texttt{ <= } t \mid \texttt{(}t\texttt{, }t\texttt{)} \mid \texttt{fst } t \mid \texttt{snd } t \mid \texttt{thunk } t \mid \texttt{undefined} \\
& \mid & \texttt{case } t \texttt{ of thunk } x \texttt{ => } t\texttt{; undefined => } t \mid n \mid t \texttt{ + } t \mid \texttt{[]} \mid t \texttt{ :: } t \\
& \mid & \texttt{foldr(}\lambda x x.t\texttt{, }t\texttt{, }t\texttt{)} \mid t \texttt{ ? } t \mid \texttt{free } A \mid \texttt{fail}
\end{array}
$$

Typing

$$\frac{\Gamma \vdash^{\mathbf{L}} t : A}{\Gamma \vdash^{\mathbf{L}} \texttt{thunk } t : \texttt{Thunk } A} \qquad \frac{}{\Gamma \vdash^{\mathbf{L}} \texttt{undefined} : \texttt{Thunk } A}$$

$$\frac{\Gamma \vdash^{\mathbf{L}} t_1 : \texttt{Thunk } A \qquad \Gamma, x : A \vdash^{\mathbf{L}} t_2 : B \qquad \Gamma \vdash^{\mathbf{L}} t_3 : B}{\Gamma \vdash^{\mathbf{L}} \texttt{case } t_1 \texttt{ of thunk } x \texttt{ => } t_2\texttt{; undefined => } t_3 : B}$$

$$\frac{}{\Gamma \vdash^{\mathbf{L}} \texttt{free } A : A} \qquad \frac{\Gamma \vdash^{\mathbf{L}} t_1 : A \qquad \Gamma \vdash^{\mathbf{L}} t_2 : A}{\Gamma \vdash^{\mathbf{L}} t_1 \texttt{ ? } t_2 : A} \qquad \frac{}{\Gamma \vdash^{\mathbf{L}} \texttt{fail} : A}$$

Fig. 4: The syntax and typing rules of the logic language **L**.

special value $\bot$; `case` discriminates between these two cases. In the logical clairvoyance translation, the `thunk` and `undefined` constructors indicate whether a computation is evaluated or not; in the logical demand translation, they indicate whether a computation is demanded or not.

Language **L** also has constructs for logic programming. Free terms (`free`) may evaluate to any value of a specified type; they can be thought of as "anonymous logic variables". The choice operator (`?`) evaluates nondeterministically to either of its operands. Failure (`fail`) never evaluates to anything.

Finally, **L** has two new relational operators, `==` and `<=`. The `==` operator indicates equality, as usual. The `<=` does *not* indicate the standard order on natural numbers; rather, it indicates the *definedness* relation. Given two values a, a', we write a`<=`a', pronounced "a is less defined than a'," if a has the same shape as a', but with some **thunk** parts possibly replaced by $\bot$.

We define a simple logic language instead of using a specific functional logic language like Curry [10, 28] or the Verse calculus [1] to keep our approach generally applicable.

We show the semantic rules of **L** that are related to the choice operator (`?`) and free terms in Fig. 4. Note that this semantics is eager, and due to `fail`, it is partial.

Derived terms in **L**. We use **L** to implement functional logic programs that "carry" a computation result and its associated computation cost. To encapsulate operations related to computation costs, we define some derived term constructs in **L** in Fig. 5. We use `return` t to lift t to a computation that costs 0. We use x `<-` t_1`;` t_2 to chain computations t_1 and t_2 while accumulating their costs. We use `foldM(`$\lambda x_1 x_2.t_1$`,` t_2`,` t_3`)` to chain a list of computations, accumulating all

$$
\begin{aligned}
\texttt{M}\ A &= A\ \texttt{*}\ \texttt{Nat} \\
\texttt{return}\ t &= (t,\ 0) \\
x\ \texttt{<-}\ t_1\texttt{;}\ t_2 &= \texttt{let}\ z_1\ \texttt{=}\ t_1\ \texttt{in} \\
&\quad\ \texttt{let}\ x\ \texttt{=}\ \texttt{fst}\ t_1\ \texttt{in} \\
&\quad\ \texttt{let}\ z_2\ \texttt{=}\ t_2\ \texttt{in} \\
&\quad\ (\texttt{fst}\ z_2,\ \texttt{snd}\ z_1\ \texttt{+}\ \texttt{snd}\ z_2) \\
\texttt{foldM}(\lambda x_1 x_2.t_1,\ t_2,\ t_3) &= \texttt{foldr}(\lambda x_1 z_2.x_2\ \texttt{<-}\ \texttt{transpose}\ z_2\texttt{;}\ t_1,\ t_2,\ t_3) \\
\texttt{transpose}\ t &= \texttt{case}\ t\ \texttt{of} \\
&\quad\ \texttt{thunk}\ z_1\ \texttt{->}\ (z_2\ \texttt{<-}\ z_1\texttt{;}\ \texttt{return}\ (\texttt{thunk}\ z_2))\texttt{;} \\
&\quad\ \texttt{undefined}\ \texttt{->}\ \texttt{return}\ \texttt{undefined} \\
\texttt{assert}\ t_1\ \texttt{in}\ t_2 &= \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ \texttt{fail}
\end{aligned}
$$

$$
\frac{\Gamma \vdash^{\mathbf{L}} t : A}{\Gamma \vdash^{\mathbf{L}} \texttt{return}\ t : \texttt{M}\ A}
\qquad
\frac{\Gamma \vdash^{\mathbf{L}} t_1 : \texttt{M}\ A \qquad \Gamma, x : A \vdash^{\mathbf{L}} t_2 : \texttt{M}\ B}{\Gamma \vdash^{\mathbf{L}} x\ \texttt{<-}\ t_1\texttt{;}\ t_2 : \texttt{M}\ B}
$$

$$
\frac{\Gamma, x_1 : A, x_2 : \texttt{Thunk}\ B \vdash^{\mathbf{L}} t_1 : \texttt{M}\ B \qquad \Gamma \vdash^{\mathbf{L}} t_2 : \texttt{M}\ B \qquad \Gamma \vdash^{\mathbf{L}} t_3 : \texttt{List}\ A}{\Gamma \vdash^{\mathbf{L}} \texttt{foldM}(\lambda x_1 x_2.t_1,\ t_2,\ t_3) : \texttt{M}\ B}
$$

$$
\frac{\Gamma \vdash^{\mathbf{L}} t : \texttt{Thunk}\ (\texttt{M}\ A)}{\Gamma \vdash^{\mathbf{L}} \texttt{transpose}\ t : \texttt{M}\ (\texttt{Thunk}\ A)}
\qquad
\frac{\Gamma \vdash^{\mathbf{L}} t_1 : \texttt{Bool} \qquad \Gamma \vdash^{\mathbf{L}} t_2 : A}{\Gamma \vdash^{\mathbf{L}} \texttt{assert}\ t_1\ \texttt{in}\ t_2 : A}
$$

Fig. 5: Derived constructs in **L** and their typing rules. Readers who are familiar with monads might recognize that `M` is a writer monad [20, 29].

of their costs. The `foldM` operation employs an auxiliary construct `transpose`, which converts terms of type `Thunk (M A)` to terms of type `M (Thunk A)`. In addition, we define `assert`, which ensures that a given condition holds, causing the computation to fail otherwise.

We show all the typing rules related to these derived terms in Fig. 5.

2.3 Logical Clairvoyance Translation and Logical Demand Translation

We define the logical clairvoyance translation $\mathbb{C}\lfloor\cdot\rfloor$ in Fig. 6. The logical clairvoyance translation translates terms in **B** to terms in **L** that simulate clairvoyance semantics. Values and variables are translated using `return`. Variable bindings (`let`) and conditionals (`if`) are translated via sequencing ($\cdot$ `<-` $\cdot$`;` $\cdot$). Folds (`foldr`) are translated to the derived term construct `foldM`. Readers familiar with monads might recognize that this is essentially a monadic translation [20, 24, 29] with adaptations for the `Thunk` type [17].

The most important cases of the translation are those for `lazy` and `force`. When the original term is `lazy` t, we use the nondeterministic choice operator `?` to either skip the inner term (`return undefined`) or evaluate the inner term and wrap the result inside a `thunk`. When the original term is `force` t, we first evaluate t and then pattern match on the result z_1: if z_1 evaluates to

Logical Clairvoyance Translation:

$$\begin{aligned}
\mathbb{C}\lfloor x \rfloor &= \texttt{return } x \qquad \mathbb{C}\lfloor \texttt{[]} \rfloor = \texttt{return []} \\
\mathbb{C}\lfloor \texttt{let } x \texttt{ = } t_1 \texttt{ in } t_2 \rfloor &= x \texttt{ <- } \mathbb{C}\lfloor t_1 \rfloor\texttt{; } \mathbb{C}\lfloor t_2 \rfloor \\
\mathbb{C}\lfloor t_1 \texttt{ :: } t_2 \rfloor &= z_1 \texttt{ <- } \mathbb{C}\lfloor t_1 \rfloor\texttt{; } z_2 \texttt{ <- } \mathbb{C}\lfloor t_2 \rfloor\texttt{; return } (z_1 \texttt{ :: } z_2) \\
\mathbb{C}\lfloor \texttt{foldr}(\lambda x_1 x_2 . t_1\texttt{, } t_2\texttt{, } t_3) \rfloor &= z \texttt{ <- } \mathbb{C}\lfloor t_3 \rfloor\texttt{; foldM}(\lambda x_1 x_2 . \mathbb{C}\lfloor t_1 \rfloor\texttt{, } \mathbb{C}\lfloor t_2 \rfloor\texttt{, } z) \\
\mathbb{C}\lfloor \texttt{lazy } t \rfloor &= \texttt{return undefined ? } z \texttt{ <- } t\texttt{; return (thunk } z\texttt{)} \\
\mathbb{C}\lfloor \texttt{force } t \rfloor &= z_1 \texttt{ <- } t\texttt{; case } z_1 \texttt{ of} \\
&\qquad \texttt{thunk } z_2 \texttt{ => return } z_2\texttt{; undefined => fail} \\
\mathbb{C}\lfloor \texttt{tick } t \rfloor &= \texttt{let } z \texttt{ = } \mathbb{C}\lfloor t \rfloor \texttt{ in (fst } z\texttt{, snd } z + 1\texttt{)}
\end{aligned}$$

Logical Demand Translation:

$$\begin{aligned}
\mathbb{D}_{x_1:A_1,\ldots,x_n:A_n}\lfloor t \rfloor = \texttt{let } x_1' \texttt{ = free } A_1 \texttt{ in assert } x_1' \texttt{ <= } x_1 \texttt{ in} \\
\vdots \\
\texttt{let } x_n' \texttt{ = free } A_n \texttt{ in assert } x_n' \texttt{ <= } x_n \texttt{ in} \\
\texttt{let } z \texttt{ = } \mathbb{C}\lfloor t' \rfloor \texttt{ in} \\
\texttt{assert fst } z \texttt{ == } y \texttt{ in ((} x_1'\texttt{, ..., } x_n'\texttt{), snd } z\texttt{)} \\
\text{where } t' = t[x_1'/x_1, \ldots, x_n'/x_n].
\end{aligned}$$

Fig. 6: The logical clairvoyance translation $\mathbb{C}\lfloor \cdot \rfloor$ and logical demand translation $\mathbb{D}_\Gamma\lfloor \cdot \rfloor$.

`thunk` z_2, then we return z_2; otherwise, we `fail`. Note that `fail` only terminates the current computation branch, so any branches where z_1 evaluates to a `thunk` will continue. To `tick` a computation, we simply run it and increment its accumulated cost by 1. This translation correctly models sharing—a feature that typically poses a challenge to cost analysis—because the result term in **B** eagerly evaluates each value only once in every nondeterministic branch.

We also define the logical demand translation $\mathbb{D}_\Gamma\lfloor t \rfloor$ from **B** to **L** in Fig. 6.

In general, the logical demand translation operates on open terms with an arbitrary number of free variables, but we can build an intuition for how it works by considering terms with just one free variable. Let $t(x)$ be a term in **L** with one free variable x. Intuitively, the demand semantics interpretation of t is a term $t'(x, y)$ that takes the original parameter x and a new *demand* parameter y representing the demand on $t(x)$. The term $t'(x, y)$ computes the minimal demand on x needed to produce y. From this specification, we can see that $t'(x, y)$ acts as an *inverse* of $t(x)$ in the parameter y. In general, any term $t(x)$ has a canonical inverse term $t^{-1}(y)$ given by

$$t^{-1}(y) = \texttt{let } x \texttt{ = free } A \texttt{ in assert } t(x) \texttt{ == } y \texttt{ in } x.$$

The logical demand translation *extends* such an inversion: $t'(x, y)$ does not just produce any value inverse to y, it produces one that is less defined than x. We

add this constraint:

$$t'(x,y) = \texttt{let } x' \texttt{ = free } A \texttt{ in assert } x' \texttt{ <= } x \texttt{ in}\\ \texttt{assert } t(x') \texttt{ == } y \texttt{ in } x'.$$

The missing piece is that the logical demand translation must turn terms in **B** into terms in **L**. We can achieve this by composing t' with some previously-defined translation. As we show in Section 3, the logical clairvoyance translation is exactly the translation that we need. After adjusting for the fact that the logical clairvoyance translation produces terms of type `M`, we arrive at the single-variable case of the logical demand translation:

$$\mathbb{D}\lfloor t \rfloor(x,y) = \texttt{let } x' \texttt{ = free } A \texttt{ in assert } x' \texttt{ <= } x \texttt{ in}\\ \texttt{let } z \texttt{ = } \mathbb{C}\lfloor t \rfloor(x') \texttt{ in assert fst } z \texttt{ == } y \texttt{ in}\\ (x', \texttt{ snd } z).$$

Extending this translation to multiple variables gives us the translation in Fig. 6.

Notably, the demand semantics of Bjerner and Holmström [2] and Xia *et al.* [30] define how to invert every term explicitly, whereas we simply rely on logic variable resolution techniques such as narrowing.

We prove that logical clairvoyance translation and logical demand translation preserve the well-typedness of the original term in **B**:

Theorem 1. *If* $\Gamma \vdash^{\mathbf{B}} t : A$*, then* $\Gamma \vdash^{\mathbf{L}} \mathbb{C}\lfloor t \rfloor : \texttt{M } A$.

Theorem 2. *If* $\Gamma \vdash^{\mathbf{B}} t : A$*, then* $\Gamma, x : A \vdash^{\mathbf{L}} \mathbb{D}_{\Gamma}\lfloor t \rfloor : \texttt{M } |\Gamma|$.

The operation $|\cdot|$ turns typing contexts into product types. Both theorems are formalized and proven in Agda.

3 Correctness

In this section, we prove that logical clairvoyance translation and logical demand translation correctly implement clairvoyance semantics and demand semantics, respectively. All of our proofs are mechanized in the Agda proof assistant [22]. Our proofs rely on the equivalence between clairvoyance and demand semantics proven by Xia *et al.* [30, Section 3.3], which we axiomatize in our formalization.

We show an overview of how we formally prove correctness theorems in Fig. 7. We first define clairvoyance semantics $\mathbb{C}[\![\cdot]\!]$ on **B**. We then show that starting from a term t in **B**, we can translate it to a term in **L** using the logical clairvoyance translation $\mathbb{C}\lfloor t \rfloor$ that has the same meaning as $\mathbb{C}[\![t]\!]$. We also prove a similar theorem relating demand semantics to the logical demand translation.

3.1 The logical clairvoyance translation

Clairvoyance semantics is a big-step operational semantics with the judgment schema $\gamma \,;\, t \Downarrow_c^{\mathbb{C}} v$. Intuitively, c is the number of `tick`s encountered while evaluating t to v. Our semantics is similar to the original clairvoyance semantics

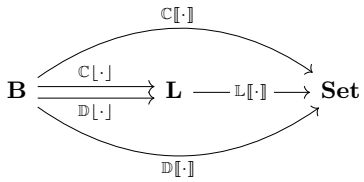

Fig. 7: An overview of our work. $\mathbb{C}[\![\cdot]\!]$ and $\mathbb{D}[\![\cdot]\!]$ respectively represent clairvoyance and demand semantics; $\mathbb{L}[\![\cdot]\!]$ is the semantics of $\mathbf{L}$; $\mathbb{C}\lfloor\cdot\rfloor$ and $\mathbb{D}\lfloor\cdot\rfloor$ are term translations. All arrows from $\mathbf{B}$ to $\mathbf{Set}$ commute (up to a suitable notion of equivalence).

of Hackett and Hutton [8]. Like their semantics, ours is based on improvement theory [21]. The most significant difference is that our calculus has explicit `lazy` and `force` operators. Therefore, whereas Hackett and Hutton introduce nondeterminism at `let` bindings, we must introduce it at `lazy` terms. Readers can find detailed definitions of our clairvoyance semantics in Appendix B and in our artifact.

Theorem 3 (Correctness of the logical clairvoyance translation). *$\gamma\ ;\ t \Downarrow_c^{\mathbb{C}} v$ if and only if $\gamma\ ;\ \lfloor t \rfloor \Downarrow^{\mathbb{L}} (v, c)$.*

3.2 The logical demand translation

Demand semantics is a deterministic semantics that interprets a term t as a function $\mathbb{D}[\![t]\!]$ that takes two arguments: an *exact* evaluation context γ (containing only values with no $\bot$ parts) and an *approximation* value that represents the demand on the result of $[\![t]\!](\gamma)$. It produces a list γ^A of minimal demands on γ and a computation cost. Intuitively, demand semantics computes "how much" of the input γ is required to compute a desired "amount" of the output $[\![t]\!](\gamma)$ and how much it costs to perform the computation.

We omit detailed definitions of demand semantics on $\mathbf{B}$, as it is essentially the same as that of Xia *et al.* [30]. Interested readers can refer to them in Appendix C and in our artifact.

Lemma 1 (Semantics of the logical demand translation). *$\gamma, y^A \mapsto a^A\ ;\ \mathbb{D}_\Gamma\lfloor t \rfloor \Downarrow^{\mathbb{L}} (|\gamma^A|, c)$ if and only if (1) $\gamma^A \leq \gamma$, and (2) $\gamma^A\ ;\ t \Downarrow_c^{\mathbb{C}} a^A$.*

Proof. By construction, applying Theorem 3.

We can now establish the correctness of the logical demand translation via two theorems: adequacy and soundness.

Theorem 4 (Adequacy of the logical demand translation). *Define $(\gamma^A, c) = \mathbb{D}[\![t]\!](\gamma, a^A)$ and let $\gamma^{A\prime} \geq \gamma^A$. Then there exists some $a^{A\prime} \geq a^A$ such that $\gamma, y^A \mapsto a^{A\prime}\ ;\ \mathbb{D}_\Gamma\lfloor t \rfloor \Downarrow^{\mathbb{L}} (\gamma^{A\prime}, c)$.*

```
newtype Tick a =
  Tick { runTick :: (a, Int) }

data T a = Undefined | Thunk a

thunk :: Applicative f =>
  f a -> f (T a)
thunk m =
  pure Undefined ? Thunk <$> m

force :: Applicative f =>
  T a -> f a
force (Thunk x) = pure x
```

```
data List a =
  Nil | (:~) a (T (List a))

class Data a => Approx a where
  (<~) :: a -> a -> Bool

approx :: a -> a
approx x | y <~ x = y where y free

instance Approx a => Approx (T a) where
  Undefined <~ _         = True
  Thunk _   <~ Undefined = False
  Thunk x   <~ Thunk y   = x <~ y
```

Fig. 8: Key definitions for implementing the logical clairvoyance semantics and the logical demand semantics in Curry. See Fig. 2 for the definitions of `thunk` and `T`.

Theorem 5 (Soundness of the logical demand translation). *If* $\gamma, y^A \mapsto a^A \; ; \mathbb{D}_\Gamma \lfloor t \rfloor \Downarrow^{\mathbb{L}} (|\gamma^A|, c)$*, then* $\mathbb{D}[\![t]\!](\gamma, a^A) \leq (\gamma^A, c)$.

The proofs of these theorems depend on Lemma 1 and the theorems of cost existence and cost minimality in Xia *et al.* [30].

4 Case Studies

To show that the logical clairvoyance translation and logical demand translation work in practice, we conduct some case studies by manually applying these translation strategies on classical Haskell functions and implementing all translated functions in Curry [10]. These examples include appending lists, insertion sort, and Okasaki's banker's queue [23]. We use the KiCS2 Curry compiler [4] to test these implementations. We discuss the details of our implementation in this section.

In addition, we evaluate the performance of translated logical demand functions. Our results show that, although there are slowdowns compared with the specialized demand translation of Xia *et al.* [30], they achieve acceptable performance despite being much simpler. All the code of our case studies can be found in our publicly-available artifact.

Foundations. We begin by building a framework for representing the logical clairvoyance semantics and the logical demand semantics in Curry.

We show key definitions of this framework in Fig. 8. We represent approximation types using the `Approx` typeclass. The typeclass contains an `<~` operator, which indicates its left-hand operand approximates its right-hand operand. The other method, `approx`, nondeterministically returns a value that approximates

its argument. We include a default implementation of `approx` using free variables, but a specialized implementation can be much faster. The `Approx` typeclass depends on the `Data` typeclass, which is required for free variables. The `List` type models lazy lists: `Nil` is the empty list, and the `:~` operator produces a new list by adding an element to the front of a (thunked) list.

The `Tick` datatype is a product type which uses an `Int` to represent the computation cost associated with its value. It is essentially a writer monad [20, 29]. `List a` is the approximation type of lists `[a]`. These types have all of the expected typeclass instances; *e.g.*, `Tick` is a `Monad`. These definitions are standard, and we omit them for brevity.

Examples implemented. We manually apply the logical clairvoyance translation and logical demand translation on classical Haskell functions and implement all translated functions in Curry. Examples include appending lists, insertion sort, and Okasaki's banker's queue [23]. Interested readers can view an insertion sort case study in Appendix E, and more in our publicly available artifact.

Running computations. All the translated functions can be executed using the KiCS2 Curry compiler, including all the logical clairvoyance functions. In our implementation, we arrange our definitions so that the less-defined result is always on the left branch of a search tree; thus, by using Curry's *search tree* package [3], we can always find the least result via a depth-first search.

Guards vs. generators. In Curry, there are two ways of implementing logical demand functions: guards and generators. Although they are semantically equivalent, there is a huge performance difference. Naïvely, we can implement `takeD` as follows:

```
takeD' :: Approx a => Int -> T (List a) -> List a -> Tick (T (List a))
takeD' xsT ysD |  xsTD <~ xsT
               && takeC n xsTD =:= Tick (ysD, c)
               =  Tick (xsTD, c)
  where xsTD, c free
```

In contrast to the definition in Fig. 2, we use the `<~` operator in a guard condition in `takeD'` to assert that `xsTD` is an approximation of `xsT`. In this case, Curry must "guess" a valid input demand, performing poorly. On the other hand, our implementation in Fig. 2 uses a generator `approx` defined for the `List` datatype, which is more efficient.

Performance. Although the logical demand translation is simpler than the original demand translation, the question remains whether it is reasonably efficient, as it depends on the logical clairvoyance translation and free variables. To evaluate this, we ran `insertionSortD` on lists of the form `[n, n-1 .. 1]` with sizes ranging from 1 to 100, demanding the full list each time. We also ran `pushPopD`, the logical demand translation of a function that pushes n elements onto a banker's queue [23] and then pops them all. We show the result of our performance evaluation in Fig. 9. All experiments were performed on an AMD Ryzen

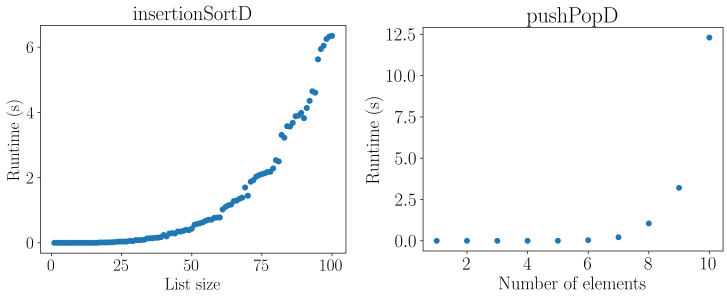

Fig. 9: The runtime of `insertionSortD` and `pushPopD`.

3900X running Linux 6.12.58 using the built-in timing functionality of KiCS2 (version 3.5.0), taking the average of 10 executions.

Asymptotically, `insertionSortD` appears to run in $O(n^3)$ time: a cubic interpolation estimates it well. This asymptotic bound is shared by the original demand translation. However, the constants are much worse: `insertionSortD` takes 6.44 seconds on a 100-element list, while the original demand translation version in Curry is fast enough that its runtime cannot be reliably measured. Meanwhile, the `pushPopD` function takes 12.30 seconds to execute on 10 elements.

This result is not surprising, as the original demand semantics is specialized to computing minimal input demands, but the logical demand translation is built on a general-purpose logic language. In other words, logical demand functions trade off performance for simplicity. However, the performance is still useful for cost analysis. We discuss this in more detail in Section 6.

5 Related Work

Lazy cost analysis. Existing research on lazy cost analysis can be divided into roughly three different approaches. The first approach, represented by work on Iris$^{\$}$ [19, 25], deals with mutable heaps directly, but requires using special programming logics like the separation logic [26]. The second approach, represented by LiquidHaskell [9, 14], uses a graded monad proposed by Danielsson [6] to track computation cost. However, this approach needs to deal with sharing manually. The third approach utilizes alternative semantics, such as clairvoyance semantics [8, 17] and demand semantics [2, 30], that are equivalent to lazy evaluation in terms of computation cost. In contrast to the natural semantics of lazy evaluation [15], these semantics reify computation cost in pure and eager semantics, so we can use them to reason about lazy computation cost without being concerned with mutable heaps, sharing, or our-of-order executions.

Clairvoyance semantics was proposed by Hackett and Hutton [8], who proved its equivalence with Launchbury's semantics. Li *et al.* extended clairvoyance semantics to a typed setting and mechanized it in Rocq, along with a framework for mechanically reasoning computation costs [17]. Demand semantics was proposed

by Bjerner and Holmström [2]. Xia *et al.* later extended it to a typed setting and proved its equivalence to clairvoyance semantics, in forms of cost existence and cost minimality theorems [30]. We rely on these results in our correctness theorems in Section 3. They further use demand semantics to mechanically verify amortized costs and persistence of two of Okasaki's data structures: the banker's queue and the implicit queue [23] in Rocq. Recently, a third semantics, memorist semantics, was proposed by Li *et al.* [16]. Memorist semantics works by tracking all the computation dependencies. As it works differently than clairvoyance semantics and demand semantics, it is unclear if there's a connection we can draw like in this paper.

There has also been work on applying property-based testing to lazy computation cost analysis, such as Foner *et al.* [7] and Lorenzen [18]. We leave it as future work to extend our work to property-based testing frameworks in Curry like EasyCheck [5] and CurryCheck [11].

Functional logic programming. We implemented our examples in Curry, but most of the techniques in this paper are not specific to Curry. Other functional logic languages, such as Verse [1] or Mercury [27], could also have proven sufficient. Because **L** uses eager evaluation, we could have even translated to a pure logic language such as Prolog [12]. However, the search tree traversal employed in this work is Curry-specific [3].

6 Conclusion and Future Work

We have shown that clairvoyance and demand semantics are two sides of the same coin using logic programming as an intermediary. This connection is both theoretically interesting and practically useful: it demonstrates that functional logic programming concepts have value in seemingly unrelated areas. We also obtained an executable clairvoyance semantics and a simpler demand semantics built on functional logic programming semantics.

As discussed in Section 4, our technique is slow with even modestly-sized inputs. Although unfortunate, this may be acceptable for most applications. A major motivation behind this line of research was the desire to find a lightweight representation of demand functions that could be used to test their asymptotic bounds (constants included) *before* trying to formally verify them. For this purpose, it usually suffices to test quite small inputs, where our technique performs quite well in absolute (wall-clock) terms. However, some applications seem to perform much worse than others for reasons that we do not yet understand. Further investigation is left as future work.

Acknowledgement

We thank all FLOPS 2026 reviewers for their constructive and insightful comments. We thank Alex Hodges and Xing Li for their feedback to a draft of this paper. This research idea was initially sparked by a discussion after a remote talk given by Shiwei Weng at Portland State University.

References

1. Augustsson, L., Breitner, J., Claessen, K., Jhala, R., Jones, S.P., Shivers, O., Steele, Jr., G.L., Sweeney, T.: The Verse calculus: A core calculus for deterministic functional logic programming. Proc. ACM Program. Lang. **7**(ICFP), 417–447 (2023). https://doi.org/10.1145/3607845
2. Bjerner, B., Holmström, S.: A composition approach to time analysis of first order lazy functional programs. In: Stoy, J.E. (ed.) Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989. pp. 157–165. ACM (1989). https://doi.org/10.1145/99370.99382
3. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. J. Funct. Log. Program. (2004), http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2004/S04-01/A2004-06/JFLP-A2004-06.pdf
4. Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A new compiler from Curry to Haskell. In: Kuchen, H. (ed.) Functional and Constraint Logic Programming - 20th International Workshop, WFLP 2011, Odense, Denmark, July 19th, Proceedings. Lecture Notes in Computer Science, vol. 6816, pp. 1–18. Springer (2011). https://doi.org/10.1007/978-3-642-22531-4_1
5. Christiansen, J., Fischer, S.: EasyCheck - test data for free. In: Garrigue, J., Hermenegildo, M.V. (eds.) Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4989, pp. 322–336. Springer (2008). https://doi.org/10.1007/978-3-540-78969-7_23
6. Danielsson, N.A.: Lightweight semiformal time complexity analysis for purely functional data structures. In: Necula, G.C., Wadler, P. (eds.) Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008. pp. 133–144. ACM (2008). https://doi.org/10.1145/1328438.1328457
7. Foner, K., Zhang, H., Lampropoulos, L.: Keep your laziness in check. Proc. ACM Program. Lang. **2**(ICFP), 102:1–102:30 (2018). https://doi.org/10.1145/3236797
8. Hackett, J., Hutton, G.: Call-by-need is clairvoyant call-by-value. Proc. ACM Program. Lang. **3**(ICFP), 114:1–114:23 (2019). https://doi.org/10.1145/3341718
9. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate your assets: reasoning about resource usage in liquid haskell. Proc. ACM Program. Lang. **4**(POPL), 24:1–24:27 (2020). https://doi.org/10.1145/3371092
10. Hanus, M.: Functional logic programming: From theory to Curry. In: Voronkov, A., Weidenbach, C. (eds.) Programming Logics - Essays in Memory of Harald Ganzinger. Lecture Notes in Computer Science, vol. 7797, pp. 123–168. Springer (2013). https://doi.org/10.1007/978-3-642-37651-1_6
11. Hanus, M.: CurryCheck: Checking properties of Curry programs. In: Hermenegildo, M.V., López-García, P. (eds.) Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10184, pp. 222–239. Springer (2016). https://doi.org/10.1007/978-3-319-63139-4_13
12. Hanus, M.: From logic to functional logic programs. Theory Pract. Log. Program. **22**(4), 538–554 (2022). https://doi.org/10.1017/S1471068422000187
13. Hughes, J.: Why functional programming matters. Comput. J. **32**(2), 98–107 (1989). https://doi.org/10.1093/comjnl/32.2.98

14. Kastner, J.: liquid-structures (2022), https://github.com/john-h-kastner/liquid-structures, open-source code on GitHub
15. Launchbury, J.: A natural semantics for lazy evaluation. In: Deusen, M.S.V., Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993. pp. 144–154. ACM Press (1993). https://doi.org/10.1145/158511.158618
16. Li, X., Li, Y., Schachte, P., Rizkallah, C.: The memorist tale: Every thunk every cost all at once. In: Programming Languages and Systems - 35th European Symposium on Programming, ESOP 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings. Lecture Notes in Computer Science, Springer (2026)
17. Li, Y., Xia, L., Weirich, S.: Reasoning about the garden of forking paths. Proc. ACM Program. Lang. **5**(ICFP), 1–28 (2021). https://doi.org/10.1145/3473585
18. Lorenzen, A.: Lightweight testing of persistent amortized time complexity in the credit monad. In: Morris, J.G., Xie, N. (eds.) Proceedings of the 18th ACM SIGPLAN International Haskell Symposium, Haskell 2025, Singapore, October 12-18, 2025. pp. 80–93. ACM (2025). https://doi.org/10.1145/3759164.3759351
19. Mével, G., Jourdan, J., Pottier, F.: Time credits and time receipts in Iris. In: Caires, L. (ed.) Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11423, pp. 3–29. Springer (2019). https://doi.org/10.1007/978-3-030-17184-1_1
20. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
21. Moran, A., Sands, D.: Improvement in a lazy context: An operational theory for call-by-need. In: Appel, A.W., Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999. pp. 43–56. ACM (1999). https://doi.org/10.1145/292540.292547
22. Norell, U.: Dependently typed programming in Agda. In: Kennedy, A., Ahmed, A. (eds.) Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009. pp. 1–2. ACM (2009). https://doi.org/10.1145/1481861.1481862, http://dl.acm.org/citation.cfm?id=1481861
23. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
24. Petricek, T.: Evaluation strategies for monadic computations. In: Chapman, J., Levy, P.B. (eds.) Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012. EPTCS, vol. 76, pp. 68–89 (2012). https://doi.org/10.4204/EPTCS.76.7
25. Pottier, F., Guéneau, A., Jourdan, J.H., Mével, G.: Thunks and debits in separation logic with time credits. Proc. ACM Program. Lang. **8**(POPL) (2024), https://hal.science/hal-04238691/file/main.pdf
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings. pp. 55–74. IEEE Computer Society (2002). https://doi.org/10.1109/LICS.2002.1029817
27. Somogyi, Z., Henderson, F.: The design and implementation of Mercury. Joint International Conference and Symposium on Logic Programming (Sep 1996)

28. Tolmach, A.P., Antoy, S.: A monadic semantics for core Curry. In: Brim, L., Grumberg, O. (eds.) 12th International Workshop on Functional and Constraint Logic Programming, WFLP 2003, in connection with RDP'03, Federated Conference on Rewriting, Deduction and Programming, Boulder, Colorado, USA, July 14, 2003. Electronic Notes in Theoretical Computer Science, vol. 86, pp. 16–34. Elsevier (2003). https://doi.org/10.1016/S1571-0661(04)80691-1
29. Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **2**(4), 461–493 (1992). https://doi.org/10.1017/S0960129500001560
30. Xia, L., Israel, L., Kramarz, M., Coltharp, N., Claessen, K., Weirich, S., Li, Y.: Story of your lazy function's life: A bidirectional demand semantics for mechanized cost analysis of lazy programs. Proc. ACM Program. Lang. **8**(ICFP), 30–63 (2024). https://doi.org/10.1145/3674626

A The Evaluation Semantics of B

$$
\begin{aligned}
[\![x]\!](\gamma) &= \gamma(x) \\
[\![\texttt{let } x \texttt{ = } t_1 \texttt{ in } t_2]\!](\gamma) &= [\![t_2]\!](\gamma, x \mapsto [\![t_1]\!](\gamma)) \\
[\![\texttt{true}]\!](\gamma) &= \mathbf{true} \\
[\![\texttt{false}]\!](\gamma) &= \mathbf{false} \\
[\![\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3]\!](\gamma) &= \begin{cases} [\![t_2]\!](\gamma) & \text{if } [\![t_1]\!](\gamma) = \mathbf{true} \\ [\![t_3]\!](\gamma) & \text{otherwise} \end{cases} \\
[\![\texttt{[]}]\!](\gamma) &= \varepsilon \\
[\![t_1 \texttt{ :: } t_2]\!](\gamma) &= \mathbf{cons}([\![t_1]\!](\gamma), [\![t_2]\!](\gamma)) \\
[\![\texttt{foldr(}\lambda x_1 x_2 . t_1\texttt{, } t_2\texttt{, } t_3\texttt{)}]\!](\gamma) &= [\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, [\![t_3]\!](\gamma)) \\
[\![\texttt{lazy } t]\!](\gamma) &= [\![t]\!](\gamma) \\
[\![\texttt{force } t]\!](\gamma) &= [\![t]\!](\gamma) \\
[\![\texttt{tick } t]\!](\gamma) &= [\![t]\!](\gamma)
\end{aligned}
$$

$$
\begin{aligned}
[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, \mathbf{nil}) &= [\![t_2]\!](\gamma) \\
[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, \mathbf{cons}(v_1, v_2)) &= [\![t_1]\!](\gamma, x_1 \mapsto v_1, x_2 \mapsto [\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, v_2))
\end{aligned}
$$

B The Clairvoyance Semantics of B

$$\frac{\gamma(x) = v}{\gamma \,;\, x \Downarrow_0^{\mathbb{C}} v}\text{C-Var} \qquad \frac{\gamma \,;\, t_1 \Downarrow_{c_1}^{\mathbb{C}} v_1 \qquad \gamma, x \mapsto v_1 \,;\, t_2 \Downarrow_{c_2}^{\mathbb{C}} v_2}{\gamma \,;\, \texttt{let } x \texttt{ = } t_1 \texttt{ in } t_2 \Downarrow_{c_1+c_2}^{\mathbb{C}} v_2}\text{C-Let}$$

$$\frac{}{\gamma \,;\, \texttt{true} \Downarrow_0^{\mathbb{C}} \textbf{true}}\text{C-True} \qquad \frac{}{\gamma \,;\, \texttt{false} \Downarrow_0^{\mathbb{C}} \textbf{false}}\text{C-False}$$

$$\frac{\gamma \,;\, t_1 \Downarrow_{c_1}^{\mathbb{C}} \textbf{true} \qquad \gamma \,;\, t_2 \Downarrow_{c_2}^{\mathbb{C}} v}{\gamma \,;\, \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow_{c_1+c_2}^{\mathbb{C}} v}\text{C-If-True} \qquad \frac{\gamma \,;\, t_1 \Downarrow_{c_1}^{\mathbb{C}} \textbf{false} \qquad \gamma \,;\, t_3 \Downarrow_{c_3}^{\mathbb{C}} v}{\gamma \,;\, \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow_{c_1+c_3}^{\mathbb{C}} v}\text{C-If-False}$$

$$\frac{}{\gamma \,;\, \texttt{[]} \Downarrow_0^{\mathbb{C}} \textbf{nil}}\text{C-Nil} \qquad \frac{\gamma \,;\, t_1 \Downarrow_{c_1}^{\mathbb{C}} v_1 \qquad \gamma \,;\, t_2 \Downarrow_{c_1}^{\mathbb{C}} v_2}{\gamma \,;\, t_1 \texttt{ :: } t_2 \Downarrow_{c_1+c_2}^{\mathbb{C}} \textbf{cons}(v_1, v_2)}\text{C-Cons}$$

$$\frac{\gamma \,;\, t_3 \Downarrow_{c_1}^{\mathbb{C}} v_1 \qquad \gamma \,;\, \lambda x_1 x_2 . t_1, t_2, v_1 \Downarrow_{c_2}^{\mathbb{C}\texttt{-foldr}} v_2}{\gamma \,;\, \texttt{foldr}(\lambda x_1 x_2 . t_1\texttt{, } t_2\texttt{, } t_3) \Downarrow_{c_1+c_2}^{\mathbb{C}} v_2}\text{C-Foldr}$$

$$\frac{\gamma \,;\, t \Downarrow_c^{\mathbb{C}} v}{\gamma \,;\, \texttt{lazy } t \Downarrow_c^{\mathbb{C}} \textbf{thunk } v}\text{C-Lazy-Result} \qquad \frac{}{\gamma \,;\, \texttt{lazy } t \Downarrow_0^{\mathbb{C}} \bot}\text{C-Lazy-Undefined}$$

$$\frac{\gamma \,;\, t \Downarrow_c^{\mathbb{C}} \textbf{thunk } v}{\gamma \,;\, \texttt{force } t \Downarrow_c^{\mathbb{C}} v}\text{C-Force} \qquad \frac{\gamma \,;\, t \Downarrow_c^{\mathbb{C}} v}{\gamma \,;\, \texttt{tick } t \Downarrow_{c+1}^{\mathbb{C}} v}\text{C-Tick}$$

C The Demand Semantics of B

$$
\begin{aligned}
\mathbb{D}[\![x]\!](\gamma, a) &= (\{x \mapsto a\}, 0) \\
\mathbb{D}[\![\texttt{let } x \texttt{ = } t_1 \texttt{ in } t_2]\!](\gamma, a) &= (d_1, c_1) \sqcup (d_2, c_2) && \text{where } ((d_2, x \mapsto b), c_2) = \mathbb{D}[\![t_2]\!](\gamma, x \mapsto [\![t \\
& && \text{and } (d_1, c_1) = \mathbb{D}[\![t_1]\!](\gamma, b) \\
\mathbb{D}[\![\texttt{tick } t]\!](\gamma, a) &= (d, c+1) && \text{where } (d, c) = \mathbb{D}[\![t]\!](\gamma, a) \\
\mathbb{D}[\![\texttt{lazy } t]\!](\gamma, \bot) &= (\bot, 0) \\
\mathbb{D}[\![\texttt{lazy } t]\!](\gamma, \textbf{thunk } a) &= \mathbb{D}[\![t]\!](\gamma, a) \\
\mathbb{D}[\![\texttt{force } t]\!](\gamma, a) &= \mathbb{D}[\![t]\!](\gamma, \textbf{thunk } a) \\
\mathbb{D}[\![\texttt{true}]\!](\gamma, a) &= (\bot_\gamma, 0) \\
\mathbb{D}[\![\texttt{false}]\!](\gamma, a) &= (\bot_\gamma, 0) \\
\mathbb{D}[\![\texttt{[]}]\!](\gamma, a) &= (\bot_\gamma, 0) \\
\mathbb{D}[\![t_1 \texttt{ :: } t_2]\!](\gamma, a) &= \mathbb{D}[\![t_1]\!](\gamma, a) \sqcup \mathbb{D}[\![t_2]\!](\gamma, a) \\
\mathbb{D}[\![\texttt{foldr}(\lambda x_1 x_2 . t_1\texttt{, } t_2\texttt{, } t_3)]\!](\gamma, b) &= (d', c) \sqcup \mathbb{D}[\![t]\!](\gamma, b') && \text{where } (g, b', c) = \mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, [\![t
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, \textbf{nil}, b) &= (d, \textbf{nil}, c) && \text{where } (d, c) = \mathbb{D}[\![t_2]\!](\gamma, b) \\
\mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, \textbf{cons}(v_1, v_2), b) &= (g_1 \sqcup g_2, \textbf{cons}(a', d), c_1 + c_2) \\
& \text{where } ((g_1, x_1 \mapsto a', x_2 \mapsto b'), c_1) = \mathbb{D}[\![t_1]\!](\gamma, x_1 \mapsto v_1, x_2 \mapsto [\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, v_2), b) \\
& \text{and } (g_2, d, c) = \mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}'}(t_2, \gamma, v_2, b')
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}'}(t_2, \gamma, v, \bot) &= \bot \\
\mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}'}(t_2, \gamma, v, \textbf{thunk } b) &= (g, \textbf{thunk } d, c) && \text{where } (g, d, c) = \mathbb{D}[\![\lambda x_1 x_2 . t_1]\!]_{\text{foldr}}(t_2, \gamma, v,
\end{aligned}
$$

In the above, $\sqcup$ means "least upper bound." In the first instance, it applies to the approximations of a fixed exact value e. It is then extended to approximation environments: if $g_1, g_2 \prec \gamma$, then $g_1 \sqcup g_2$ is the elementwse least upper bound of g_1 and g_2. Finally, it is extended to pairs (g, c), where $g \prec \gamma$ and $c \in \mathbb{N}$, as

$$(g_1, c_1) \sqcup (g_2, c_2) = (g_1 \cup g_2, c_1 + c_2).$$

If e is an exact value, then $\bot_e$ means the least approximation of e. If γ is an exact environment, then $\bot_\gamma$ means the least environment approximating γ.

$$\bot_{\textbf{tt}} = \textbf{tt} \qquad \bot_{\textbf{false}} = \textbf{false} \qquad \bot_{\textbf{true}} = \textbf{true} \qquad \bot_n = n$$

$$\bot_{(e_1, e_2)} = (\bot_{e_1}, \bot_{e_2}) \qquad \bot_{\textbf{nil}} = \textbf{nil} \qquad \bot_{\textbf{cons}(e_1, e_2)} = \textbf{cons}(\bot_{e_1}, \bot_{e_2})$$

D The Semantics of L

$$\frac{\gamma\,;\,t \Downarrow^{\mathbb{L}} v}{\gamma\,;\,\texttt{thunk}\ t \Downarrow^{\mathbb{L}} \textbf{thunk}\ v}\text{L-Result} \qquad \frac{}{\gamma\,;\,\texttt{undefined} \Downarrow^{\mathbb{L}} \bot}\text{L-Undefined}$$

$$\frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} \textbf{thunk}\ v_1 \qquad \gamma, x \mapsto v_1\,;\,t_2 \Downarrow^{\mathbb{L}} v_2}{\gamma\,;\,\texttt{case}\ t_1\ \texttt{of thunk}\ x\ \texttt{=>}\ t_2\texttt{; undefined =>}\ t_3 \Downarrow^{\mathbb{L}} v_2}\text{L-Case-Result}$$

$$\frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} \bot \qquad \gamma\,;\,t_3 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,\texttt{case}\ t_1\ \texttt{of thunk}\ x\ \texttt{=>}\ t_2\texttt{; undefined =>}\ t_3 \Downarrow^{\mathbb{L}} v}\text{L-Case-Undefined}$$

$$\frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} v \qquad \gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,t_1\ \texttt{==}\ t_2 \Downarrow^{\mathbb{L}} \textbf{true}}\text{L-Eq-True} \qquad \frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} v_1 \qquad \gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v_2 \qquad v_1 \neq v_2}{\gamma\,;\,t_1\ \texttt{==}\ t_2 \Downarrow^{\mathbb{L}} \textbf{false}}\text{L-Eq-False}$$

$$\frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} v_1 \qquad \gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v_2 \qquad v_1 \leq v_2}{\gamma\,;\,t_1\ \texttt{<=}\ t_2 \Downarrow^{\mathbb{L}} \textbf{true}}\text{L-Approx-True}$$

$$\frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} v_1 \qquad \gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v_2 \qquad v_1 \not\leq v_2}{\gamma\,;\,t_1\ \texttt{<=}\ t_2 \Downarrow^{\mathbb{L}} \textbf{false}}\text{L-Approx-False}$$

$$\frac{v : A}{\gamma\,;\,\texttt{free}\ A \Downarrow^{\mathbb{L}} v}\text{L-Free} \qquad \frac{\gamma\,;\,t_1 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,t_1\ \texttt{?}\ t_2 \Downarrow^{\mathbb{L}} v}\text{L-Choice-L} \qquad \frac{\gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,t_1\ \texttt{?}\ t_2 \Downarrow^{\mathbb{L}} v}\text{L-Choice-R}$$

$$\frac{\gamma\,;\,t_3 \Downarrow^{\mathbb{L}} v_1 \qquad \gamma\,;\,\lambda x_1 x_2.t_1, t_2, v_1 \Downarrow^{\mathbb{L}\texttt{-foldr}} v_2}{\gamma\,;\,\texttt{foldr}(\lambda x_1 x_2\texttt{.}t_1\texttt{,}\ t_2\texttt{,}\ t_3) \Downarrow^{\mathbb{L}} v_2}\text{L-Foldr}$$

$$\frac{\gamma\,;\,t_2 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,\lambda x_1 x_2.t_1, t_2, \textbf{nil} \Downarrow^{\mathbb{L}\texttt{-foldr}} v}\text{L-Foldr-Nil} \qquad \frac{\gamma, x_1 \mapsto v_1, x_2 \mapsto \bot\,;\,t_1 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,\lambda x_1 x_2.t_1, t_2, \textbf{cons}(v_1, \bot) \Downarrow^{\mathbb{L}\texttt{-foldr}} v}\text{L-Foldr-Undefined}$$

$$\frac{\gamma\,;\,\lambda x_1 x_2.t_1, t_2, \textbf{thunk}\ v_2 \Downarrow^{\mathbb{L}\texttt{-foldr}} v_3 \qquad \gamma, x_1 \mapsto v_1, x_2 \mapsto \textbf{thunk}\ v_3\,;\,t_1 \Downarrow^{\mathbb{L}} v}{\gamma\,;\,\lambda x_1 x_2.t_1, t_2, \textbf{cons}(v_1, v_2) \Downarrow^{\mathbb{L}\texttt{-foldr}} v}\text{L-Foldr-Result}$$

E Full Case Study on Insertion Sort

A classic fact about lazy evaluation is that one can efficiently find the n smallest elements in a list by first sorting it and then taking a length-n prefix (provided that one uses a sufficiently lazy sorting algorithm). To test this, we first define insertion sort in clairvoyance semantics (`insertionSortC`).

```
insertC :: Ord a => a -> List a -> Tick (List a)
insertC x ys = do
  tick
  fcase ys of
```

```
    Nil -> do
      ys' <- nilC
      return (x :~ ys')
    y :~ ysT' ->
      if x <= y then do
        ysT <- thunk (return ys)
        return (x :~ ysT)
      else do
        ysT'' <- with ysT' (insertC x)
        return (y :~ ysT'')

insertionSortC :: Ord a => List a -> Tick (List a)
insertionSortC xs = do
  tick
  fcase xs of
    Nil -> return Nil
    x :~ xsT' -> do
      ysT' <- with xsT' insertionSortC
      ys' <- force ysT'
      insertC x ys'
```

Next, we define `nminC` as a composition of `takeC` and `insertionSortC`, along with its demand-semantics counterpart `nminD`.

```
nminC :: Ord a => Int -> T (List a) -> Tick (List a)
nminC n xsT = takeC n =<< with xsT insertionSortC

nminD :: (Ord a, Approx a) => Int -> T (List a) -> List a -> Tick (T (List a))
nminD n xsT ysTD | nminC n xsTD =:= Tick (ysTD, c)
                 = Tick (xsTD, c)
  where xsTD = approx xsT
        c free
```

Now, how much does it cost to sort a 3-element list?

```
> insertionSortDG (fromList [3, 2, 1]) (fromList [1, 2, 3]) :: Tick (List Int)
(Tick ((:~ 3 (Thunk (:~ 2 (Thunk (:~ 1 (Thunk Nil))))))),10))
```

If we demand the entire result, we incur a cost of 10 ticks. What if we just want the first element?

```
> someValueWith dfsStrategy $ nminDG 1 (Thunk (fromList [3, 2, 1])) (fromList [1])
    :: Tick (T (List Int))
(Tick ((Thunk (:~ 3 (Thunk (:~ 2 (Thunk (:~ 1 (Thunk Nil)))))))),9))
```

The resulting computation only costs 9 ticks.