Reasoning about the Garden of Forking Paths

YAO LI, University of Pennsylvania, USA
LI-YAO XIA, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

Lazy evaluation is a powerful tool for functional programmers. It enables the concise expression of on-demand computation and a form of compositionality not available under other evaluation strategies. However, the stateful nature of lazy evaluation makes it hard to analyze a program's computational cost, either informally or formally. In this work, we present a novel and simple framework for formally reasoning about lazy computation costs based on a recent model of lazy evaluation: clairvoyant call-by-value. The key feature of our framework is its simplicity, as expressed by our definition of the *clairvoyance monad*. This monad is both simple to define (around 20 lines of Coq) and simple to reason about. We show that this monad can be effectively used to mechanically reason about the computational cost of lazy functional programs written in Coq.

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, monad

ACM Reference Format:
Yao Li, Li-yao Xia, and Stephanie Weirich. 2021. Reasoning about the Garden of Forking Paths. *Proc. ACM Program. Lang.* 5, ICFP, Article 80 (August 2021), 28 pages. https://doi.org/10.1145/3473585

1 INTRODUCTION

Lazy evaluation [Henderson and Morris 1976], or the *call-by-need* calling convention, is a distinguishing feature in some functional programming languages—Haskell being the most notable example. Rather than evaluating eagerly, a lazy evaluator stores computations in a thunk and only evaluates the thunk when the data is needed. This feature avoids unneeded computation and enables better modularity in functional programming [Hughes 1989]. However, with convenience in expressiveness comes challenge in reasoning—especially so for cost analysis—because it's far less obvious if, when, and how much a computation is evaluated. We believe proof assistants can help to reason formally about semantics so intricate and subtle.

However, modeling laziness formally is difficult. The semantics for call-by-need evaluation is more complex than that of call-by-name or call-by-value, which can be described merely through substitution. Traditional presentations [Josephs 1989; Launchbury 1993] of call-by-need semantics are fundamentally stateful, based on heaps that contain thunks which must be updated during evaluation.

Rather than directly dealing with such complexity, we take advantage of a new way of modeling call-by-need: *clairvoyant call-by-value* [Hackett and Hutton 2019]. The key observation of this new model is that although *whether* a term gets evaluated matters, it doesn't matter *when* in

Authors' addresses: Yao Li, liyao@cis.upenn.edu, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA; Li-yao Xia, xialiyao@cis.upenn.edu, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA; Stephanie Weirich, sweirich@cis.upenn.edu, University of Pennsylvania, 3330 Walnut St, Philadelphia, PA, 19104, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2021 Copyright held by the owner/author(s).

2475-1421/2021/8-ART80
https://doi.org/10.1145/3473585

run-time cost analysis. Therefore, instead of storing the computations in a thunk, the clairvoyant call-by-value model makes use of nondeterminism to evaluate the data in one branch and skip evaluation in another. Eventually, one successful branch of evaluation will faithfully model the result and cost of the call-by-need evaluation.

Based on clairvoyant evaluation, we propose a novel framework for reasoning about laziness using the Coq proof assistant, a dependently-typed interactive theorem prover [Coq development team 2021]. Our framework is based on an annotated model similar to that of Danielsson [2008] and of Handley et al. [2020], but our work does not require an explicit notion of laziness to reason about computation costs under lazy evaluation.

We make the following contributions:

- We present the clairvoyance monad, a model of laziness that can be shallowly embedded in Coq by distilling two core features of clairvoyant evaluation: nondeterminism and cost accumulation. One key feature of the clairvoyance monad is its simplicity: it consists of only a handful of basic combinators and can be defined with around 20 lines of code in Coq (Section 3).
- We develop a translation from a lazy calculus to programs in this monad that captures the semantics of clairvoyant call-by-value evaluation. Compared with the denotational semantics of Hackett and Hutton [2019], our translation deals with typed programs, does not rely on domain theory, and accounts for the cost of every nondeterministic execution (Section 4).
- One challenge arising from clairvoyant evaluation is to reason about nondeterminism. We develop dual logics of *over-* and *under-approximations* similar to those of de Vries and Koutavas [2011]; Hoare [1969]; O'Hearn [2020] to reason about lazy programs. We show that formal reasoning of computation cost based on these logics can be done in a *local* and *modular* way (Section 5).
- We demonstrate the usefulness of our technique via case studies of tail recursion that capture the unique characteristics of lazy evaluation (Section 6).

Our paper ends with a comparison of related research in Section 8 and a discussion of future work in Section 9. In the next section, below, we introduce our approach at a high level and present our running example.

2 MOTIVATING EXAMPLE

We start by providing an informal overview of our approach on a small example that exhibits laziness. Consider the following program, written in Gallina, the functional programming language of the Coq proof assistant [Coq development team 2021]:

```
Definition p {a} (n : nat) (xs ys : list a) : list a :=
  let zs := append xs ys in
  take n zs.
```

Gallina can be compiled to use lazy evaluation via extraction to Haskell. The functions append and take in this example are equivalent to their Haskell counterparts, and their definitions are shown in Fig. 1. [1] These examples use Gallina's inductively defined lists, which are a subset of Haskell's list type. Although working with infinite data types is another useful application of lazy evaluation, many algorithms manipulate only finite data structures [Okasaki 1999]. Hence, we believe inductive lists are representative of how lists are used in practice even in Haskell.

[1]For reasons that we will explain in Section 4, these functions are written in A-Normal Form [Sabry and Felleisen 1992], but that doesn't matter so much here.

```
Fixpoint append {a} (xs ys : list a) : list a :=
  match xs with
  | nil => ys
  | cons x xs1 => let zs := append xs1 ys in x :: zs
  end.

(* returns the prefix of xs of length n or xs when n > length xs. *)
Fixpoint take {a} (n : nat) (xs : list a) : list a :=
  match n, xs with
  | O, _ => nil
  | S _, nil => nil
  | S n1, x :: xs1 => let zs := take n1 xs1 in x :: zs
  end.
```

Fig. 1. The pure functional definitions of append and `take`.

To estimate the time it takes to evaluate a program, its cost, we can start by counting the number of steps in some operational semantics, or some proportional quantity. Let us count function calls informally.

Lazy evaluation leads us immediately to an impasse, because it is not even clear what it means to “run” a lazy program. Lazy programs are demand-driven, so we have to specify some model of “demand”. A common working model is that lazy programs will be forced during the evaluation of a whole program, but it is not so practical to reason about the behaviors of arbitrary programs. A more useful approach is to start from a more familiar place: call-by-value. Indeed, programs under the call-by-value evaluation strategy have a relatively straightforward cost model. Laziness adds a twist to it: we might not need all of the result, in which case we allow ourselves to skip some computations.

With that new ability, we face the problem of deciding *which* computations to skip. This decision inherently depends on how much of the overall result will be needed. For concreteness, let us require all of our example list `take n (append xs ys)` to be evaluated. We start by evaluating `append xs ys`, unfolding the program in call-by-value. There are two cases to consider: the length of `xs` may be less than `n`, in which case we will fully evaluate `append xs ys` in `length xs + 1` calls—where the final `nil` takes one call. Or `length xs` may be greater than or equal to `n`, then we can stop after `n` calls, leaving the result of the next call “undefined”. Either way, we will produce some partially defined list `zs` after at most `n` calls. We then let `take n zs` run to the end in at most `n + 1` calls, thus producing all elements of `take n (append xs ys)`, as we demanded initially. In total, that took at most `2 * n + 1` calls. In particular, that cost is independent of the length of `xs` or `ys`. That exemplifies one of the core motivations of laziness: you only pay for what you need.

That idea of “call-by-value with a twist” is made formal by the concept of *clairvoyant evaluation* [Hackett and Hutton 2019].

2.1 Clairvoyant Evaluation

The key to formalize the reasoning above is to view lazy programs as *nondeterministic* programs. Clairvoyant evaluation works in a way similar to nondeterministic automata, which choose one of multiple successor states by *“guessing”* the path to success. In our earlier reasoning, we evaluated `append xs ys` in call-by-value until we decided to stop at a point. *When* to stop was not decided by the state of the program, but by a “guess” based on the clairvoyant knowledge that we would only

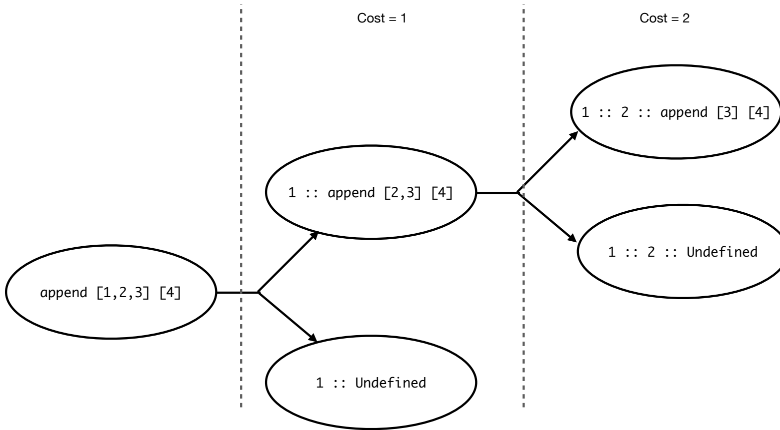

Fig. 2. Parts of the nondeterministic clairvoyant call-by-value evaluation of the append function applied to two lists `[1, 2, 3]` and `[4]`.

need `n` elements in the end. This intuition allows us to define a general semantics that the meaning of a lazy program comprises all of its nondeterministic evaluations and the meaning can be refined later in light of new external information.

The equivalence of clairvoyant evaluation to the natural heap-based definition of laziness [Launchbury 1993] was proved by Hackett and Hutton [2019]: the cost of any execution in clairvoyant call-by-value is an upper bound of the cost in call-by-need, and there is some clairvoyant execution whose cost is actually the same as in call-by-need. In this paper, we carry on with the clairvoyant interpretation.

Taking the append function as an example (recall its definition in Fig. 1), when it makes a recursive call, we fork the evaluation into two branches: in branch (1), we perform the recursive call; and in branch (2), we skip that call. A skipped call yields a placeholder value as a result, which we call $\bot$ or `Undefined`.

Suppose that all future demands only require the first element of the result list, then branch (2) would suffice for offering that result. However, if a future demand requests more than that, branch (2) would fail to proceed because the requested data is `Undefined`. Therefore, branch (2) would get stuck and not yield any result at all. Fortunately, there would still be branch (1) to return the result. Furthermore, in branch (1), the append function may make another recursive call to itself, as long as the first argument list is not `nil`. In that case, this branch would be once again forked into two branches. This is illustrated by Fig. 2.

Although lazy programs are now interpreted nondeterministically, nondeterminism is used in a very controlled manner. The only choices we make are whether to perform a computation or to skip it. This means that the value of a program, if it exists, is still unique in some sense: the only possible changes are that parts of the value are replaced with `Undefined`.

If we know which branch leads to a successful evaluation for the append function, we can just look at that branch and add its cost to obtain the total cost of the program, which gives us a local reasoning methodology. Of course we cannot know this in advance, but there are some reasoning principles that can help.

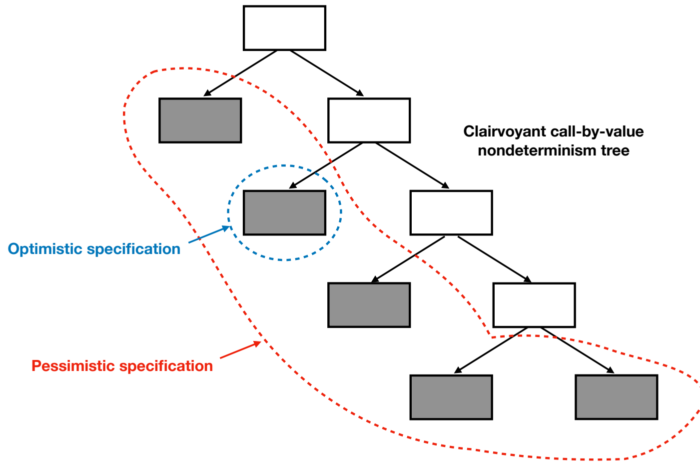

Fig. 3. The relations among a clairvoyant evaluation, a pessimistic specification, and an optimistic specification. The tree in the middle of the figure represents the nondeterminism tree of clairvoyant evaluation. The gray nodes represent the end results of their branches. A pessimistic specification specifies the nodes in the red circle. And an optimistic specification specifies the node in the blue circle.

2.2 A Dual Reasoning Principle

We would like to have a reasoning methodology that is both *local* and *modular*, just like what we would expect from functional programming. This means that we should be able to use some relations to specify the behaviors of each individual function. And when we want to reason about a program, we can just do that by composing the relations of its functions.

We use a dual reasoning principle to achieve the locality and modularity for clairvoyant call-by-value evaluation. First, we have a *pessimistic* specification that describes the behaviors of *all* of the function's nondeterministic branches. The pessimistic specification can offer us an accurate description of functional correctness under call-by-need evaluation. However, the specification is pessimistic because it does not rule out the branches that contain redundant steps and would not appear in an actual call-by-need evaluation.

To be more selective in those branches, we use an optimistic specification that describes the behaviors on a specific branch. The specification is optimistic because it can be used to specify a more accurate bound for the cost under call-by-need evaluation.

Figure 3 shows the relations among a clairvoyant evaluation, a pessimistic specification, and an optimistic specification.

Getting back to append, its pessimistic specification states:

> For *all* the nondeterministic branches of the append function, if the branch evaluates successfully, it will return a cost $c \in [1, \text{length } xs + 1]$.

For simplicity, we omit the functional correctness part of the specification here. The pessimistic specification only specifies a coarse range for append's costs. If we want to reason about our earlier example p, we could not deduce that its cost would never go over n by reasoning about append and take abstractly using their specifications. Instead, we can only deduce that the upper bound of the cost is the length of xs, which may be much larger than n.

For this sort of analysis, we need the optimistic specification of `append`:

> For any number $n \in [1, \text{length } xs]$ (or $n \in [1, \text{length } xs + 1]$ if xs does not contain any undefined part), [2] there *exists* a nondeterministic branch of the `append` function that evaluates successfully and returns a cost $c = n$.

A major difference between the pessimistic and the optimistic specifications is that the latter does not only show a range of costs; it shows what exactly are the possible costs within this range. With the help of this specification, we can "pick" one branch where the possible cost, which might be much smaller than the length of xs, barely suffices for producing a list that `take` needs.

Both the pessimistic and optimistic specifications can be proved on the `append` function. And when reasoning about a larger program like `p`, all we need is to compose the specifications of `append` with the specifications of other functions like `take`.

2.3 The Missing Pieces

So far, we have informally discussed our methodology. The main missing piece to develop in the rest of the paper is to implement this methodology in the formal environment of the Coq proof assistant.

The first step is to associate the pure functional programs with versions that track execution costs and allow nondeterminism. It turns out that we can do that with monads [Moggi 1991; Wadler 1992]. In the next section, we define the *clairvoyance monad*, which distills the main features of clairvoyant evaluation. One attraction of the clairvoyance monad is its simplicity: its core definitions consist of merely 21 nonblank, noncomment lines of code in Coq. Then, we translate pure functions into the clairvoyance monad in Section 4. The final step, in Section 5, is to build a program logic for the clairvoyance monad that enables local and modular formal cost analysis in the style of optimistic and pessimistic specifications.

3 THE CLAIRVOYANCE MONAD

The clairvoyance monad (Fig. 4) is a lightweight abstraction that can express the semantics of instrumented lazy evaluation, suitable for cost analysis. Based on the ideas of clairvoyant evaluation [Hackett and Hutton 2019], its simplicity is largely due to the absence of higher-order state commonly associated with laziness.

To model clairvoyant evaluation, we need a monad that can encode all the following three ingredients: (1) costs, (2) nondeterminism, and (3) failures on some nondeterministic branches. The clairvoyance monad is the simplest monad that meets the criterion.

A computation in the clairvoyance monad, of type `M a`, nondeterministically yields a value `v : a` after some time `n`. A computation is defined as a set of such pairs `(v, n)`, encoded in Coq as a predicate `a -> nat -> Prop`. The `return` of the monad yields the given value `v` with cost 0; it is a set containing only `(v, 0)`. The `bind` of the monad sequences computation by getting a result `(x, nx)` from the first operand `u`, and then feeding the value `x` to the continuation `k`, which then yields another result `(y, ny)`. The overall result is that latter value paired with the total cost `(y, nx + ny)`.

To track time, `tick` [Moran and Sands 1999] is a computation with unit cost. This design follows the same rationale as Danielsson [2008]: explicit ticks make the library lightweight and flexible to experiment with different cost models. For a given cost model, one can ensure that ticks are added consistently by an automatic translation. We present such a translation in Section 4.

[2] When xs does not contain any undefined part, `append` might go over the entire list and take one extra cost pattern matching on `nil`. This is the only case the cost of `append` will be bigger than the length of xs.

```
(* A computation that produces a value of type "a" after some number of ticks. *)
Definition M (a : Type) : Type := a -> nat -> Prop.

(* A computation that takes no time and yields a single value. *)
Definition ret {a} (v : a) : M a :=
  fun y n => (y, n) = (v, 0).

(* Sequence two computations and add their time. *)
Definition bind {a b} (u : M a) (k : a -> M b) : M b :=
  fun y n => exists x nx ny, u x nx /\ k x y ny /\ n = nx + ny.

(* A computation with unit cost. *)
Definition tick : M unit :=
  fun _ n => n = 1.

(* A thunk: either a known value or unused. *)
Inductive T (a : Type) : Type :=
| Thunk (x : a)
| Undefined.

(* Store a computation without evaluating it (zero cost). *)
Definition thunk {a} (u : M a) : M (T a) :=
  fun t n => match t with
             | Thunk v => u v n
             | Undefined => n = 0
             end.

(* Either continue computation with the value of a thunk or fail. *)
Definition forcing {a b} (t : T a) (f : a -> M b) : M b :=
  match t with
  | Thunk v => f v
  | Undefined => fun _ _ => False
  end.

(* Force a thunk. *)
Definition force {a} (t : T a) : M a := forcing t ret.
```

Fig. 4. Core definitions of the clairvoyance monad `M`.

The type of thunks `T` is structurally an `option` type. A thunk is either a known value, under the `Thunk` constructor, or it is `Undefined`. `Undefined` thunks are placeholders introduced when a computation is "skipped," because its result won't be needed.

For "laziness," we add two operations to create and force thunks. Intuitively, the `thunk` function stores a computation of type `M a` without evaluating it, and yields a *thunk*: a reference to that stored computation, of type `T a`. The `forcing` function looks up that reference to evaluate the corresponding computation and passes it to the continuation. This result is also stored in place of the computation, so that subsequent uses of `force` will not recompute the result.

$$\frac{\Gamma \vdash \texttt{t : M a} \qquad \Gamma, (\texttt{x : a}) \vdash \texttt{s : M b}}{\Gamma \vdash \texttt{let! x := t in s : M b}}\,\texttt{let!} \qquad \frac{\Gamma \vdash \texttt{t : M a} \qquad \Gamma, (\texttt{xA : T a}) \vdash \texttt{s : M b}}{\Gamma \vdash \texttt{let\~{} xA := t in s : M b}}\,\texttt{let\~{}}$$

$$\frac{\Gamma \vdash \texttt{f : a -> M b} \qquad \Gamma \vdash \texttt{xA : T a}}{\Gamma \vdash \texttt{f \$! xA : M b}}\,(\texttt{\$!})$$

Fig. 5. Typing rules for `let!`, `let~`, and `$!`

In the clairvoyance model, `thunk`'s implementation nondeterministically chooses between (1) running the computation, yielding any one of its results in a `Thunk`, and (2) skipping it, yielding an `Undefined` result at no cost. The set of possible outcomes is implemented as a predicate: it accepts any pair `(Thunk v, n)` such that `(v, n)` is accepted by the given computation `u` and the pair `(Undefined, 0)`.

The `forcing` operation accesses the result stored in a thunk and passes it to a continuation. If there is indeed a value `Thunk v`, then `v` is the result, and we just pass it to the continuation `k`. We do not need to add any costs in this step: we already paid the cost of computing `v` on the thunk's creation. If the thunk is `Undefined`, then the computation fails: it has no result, as denoted by the empty set. Note that the only way to fail among the above five combinators is to use `forcing` and that `thunk` is the only way to produce thunks to apply `forcing` to. In spite of this underlying potential for failure, computations definable with these combinators always have at least one successful execution by never skipping a `thunk`. In that sense, our combinators adequately model a total language.

The empty computation `fun _ _ => False : M a` could also be added to the core definitions. In this paper, we will stick to total functional programming [Turner 2004].

When programming, we also rely on Coq notations for a few well-known monadic operations in addition to these core definitions. The infix notation ">>" abbreviates `bind` with a constant continuation:

```
Notation "t >> s" := (bind t (fun _ => s)).
```

In the clairvoyance monad, a common idiom is `tick >> t` to increase the cost of `t` by one.

Functions whose arguments are thunks are called *lazy*, in the sense that their arguments may not always be defined. Otherwise they are *eager*. Let us define the following notations, wrapping the monadic `bind` in more familiar syntax, akin to do-notation in Haskell and overloaded `let` in OCaml. The infix `$!` is named after a standard Haskell operator which makes a function strict. For reference, typing rules for these constructs are given in Fig. 5.

```
Notation "let! x  := t in s" := (bind t (fun x => s)).
Notation "let~ xA := t in s" := (bind (thunk t) (fun xA => s)).
Notation "f $! xA" := (forcing xA f).
```

We thus view the combinator `bind` as an "eager" `let!` construct, where the bound variable `x` is the result of the computation `t`. In contrast, a composition of `bind` and `thunk` provides a "lazy" `let~`, where `xA` is a thunk for the "delayed" computation `t`. In this paper, variables of "lifted" types `T a` will be marked with a suffix `-A`, to constrast with variables of "unlifted" types `a`.

The definition of `T` has two important features. First, a thunk is merely a value, not an indirect location in a "heap" as it would be in natural semantics [Launchbury 1993]; this is key to the simplicity of our definitions. Second, the type constructor `T` can be used in inductive type declarations

and plays well with the strict positivity condition imposed on them. [3] This will be essential in the translation of recursive types in Section 4.

In the clairvoyance model, it is useful to think of thunks as a way to construct *approximations* [Scott 1976]. The type `T a` "lifts" a type `a` with an `Undefined` value which approximates all values of type `a`. Recursive types may contain nested thunks, thus defining rich domains of approximations. In the monad `M`, we view a lazy computation as producing an approximation of some pure, complete result; more precise approximations are more costly to compute. That structure will be made explicit in Section 5.

Remark. The monad `M` coincides with the writer monad transformer [Liang et al. 1995] applied to the powerset monad `_ -> Prop`. This observation crisply summarizes the orthogonal roles of nondeterminism and accounting for time in the clairvoyance monad.

4 TRANSLATION

The clairvoyance monad provides us with an explicit model of laziness. To reason about the cost of programs in a lazy language—where laziness is implicit—we make computations explicit by translating them into monadic programs. We present such a translation in this section. We will then introduce a framework for reasoning about computations in the clairvoyance monad in the next section.

Our source language is a total lazy calculus with folds, enabling structural recursion. In practice, our programs are embedded in Gallina, a pure and total language which does not prescribe an evaluation strategy—multiple evaluation strategies are actually available. Hence, Gallina terms have no inherent notion of cost. Our monadic translation defines a second embedding that determines a lazy evaluation strategy, allowing us to reason about the cost of lazy functional programs.

Totality is arguably not a strong limitation for implementing many algorithms: in the context of complexity analysis, termination is a necessary condition for defining the cost of an algorithm.

In this section, we formalize the source language and its translation to Gallina. Our examples of source programs are embedded in Gallina, and we currently translate them by hand. This formalization will serve as the basis for a future implementation of a translation from Haskell.

4.1 Formal Monadic Translation

We define a translation from a simply typed lambda calculus to Gallina using the monad `M` and the definitions in Fig. 4. The syntax of the source calculus is summarized in Fig. 6. A primitive type of lists serves to illustrate how to translate algebraic data types, with structural recursion modeled by a FOLDR operator. This calculus is in A-normal form [Sabry and Felleisen 1992] to streamline the monadic translation by confining the bookkeeping of thunks to LET and FOLDR.

In the translation of types (Fig. 7), source function types $\tau_1 \longrightarrow \tau_2$ are translated to target function types $\mathtt{T}\ [\![\tau_1]\!] \longrightarrow \mathtt{M}\ [\![\tau_2]\!]$. The argument is wrapped in a thunk, so functions may be defined on undefined inputs. And the result is, of course, a computation. The type of lists is translated to an inductive type where fields are wrapped in a thunk `T`. This type thus represents partially defined lists, which can be seen as *approximations* of actual lists.

In the translation of terms (Fig. 8), a well-typed term $t : \tau$ is translated to a Gallina term $[\![t]\!] : \mathtt{M}\ [\![\tau]\!]$. We pun source variables $x : a$ as target variables $x : \mathtt{T}\ [\![a]\!]$. A `tick` is added uniformly in the interpretation of every non-value construct—this follows Hackett and Hutton [2019]: it is assumed that those constructs will be implemented in constant time. In our examples, we will simplify ticks further, as discussed in Section 4.4.

[3] https://coq.inria.fr/refman/language/core/inductive.html#strict-positivity

$$
\begin{array}{lll}
\textit{types} & \tau & ::= \tau \longrightarrow \tau \mid \text{LIST}\ \tau \mid \text{UNIT} \\
 & x, y, z & \in \textit{variables} \\
\textit{terms} & t, u & ::= x \mid \lambda x.\, t \mid t\ x \mid \text{LET}\ x\ =\ t\ \text{IN}\ u \\
 & & \quad \mid \text{NIL} \mid \text{CONS}\ x\ y \mid \text{FOLDR}\ t\ (\lambda x\, y.t)\ z
\end{array}
$$

$$
\frac{\Gamma \vdash x : \text{LIST}\ \tau_1 \qquad \Gamma \vdash t_l : \tau_2 \qquad \Gamma,\ y_1 : \tau_1,\ y_2 : \tau_2 \vdash t_n : \tau_2}{\Gamma \vdash \text{FOLDR}\ t_l\ (\lambda y_1 y_2.t_n)\ x : \tau_2}\ \text{FOLDR}
$$

Fig. 6. Syntax and typing rules for FOLDR

$$
\begin{aligned}
[\![\tau_1 \longrightarrow \tau_2]\!] &= \mathtt{T}\ [\![\tau_1]\!] \longrightarrow \mathtt{M}\ [\![\tau_2]\!] \\
[\![\text{LIST}\ \tau]\!] &= \mathtt{listA}\ [\![\tau]\!]
\end{aligned}
$$

```
Inductive listA (a : Type) : Type :=
  NilA | ConsA (x1 : T a) (x2 : T (listA a)).
```

Fig. 7. Monadic type translation on the left; definition of `listA` on the right

$$
\begin{aligned}
[\![\text{LET}\ x\ =\ t\ \text{IN}\ u]\!] &= \mathtt{tick} >> \mathtt{let\sim}\ x := [\![t]\!]\ \mathtt{in}\ [\![u]\!] \\
[\![x]\!] &= \mathtt{tick} >> \mathtt{force}\ x \\
[\![\lambda x.t]\!] &= \mathtt{ret}\ (\mathtt{fun}\ x \Rightarrow [\![t]\!]) \\
[\![t\ x]\!] &= \mathtt{tick} >> \mathtt{let!}\ \mathtt{f} := [\![t]\!]\ \mathtt{in}\ \mathtt{f}\ x \\
[\![\text{NIL}]\!] &= \mathtt{ret\ NilA} \\
[\![\text{CONS}\ x\ y]\!] &= \mathtt{ret}\ (\mathtt{ConsA}\ x\ y) \\
[\![\text{FOLDR}\ t_l\ (\lambda y_1 y_2.t_n)\ x]\!] &= \mathtt{foldrA}\ [\![t_l]\!]\ (\mathtt{fun}\ y_1\ y_2 \Rightarrow [\![t_n]\!])\ x
\end{aligned}
$$

Fig. 8. Monadic term translation

Types guide the design of the term translation. The source `Let` corresponds to our lazy `let~`, creating thunks, while variable expressions $[\![x]\!] : \mathtt{M}\ [\![\tau]\!]$ force the thunk denoted by the variable $x : \mathtt{T}\ [\![\tau]\!]$.

The translation of FOLDR is defined in Fig. 9. A `tick` happens at every recursive call. And recursive calls are thunked using the `let~` construct, so that they may remain unevaluated if the `c` computation doesn't need them.

Recursion introduces a wrinkle in our translation. Generally, function arguments $x : a$ are lifted to $x : \mathtt{T}\ [\![a]\!]$. However, recursive definitions in Coq must take an argument whose outer type constructor is defined using recursion, `listA a`, unlike the type of thunks `T (listA a)`. Thus the translated `foldrA` is merely a wrapper around the recursive function `foldrA'` where most of the work happens. Moreover, in `foldrA'`, the subterm `x2` is forced in continuation-passing style, using `forcing` (under the notation `$!`), in order to ensure that the recursive call to `foldrA'` is syntactically applied to a subterm of the initial list `x'`.

```
Fixpoint foldrA' {a b} (n : M b) (c : T a -> T b -> M b) (x' : listA a) : M b :=
  tick >>
  match x' with
  | NilA => n
  | ConsA x1 x2 =>
    let~ y2 := foldrA' n c $! x2 in
    c x1 y2
  end.

Definition foldrA {a b} (n : M b) (c : T a -> T b -> M b) (x : T (listA a)) : M b :=
  foldrA' n c $! x.
```

Fig. 9. Definition of the `foldrA` function used in the translation of FOLDR

4.2 Equivalence with Clairvoyant Call-By-Value

Our translation $[\![\cdot]\!]$ is a cost-aware denotational semantics for the simply-typed lambda calculus. Formally, this semantics $[\![t]\!]$ is parameterized by an *environment* ρ that maps the free variables of t to semantic values. The denotation $[\![t]\!](\rho) : \mathtt{M}\ [\![\tau]\!]$ is thus a set of cost-value pairs (n, v), with $n : \mathbb{N}$ and $v : [\![\tau]\!]$.

We have proved the equivalence between our denotational semantics and *clairvoyant call-by-value* evaluation, the operational semantics of laziness introduced by Hackett and Hutton, which itself was proved equivalent to the natural semantics of Launchbury [1993]. Clairvoyant call-by-value evaluation is defined as an inductive relation $t, h \Downarrow_n^{\mathrm{CV}} u, h'$ expressing that the term t with an initial *heap* h evaluates, with cost n, to a *value term* u and a final heap h'. To state this equivalence, we extend our denotation function $[\![\cdot]\!]$ to these auxiliary syntactic constructs: a heap h, which maps variables to terms, denotes an environment $[\![h]\!]$, and a (syntactic) value term $u : \tau$, in an environment ρ, denotes a (semantic) value $[\![u]\!](\rho) : [\![\tau]\!]$.

We can now state an equivalence between our denotational semantics $[\![\cdot]\!]$ and the operational semantics $\Downarrow^{\mathrm{CV}}$:

THEOREM 4.1. *For any well-typed term t and heap h, and for any value-cost pair (v, n), the following propositions are equivalent.*

(1) $(v, n) \in [\![t]\!]([\![h]\!])$.
(2) There exists u and h' such that $t, h \Downarrow_n^{\mathrm{CV}} u, h'$ and $v = [\![u]\!]([\![h']\!])$.

The forward direction, (1) implies (2), states the *adequacy* theorem: the denotational semantics $[\![\cdot]\!]$ is a subset of the operational semantics $\Downarrow^{\mathrm{CV}}$. Conversely, the backward direction, (2) implies (1), states our *soundness* theorem: all evaluations by the operational semantics produce denoted cost-values. Together, these results prove that our semantics is equivalent to the operational semantics of Hackett and Hutton.

We have formalized both the denotational semantics $[\![\cdot]\!]$ and the operational semantics $\Downarrow^{\mathrm{CV}}$ and proved the equivalence theorem in Coq [Li et al. 2021]. The proof, which proceeds by induction on n for adequacy and on derivations of $\Downarrow^{\mathrm{CV}}$ for soundness, is straightforward thanks to the simplicity of the language, notably excluding general recursion. Most of the work is devoted to relating mutable heaps h in the operational semantics $\Downarrow^{\mathrm{CV}}$ to environments ρ in our denotational semantics $[\![\cdot]\!]$.

In addition to an operational semantics $\Downarrow^{\mathrm{CV}}$, Hackett and Hutton [2019] also presented a denotational cost semantics, which we can compare to ours. First, the cost semantics of Hackett and Hutton is defined for an untyped recursive calculus and our translation is defined for a typed

```
Definition impl3 {a b c} (P P' : a -> b -> c -> Prop) : Prop :=
  forall x y z, P x y z -> P' x y z.

Inductive Fix {a b} (gf : (a -> M b) -> (a -> M b)) x y n : Prop :=
| MkFix (self : a -> M b) : impl3 self (Fix gf) -> gf self x y n -> Fix gf x y n.
```

Fig. 10. Possible fixpoint combinator in the monad `M`

calculus with folds—guaranteeing termination. However, since the clairvoyance monad is based on the powerset monad `_ -> Prop`, we can also define a fixpoint operator (an example is Fig. 10):

```
Fix : ((a -> M b) -> (a -> M b)) -> (a -> M b)
```

Such an operator could be used for the denotational semantics of a general recursive lazy language, but at the cost of a more complex equivalence theorem. The issue is that the unfolding lemma `Fix F <-> F (Fix F)` assumes the monotonicity of `F`, adding a significant burden to using that operator. Without using `Fix`, we have only modelled a total language, as the source language we considered above is intended to be a subset of Gallina. Many algorithms, in the functional programming literature especially, are defined using various forms of structural recursion, so they can be embedded in our framework.

Second, the denotations of Hackett and Hutton are cost-value pairs that inhabit a lattice to handle general recursion; they handle nondeterminism by joining all executions together. However, in the denotation of Let, the cost of evaluating the binding is discarded if the body of the Let does not depend strictly on the binding. In comparison, our semantics models computation using sets of pairs, so the cost of every nondeterministic path is preserved.

Key to the simplicity of our approach, the core operations of our clairvoyance monad (Section 3) are operations on sets; these operations do not rely on an abstract lattice structure for values. In exchange, our semantics is less well-behaved: `let~` expressions with unused thunks generate spurious approximations. We present a dual logic that disregards such uninformative approximations in Section 5.

4.3 An Example

We show our translation in action on the example of `append` and `take`, illustrating a few pragmatic tweaks to our formalization above. The source program with pure functions in Fig. 1 is translated into the monadic program in Fig. 11. For simplicity, we retain the use of fixpoints instead of representing all recursion with `foldrA`.

These definitions use the `listA` type from Fig. 7. This type is the corresponding *approximation type* for Gallina's native `list`, and wraps every field in the thunk type constructor `T`.

The translation of recursive functions follows a similar structure to the definition of `foldrA` in the previous section, since `append` and `take` are in fact specialized list folds (`foldr`): their translations are wrappers for the recursive `append_` and `take_` where pattern-matching happens, and the recursive calls are guarded by thunks.

We keep the primitive representation of certain types, such as `nat` in the definition of `take`, instead of using its Peano representation. The main reason to do so is that it makes the resulting program simpler by denoting "primitive" operations more directly. Although this is generally unsound for a language with pervasive laziness, this issue could be palliated by using a strictness analysis to ensure that variables of that type are never instantiated with $\bot$. Alternatively, we could consider a source language where both lifted and unlifted types coexist—Haskell is actually such

```
Fixpoint append_ {a : Type} (xs' : listA a) (ys : T (listA a)) : M (listA a) :=
  tick >>
  match xs' with
  | NilA => force ys
  | ConsA x xs1 =>
    let~ t := (fun xs1' => append_ xs1' ys) $! xs1 in
    ret (ConsA x t)
  end.

Definition appendA {a : Type} (xs ys : T (listA a)) : M (listA a) :=
  (fun xs' => append_ xs' ys) $! xs.

Fixpoint take_ {a : Type} (n : nat) (xs' : listA a) : M (listA a) :=
  tick >>
  match n, xs' with
  | O, _ => ret NilA
  | S _, NilA => ret NilA
  | S n1, ConsA x xs1 =>
    let~ t := take_ n1 $! xs1 in
    ret (ConsA x t)
  end.

Definition takeA {a : Type} (n : nat) (xs : T (listA a)) : M (listA a) :=
  take_ n $! xs.

Definition pA {a} (n : nat) (xs ys : T (listA a)) : M (listA a) :=
  tick >>
  let~ t := appendA xs ys in
  takeA n t.
```

Fig. 11. The translated code of append and take from the pure version of Fig. 1.

a language, although unlifted types are not commonly used because GHC's strictness analysis is often good enough to enable optimizations.

The append function, of type `list a -> list a -> list a`, is translated to its approximate version appendA, of type `T (listA a) -> T (listA a) -> M (listA b)`. In other words, the arguments are put under thunks `T`, and the result is produced by an explicit computation `M`. This differs slightly with our formal translation where we simply translated a term $t : \tau$ to $[\![t]\!] : \texttt{M}\ [\![\tau]\!]$. We can do away with that outer `M` because, typically, top-level functions are values.

Finally, we translate the bodies of the functions. To match the syntax of Fig. 6, we sequentialize expressions to ANF [Sabry and Felleisen 1992] (if they are not already in this form), so that every computation happens at a Let binding. We then translate the ANF program to a monadic program, following Fig. 8.

With the translated code, we can now formally analyze its cost. We will show how we specify its cost and how we reason about that in Section 5.

4.4 Simplifying Ticks

In our examples, we have simplified the translated code further to only keep a single `tick` at the head of source function bodies. This incurs a change to the cost of computations bounded by a multiplicative constant of the original cost. Considering that those costs are purely abstract quantities to begin with, this seems an acceptable trade-off to make the translated code more readable.

This simplification can be broken down in two steps. First, apply the following rewrite rules to "float up" every `tick` in every subexpression:

```
bind t (fun x => tick >> k x) = (tick >> bind t k)

thunk (tick >> u) <= (tick >> thunk u)
```

where the inequality `<=` means in this context that every execution `(x, nR)` on the right-hand side corresponds to an execution `(x, nL)` on the left-hand side with the same result but a lower or equal cost `nL <= nR`. [4] That rewriting may increase the cost of programs, which is fine since we are eventually most interested in finding upper bounds on that cost. Second, once all ticks are as high in the program as they can be, we replace all consecutive ticks with a single one. The resulting "speed-up" of the computations is bounded by a constant multiplicative factor equal to the longest chain of ticks substituted that way.

5 FORMAL REASONING

A guiding principle in designing our methodology is to have reasoning rules that are both *local* and *modular*. By local, we mean that we can reason about each function independently; and by modular, we mean we can reason about the whole program by composing the results of reasoning about its parts.

However, in doing so we face a challenge: clairvoyant call-by-value evaluation is an *over-approximation* of call-by-need evaluation: it contains nondeterministic branches that would not appear in an actual call-by-need evaluation. Therefore, to reason precisely about call-by-need execution, we need reasoning rules that are *general* enough to contain many nondeterministic results, but also *selective* enough to prune nondeterministic branches that contain redundant steps.

We address this challenge with a dual specification methodology. For *generality*, a *pessimistic* specification talks about the behaviors on all nondeterministic branches. For *selectiveness*, an *optimistic* specification describes the behavior of specific branches. [5]

5.1 The Optimistic and Pessimistic Specifications

The definitions of the pessimistic specification and the optimistic specification are shown in Fig. 12. Both are parameterized by a *specification relation* `r : a -> nat -> Prop` which specifies a set of desired values and costs. A pessimistic specification states that all nondeterministic branches of the program `u` satisfy the relation `r`. On the other hand, the optimistic specification requires the existence of *at least one* nondeterministic branch satisfying the relation `r`.

We use the following notations to denote these two kinds of specifications: [6]

```
Notation " u {{ r }} " := (pessimistic u r).
Notation " u [[ r ]] " := (optimistic  u r).
```

[4] The equality should be interpreted as extensional equality. We have formally proved these rewrite rules in Coq [Li et al. 2021].

[5] All the theorems discussed in this section have been formally proved in Coq and they are available in our artifact [Li et al. 2021].

[6] We omit the Coq notation levels in the code.

```
Definition pessimistic {a} (u : M a) (r : a -> nat -> Prop) : Prop :=
  forall x n, u x n -> r x n.

Definition optimistic {a} (u : M a) (r : a -> nat -> Prop) : Prop :=
  exists x n, u x n /\ r x n.
```

Fig. 12. The definitions of the pessimistic and optimistic specifications.

$$\frac{r\ x\ 0}{(\texttt{ret}\ x)\ \{\{\ r\ \}\}}\texttt{ret} \qquad \frac{u\ \{\{\ \lambda x\ n.(k\ x)\ \{\{\ \lambda y\ m.r\ y\ (n+m)\ \}\}\ \}\}}{(\texttt{bind}\ u\ k)\ \{\{\ r\ \}\}}\texttt{bind} \qquad \frac{r\ \texttt{tt}\ 1}{\texttt{tick}\ \{\{\ r\ \}\}}\texttt{tick}$$

$$\frac{u\ \{\{\ r\ \}\} \qquad \forall x\ n, r\ x\ n \longrightarrow r'\ x\ n}{u\ \{\{\ r'\ \}\}}\text{monotonicity}$$

$$\frac{r\ \mathsf{Undefined}\ 0 \qquad u\ \{\{\ \lambda x.r\ (\mathsf{Thunk}\ x)\ \}\}}{(\texttt{thunk}\ u)\ \{\{\ r\ \}\}}\texttt{thunk} \qquad \frac{\forall x, t = \mathsf{Thunk}\ x \longrightarrow (k\ x)\ \{\{\ r\ \}\}}{(k\ \$!\ t)\ \{\{\ r\ \}\}}\texttt{forcing}$$

$$\frac{u\ \{\{\ r\ \}\} \qquad u\ \{\{\ r'\ \}\}}{u\ \{\{\ \lambda x\ n.r\ x\ n\ \wedge\ r'\ x\ n\ \}\}}\text{conjunction}$$

Fig. 13. Reasoning rules for pessimistic specifications.

$$\frac{r\ \mathsf{Undefined}\ 0\ \vee\ u\ [[\ \lambda x.r\ (\mathsf{Thunk}\ \ x)\ ]]}{(\texttt{thunk}\ u)\ [[\ r\ ]]}\texttt{thunk} \qquad \frac{t = \mathsf{Thunk}\ x \qquad (k\ x)\ [[\ r\ ]]}{(k\ \$!\ t)\ [[\ r\ ]]}\texttt{forcing}$$

$$\frac{u\ \{\{\ r\ \}\} \qquad u\ [[\ r'\ ]]}{u\ [[\ \lambda x\ n.r\ x\ n\ \wedge\ r'\ x\ n\ ]]}\text{conjunction}$$

Fig. 14. Reasoning rules for optimistic specifications. We omit the rules for `ret`, `bind`, and `tick` as well as the monotonicity rule because these rules have the same forms as those of the pessimistic specifications.

5.2 Reasoning Rules

We can define a set of reasoning rules for the pessimistic specification and the optimistic specification, respectively. For each kind of specification, we build some reasoning rules for the five basic monadic combinators (`ret`, `bind`, `tick`, `thunk`, and `forcing`) described in Section 3. We can then reason about our programs purely based on these reasoning rules plus a monotonicity rule.

Figure 13 shows the reasoning rules for pessimistic specifications. `ret x` satisfies the pessimistic specification `r` if the result `x` and the cost 0 are in the set of `r`. `bind u k` satisfies the pessimistic specification `r` if all the results of `u` can be composed with the continuation `k` such that all the final results are in the set of `r`. A `tick` satisfies the pessimistic specification `r` if `tt` (the only value of the `unit` data type in Coq) and 1 are in the set of `r`. We also need a monotonicity rule to relax the pessimistic specification relation and a conjunction rule to combine pessimistic specifications.

The term `thunk u` satisfies the pessimistic specification `r` if both nondeterministic branches forked off from it satisfy the relation `r`. The `forcing` rule requires that its continuation `k` satisfies the relation `r` when applied to the value contained in a `Thunk`; in the case that there is no defined value within the thunk (*i.e.,* forcing an `Undefined`), the pessimistic specification is vacuously satisfied because the computation branch fails.

Figure 14 shows the reasoning rules for optimistic specifications. We omit the rules for the `ret`, `bind`, and `tick` operators and the monotonicity rule here because they have the same form as those of the pessimistic specification. There are two ways to give an optimistic specification for `thunk` terms, corresponding to selecting one of the two different nondeterministic branches that forked off from the `thunk`. In the branch where the computation is skipped, we only need to show that `Undefined` and 0 are in the relation `r`. In the branch where the computation is evaluated, we show that there exists a result in the computation `u` such that wrapping it in a `Thunk` constructor satisfies the relation `r`. The `forcing` rule requires its argument to be a defined value because forcing an `Undefined` results in failure. When reasoning about a program, we need to select the proper optimistic rule at `thunk`s so that forcing an `Undefined` value never happens.

The conjunction rule for the optimistic specification is also slightly different because its premises require both a pessimistic specification and an optimistic specification.

5.3 Approximations

Before showing how we use both the pessimistic and the optimistic specifications for reasoning about lazy programs, we need to answer this question: in what sense does an approximation function implement a pure function?

Recall the approximation types and pure types discussed in Section 4.3. We would like to base our specification on pure types, as this is what we normally write as functional programs. On the other hand, our implementation in the clairvoyance monad uses approximation types.

We can connect these approximation and pure types together. First observe that we can inject pure types into partial types by thunking each subterm. We call the result an *exact* approximation because it constructs an approximation which represents *exactly* the original list.

```
Definition exact : list a -> listA a.
```

We cannot go the opposite way with a function, since approximations generally contain less information than a full list. Instead, we generalize `exact` as a relation `is_approx xsA xs` between a pure list `xs` on the right and any of its approximations on the left. A notation turns it into an infix operator with syntax inspired by Haskell.

```
Definition is_approx : listA a -> list a -> Prop.
Infix "`is_approx`" := is_approx.
```

Approximations themselves are partially ordered, when the second is at least as defined as the first. We also use infix notation for this relation.

```
Definition less_defined : listA a -> listA a -> Prop.
Infix "`less_defined`" := less_defined.
```

In our running example using lists, we also simplify things by using the same type `a` as the type of elements for pure lists `list a` and list approximations `listA a`.

More generally, we want to overload the function `exact` and relations `is_approx` and `less_defined`, so that (1) their names can be reused for user-defined data types, (2) they are automatically lifted through the thunk type constructor `T`.

Some properties describe and relate these three relations formally. These relations must be defined and their properties must be proved for every user-defined approximation type; those

```
Theorem appendA_correct_partial {a} :
  forall (xs ys : list a) (xsA ysA : T (listA a)),
    xsA `is_approx` xs -> ysA `is_approx` ys ->
    (appendA xsA ysA) {{ fun zsA _ => zsA `is_approx` append xs ys }}.

Theorem appendA_correct_pure {a} :
  forall (xs ys : list a) (xsA ysA : T (listA a)),
    xsA = exact xs -> ysA = exact ys ->
    (appendA xsA ysA) [[ fun zsA _ => zsA = exact (append xs ys) ]].
```

Fig. 15. Definitions of *partial* functional correctness and *pure* functional correctness.

propositions and their proofs (which we omit) follow a common structure, so we conjecture that they can be automated.

The `less_defined` relation is an order relation.

```
Proposition less_defined_order : Order less_defined.
```

The set of approximations for a pure value is downward closed.

```
Proposition approx_down :
  xsA `less_defined` ysA -> ysA `is_approx` xs -> xsA `is_approx` xs.
```

The list `xsA` is an approximation of `xs` if and only if `xsA` is less defined than the exact approximation of `xs`.

```
Proposition approx_exact : xsA `is_approx` xs <-> xsA `less_defined` (exact xs).
```

Exact approximations are maximal elements for the `less_defined` order.

```
Proposition exact_max : exact xs `less_defined` xsA -> exact xs = xsA.
```

A reference implementation of all the definitions shown in this section as well as proofs for the above propositions on thunks and lists can be found in our public artifact [Li et al. 2021].

5.4 Functional Correctness

To say that our approximation function implements the pure specification, we would like two notions of correctness: (1) a *partial correctness* notion that requires all the nondeterministic results of the approximation function to be approximations of the result of the pure function; and (2) a *pure correctness* notion that states the existence of a maximal approximation result that is exactly the pure function's result.

We define the partial correctness of a function using the pessimistic specification, and the pure correctness of a function using the optimistic specification. For example, the partial and pure specifications of `appendA` (Section 4.3) are shown in Fig. 15. Given approximations of two input lists `xs` and `ys`, `appendA` always, *i.e.,* pessimistically, yields an approximation of `append xs ys`. On the other hand, `appendA` optimistically yields exactly the list `append xs ys` when applied to the exact approximations of `xs` and `ys`. Both theorems can be proved by an induction over `xs`.

5.5 Cost Specifications

In this section, we show how we use both the pessimistic and the optimistic specifications for reasoning about computation costs.

```
Fixpoint sizeX {a} (n0 : nat) (xs : T (listA a)) : nat :=
  match xs with
  | Thunk NilA => n0
  | Thunk (ConsA _ xs1) => S (sizeX n0 xs1)
  | Undefined => 0
  end.

Definition is_defined {a} (t : T a) : Prop :=
  match t with
  | Thunk _ => True
  | Undefined => False
  end.
```

Fig. 16. Definition of sizeX[7] and is_defined.

Using appendA as our running example, we first start with a pessimistic specification. Since a pessimistic specification describes all the nondeterministic branches of a clairvoyant call-by-value evaluation, it might contain spurious branches which overapproximate call-by-need evaluation too much. Thus, it can only offer a loose upper bound for the computation cost. Nevertheless, it is useful in specifying the lower bound, while we can rely on an optimistic specification to tighten the bounds.

Taking the appendA function (Fig. 11) again as our example, we can give it a pessimistic specification as follow:

```
Theorem appendA_cost_interval {a} : forall (xsA ysA : T (listA a)),
    (appendA xsA ysA)
    {{ fun zsA cost => 1 <= cost <= sizeX 1 xsA }}.
```

The xsA and ysA passed to the appendA function are approximations of the pure values xs and ys.

The size of approximation lists, defined in Fig. 16, is a function intended purely for reasoning, hence we name it sizeX, with a different suffix from implementations such as appendA. It is also parameterized by the “size” of NilA, which is 0 or 1 depending on whether its presence matters for a given specification. Here, the extra unit of time accounts for the final call to appendA which matches on NilA.

A problem with this specification is that it only gives a range of the computation costs. During the actual evaluation of the function p, the takeA function would never require more than the first n elements of appendA’s resulting list, but this specification of appendA fails to reflect that. We will only be able to compute that the combined cost has a lower bound of 3 and an upper bound of (sizeX 1 xsA) + n + 1, while in an actual call-by-need evaluation, the cost would never exceed 2n + 1. [8]

To address this problem, we give an optimistic specification to append. A first version states that appendA reaches the lower bound of the interval in at least one execution.

```
Theorem appendA_whnf_cost {a} : forall (xsA ysA : T (listA a)),
    (appendA xsA ysA)
    [[ fun zsA cost => cost <= 1 ]].
```

[7] Simplified for clarity. This is actually ill-formed because the type T (listA A) is not a recursive type (cf. Section 4.1).
[8] Note that the size of xsA can be bigger than n if it is required with a higher demand elsewhere.

The execution of `appendA` which satisfies that specification immediately discards the computation in the `let~`, producing only a result in WHNF.

That specification could be strengthened to an equality `cost = 1`. However, it is important to remember that results `(zsA, cost)` of a clairvoyant computation are formal approximations of the behavior of a lazy program. `zsA` is an approximation of the function's result, and `cost` is an upper bound on its actual cost. Hence, only upper bounds on `cost` are meaningful in optimistic specifications, while pessimistic specifications can assert both lower and upper bounds. For that reason, we leave specifications of `cost` as inequalities even though the simple specifications in this section are technically valid with equalities. Note also that pessimistic upper bounds are quite fragile; they can be broken simply by adding spurious thunks in a program.

Optimistic specifications about a single result such as `appendA_whnf_cost` are unhelpful in most proofs, of course. A more expressive way to phrase optimistic specifications is to set an arbitrary *demand*, raising the cost accordingly.

Examining executions of `appendA` more closely, we can distinguish two phases, with separate specifications. Before reaching the end of the first list `xsA`, `appendA` computes an approximation of length `n` in time `n`, for any `n` smaller than the size of `xsA`.

```
Theorem appendA_prefix_cost {a} : forall n (xsA ysA : T (listA a)),
  1 <= n <= sizeX 0 xsA ->
  (appendA xsA ysA) [[ fun zsA cost => n = sizeX 0 (Thunk zsA) /\ cost <= n ]].
```

The natural number `n` represents an explicit demand on the output of `appendA xsA ysA`: we demand an approximation with `n` constructors `ConsA`, costing at most `n` units of time.

Another specification describes the execution of `appendA` that reaches the end of the first list, yielding the most defined result—limited only by the possible partiality of `ysA`. As a necessary condition, `xsA` must be an exact approximation—modulo the definedness of its elements. Once we reach the end of the list `xsA`, the thunk `ysA` will be forced, so we require it to be defined, using the `is_defined` predicate in Fig. 16. This guarantees that `zsA` will be defined past the end of `xsA`, as specified by assigning a nonzero size to `NilA` in applications of `sizeX`.[9]

```
Theorem appendA_full_cost {a} : forall (xs : list a) (xsA := exact xs) ysA,
  is_defined ysA ->
  (appendA xsA ysA) [[ fun zsA cost =>
    length xs + sizeX 1 ysA = sizeX 1 (Thunk zsA) /\ cost <= length xs + 1 ]].
```

Natural numbers are not the most precise model of demand on lists: one may also specify whether and to what extent the elements of the list in the first field of `ConsA` constructors should be evaluated. In fact, approximation types such as `listA` are the most general way to model demand. However, when list elements are not explicitly used, a natural number is enough to formalize the main aspects of complexity analysis for list operations.

6 CASE STUDY: TAIL RECURSION

We have already demonstrated our methodology on the program described in Section 2. Here, we show how to apply this approach in another context: reasoning about functions written with tail recursion. [10] Tail recursion is a common optimization technique in the context of an eager semantics. However, it can be a cause of performance degradation if not used properly under lazy evaluation.

[9] The `(xsA := exact xs)` binding in the following snippet is desugared into a local definition using `let`.
[10] All the theorems discussed in this section have been formally proved in Coq and the proofs are available in our artifact [Li et al. 2021].

Tail Recursive take. Consider a tail recursive version of the take function from Section 2, called take'. The key difference is the addition of an accumulator to its parameters:

```
Fixpoint take'_ {a} (n : nat) (xs : list a) (acc : list a) : list a :=
  match n with
  | O => rev acc
  | S n' => match xs with
            | nil => rev acc
            | cons x xs' => take'_ n' xs' (x :: acc)
            end
  end.

Definition take' {a} (n : nat) (xs : list a) : list a := take'_ n xs nil.
```

Even though the list must be reversed in the base case, take' is better in an eager programming language because the compiler can eliminate stack allocation [Friedman and Wise 1974].

However, the original version is better for lazy evaluation, even if we ignore the cost of rev. To get an intuition of why, consider the case where we only demand the WHNF of the resulting list. This variant take' must go over n elements of the list before it returns the accumulator. In comparison, the original take can immediately reveal the first element of the resulting list in any of its pattern matching branches.

A Formal Analysis. With the help of formal reasoning, we can better understand these functions' difference from their specifications. In the specifications below, we axiomatize the cost of rev used in take'_ to have a cost of 0 [11]. We introduce this axiom so we can compare only the costs incurred by the recursive calls on take'_ and take. With this set up, let's look at the pessimistic specification of take'A_, the version of take'_ written in the clairvoyance monad: [12]

```
forall (n : nat) (xs : list a) (xsA : listA a) (acc : list a) (accA : T (listA a)),
  xsA `is_approx` xs ->  accA `is_approx` acc ->
  (take'A_ n xsA accA) {{ fun zsA cost => cost = min n (length xs) + 1 }}.
```

The pessimistic specification describes a rather precise cost on all the nondeterministic branches of take'A_. Furthermore, the cost is purely decided by the pure values n and xs—the fact that the cost does not depend on the actual approximations zsA output by take'A_ is a sign that the function may not be making effective use of laziness.

However, to show that take is better than take', we need to show that take can cost less than take'. What specification should we use to show that? One possibility is the pessimistic specification. If we take this approach, we can show that the cost of takeA (take in the clairvoyance monad) is upper bounded by n and the size of the approximation xsA:

```
forall (n : nat) (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (takeA n xsA) {{ fun zsA cost => cost <= min n (sizeX 0 xsA) + 1 }}.
```

Since approximations are always smaller than their pure values, the cost shown here is smaller than that of take'A_—*if* such costs indeed exists. Unfortunately, the pessimistic specification, which quantifies universally over all executions, does not guarantee the existence of an execution. In fact,

[11] The axiom would not make Coq's logic unsound because we can define such a function in the clairvoyance monad by not inserting ticks.

[12] For simplicity, we omit the functional correctness parts of the specifications in this section.

take'A admits no execution at all if its arguments are not sufficiently defined, so it satisfies the above specification vacuously.

To show the existence of certain costs, we need an optimistic specification:

```
forall (n m : nat) (xs : list a) (xsA : T (listA a)),
  1 <= m ->  m <= min (n + 1) (sizeX 1 xsA) ->
  xsA `is_approx` xs ->
  (takeA n xsA) [[ fun zsA cost => cost = m ]].
```

The pessimistic specification of take'A_ and the optimistic specification of takeA help unveil the key difference between these two: take'A_ does not make effective use of laziness, while the cost of takeA scales with its demand.

List Reversal. On the other hand, there are functions like rev that do benefit from tail recursion. Consider a naive version without tail recursion:

```
Definition rev' {a} (xs : list a) : list a :=
  match xs with
  | nil => nil
  | x :: xs' => append (rev' xs') (cons x nil)
  end.
```

And the version with tail recursion:

```
Fixpoint rev_ {a} (ys : list a) (xs : list a) : list a :=
  match xs with
  | nil => ys
  | x :: xs => rev_ (x :: ys) xs
  end.

Definition rev {a} (xs : list a) : list a := rev_ nil xs.
```

One reason that the non-tail-recursive version is worse is that append has a non-constant cost, which leads rev to have a cost which grows quadratically in the length of the input list. However, even if we imagine a constant time version of append (*e.g.,* difference lists [Hughes 1986], catenable lists [Okasaki 1999]), this version would not be better than the tail-recursive one. [13] Intuitively, this is because both versions need to traverse the entire list xs to return the first element of the resulting list, which is the last element of the input list.

Again we can inspect these two versions of rev formally to better understand their difference. Like above, we axiomatize append to have a cost of 0 so that our analysis only considers the cost incurred by the recursive calls of rev_ and rev'. This simplification makes rev' cost less but will only strengthen our claim that rev' would not beat rev.

First, we can show a rather specific cost with the pessimistic specification of revA:

```
forall (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (revA xsA) {{ fun zsA cost => cost = length xs + 1 }}.
```

In that specification, the cost is associated with the pure input value, and is independent of the output value, which suggests that revA does not make use of laziness. Indeed, this is true: we must iterate over the entire list xs to get the first element of the resulting list.

[13] If we consider the stack usage and compiler optimizations, the tail-recursive version would be generally more efficient and does not risk causing stack overflow.

We can also prove that `rev'A` satisfies the following pessimistic specification:

```
forall (xs : list a) (xsA : T (listA a)),
  xsA `is_approx` xs ->
  (rev'A xsA) {{ fun zsA cost => cost = length xs + 1 }}.
```

This specification shows that the cost of `rev'A` also does not depend the approximations of `xs`, confirming our intuition that `rev'A` must also iterate over the entire list.

Left and Right Folds. One famous example concerning laziness is the difference between `foldl` and `foldr`. While it seems that the major difference is just their directions of folding, they actually have rather different costs. For simplicity, let's only consider these operations on `lists`. The definitions of `foldl` and `foldr` are shown below:

```
Fixpoint foldl {a b} (f : b -> a -> b) (v : b) (xs : list a) : b :=
  match xs with
  | nil => v
  | cons x xs => foldl f (f v x) xs
  end.

Fixpoint foldr {a b} (v : b) (f : a -> b -> b)  (xs : list a) : b :=
  match xs with
  | nil => v
  | cons x xs => f x (foldr v f xs)
  end.
```

A formal analysis of these functions must also consider the cost of the function `f` passed into them. For simplicity, let's assume that `f` has a cost of only 1. We can then prove that the translated versions of the above two functions satisfy the following pessimistic specification:

```
(** The pessimistic specification of [foldlA]. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  (forall x y, (f x y) {{ fun bA cost => exists b, bA `is_approx` b /\ cost = 1 }}) ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
  (foldlA f vA xsA)
    {{ fun zsA cost => cost >= length xs + 1 /\ cost <= 2 * length xs + 1 }}.

(** The pessimistic specification of [foldrA]. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  (forall x y, (f x y) {{ fun bA cost => cost = 1 }}) ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
  (foldrA f vA xsA)
    {{ fun zsA cost => cost >= 1 /\ cost <= 2 * sizeX 0 xsA + 1 }}.
```

The pessimistic specifications suggest that `foldrA` makes better use of laziness because its cost is bounded by the length of approximation `xsA`. However, as we have discussed earlier, we need to show that there are indeed costs lower than the lower bound of `foldlA` that exists in some nondeterministic branches of `foldrA`. For that, we once again need to show the optimistic specification of `foldrA`:

```
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA) n,
  1 <= n -> n < sizeX 0 xsA ->
  xsA `is_approx` xs ->  vA `is_approx` v ->
```

```
  (forall x y, (f x y) [[ fun bA cost => cost = 1 ]]) ->
  (foldrA f vA xsA) [[ fun zsA cost => cost = 2 * n ]].

(** And a special cost exists when [xs] is fully evaluated. *)
forall f (xs : list a) (xsA : T (listA a)) (v : b) (vA : T bA),
  xsA = exact xs ->  vA `is_approx` v -> is_defined vA ->
  (forall x y, (f x y) [[ fun bA cost => cost = 1 ]]) ->
  (foldrA f vA xsA) [[ fun zsA cost => cost = 2 * length xs + 1 ]].
```

This concludes that, under lazy evaluation, `foldl` and `foldr` have the same worst-case cost, but `foldr` has a lower cost if the demand is lower.

7 DISCUSSION

Nondeterminism Monads. The clairvoyance monad is not the only way to model nondeterminism; the list monad is another possibility which allows some of these ideas to be implemented without dependent types—necessary for `exists` and `eq`. Furthermore, the list monad makes clairvoyant computations actually executable. We only require a notion of choice and emptiness for programming; for reasoning, we need some theory of membership `member : M a -> a -> Prop`, which defines a monad morphism into the powerset monad. This suggests that choosing the powerset monad as we do here yields the simplest definitions.

Recursive Let*.* A recursive Let for data types is yet another challenge. It allows the construction of certain infinite structures by "tying the knot", such as:

```
ones = 1 :: ones      and      fib = 0 :: 1 :: zip_with plus fib (tail fib)
```

However, these represent only a small fraction of programs manipulating infinite structures, and there are many others, which do not rely on recursive Let, that our reasoning framework can already handle. For instance, the coinductive versions of `append` and `take` are also actually modeled by our functions translated from the inductive versions (Fig. 11), by viewing our inductive `listA` also as an approximation type for infinite lists.

In Haskell, Let is always recursive, but it is so rarely useful for constructing data (as opposed to functions) that the shadowing it introduces is often considered undesirable, and there is at least one proposal to disable it. [14]

8 RELATED WORK

Clairvoyant Evaluation. Clairvoyant evaluation was first characterized by Hackett and Hutton [2019]. The main inspiration for our paper, this work presented an operational semantics for laziness as an alternative to the natural semantics of Launchbury [1993], as well as a denotational cost semantics, following precursory ideas by Maraist et al. [1995]. We have compared our translation with the semantics of Hackett and Hutton as well as established an equivalence relation between them in Section 4.2.

Monadic Translation. Moggi [1991] uses monads to describe computational effects and defines various translations corresponding to different calling conventions. Wadler [1992] follows and describes the translations for the call-by-value and call-by-name semantics, but leaves the translation for call-by-need as an open problem. Uustalu [2002] further adds positive inductive and coinductive types to these translations.

[14] https://github.com/ghc-proposals/ghc-proposals/blob/68164fb2d5a71b62223a8287c0c0390147c0dc2f/proposals/0000-letrec.md

Petricek [2012] proposes a monadic translation that can be used under all three different calling conventions, generalizing Wadler [1992], by defining a function called `malias` which would be given different meanings under different semantics. In particular, `malias : M a -> M (M a)` is closely related to our function `thunk : M a -> M (T a)`: it occupies the same position in the translation of LET. The main difference is that, whereas `malias` wraps thunks in computations; we expose thunks `T` as a separate type. In fact, we could define `malias t = bind (thunk t) force` to hide thunks as well. However, exposing `T` is crucial for use in nested recursive type declarations; using `M` instead would violate Coq's strict positivity condition. Furthermore, in Petricek [2012], `M` is defined using Haskell's `ST` monad, relying on a polymorphic interface of mutable references to represent the heap. We believe that it would be challenging to emulate such an interface in Coq.

Computation Cost and Laziness. There is much work on reasoning formally about computation costs. For example, Crary and Weirich [2000]; Danielsson [2008]; Hoffmann et al. [2012]; Lago [2011]; Rajani et al. [2021]; Wang et al. [2017] study intrinsic approaches to formal cost analysis. Our work is in the extrinsic context of an interactive proof assistant.

On the extrinsic side, Charguéraud and Pottier [2019]; Guéneau et al. [2018] use separation logic for reasoning about computation costs under call-by-value evaluation using amortized analysis. Compared to these works, our goal of reasoning about lazy pure functional programs does not require separation logic. Cost specifications could be made more modular by hiding implementation-specific constant factors and formulating costs in asymptotic terms. Works on formalizing asymptotic complexity include Cutler et al. [2020]; Eberl [2021]; Guéneau [2019].

Danielsson [2008]; Handley et al. [2020] reason about lazy functional programs in a monadic syntax annotated with ticks. An issue in both works is that they require an explicit notion of laziness to model sharing: for example, in practice, a list that is evaluated once will not be evaluated again under lazy evaluation. To avoid a "double counting" of the cost in a thunk, a `pay : nat -> M a -> M (M a)` combinator with an explicit representation of cost must be annotated in the code [15]. This prevents both works to be fully extrinsic in reasoning about laziness. With the clairvoyance monad, thunks are either paid for or discarded immediately, so it is impossible to count the cost of a thunk twice. This enables us to translate pure lazy functions mechanically to monadic programs, and our proofs are completely extrinsic.

On the automated reasoning side, Madhavan et al. [2017] verify a purely functional subset of Scala by translating higher-order functions to first-order programs via defunctionalization. They also model memoization by encoding the cache as an expression that changes during the execution of the program.

For testing lazy functions in Haskell, Sloth is a tool that automatically generates test cases to check if a function is "unnecessarily strict" [Christiansen 2011]. This tool relies on a "less-strict" ordering of functions. One function is less-strict than another when, given the same input, its result is less defined [Christiansen and Seidel 2011].

Foner et al. [2018] develop a library that generates random demands on the output of a function and instruments inputs to record induced demand. Demands take the form of approximations whose structure is also derived from pure data types.

Haskell. Although we only discuss Coq here, Haskell is also a potential target of our approach. The `hs-to-coq` tool automatically translates Haskell programs to Gallina—Coq's specification language—using shallow embeddings [Spector-Zabusky et al. 2018]. It has been used for verifying a significant portion of Haskell's `containers` library [Breitner et al. 2021] and one part of

[15] The `pay` combinator is also an annotated version of `malias : M a -> M (M a)` [Petricek 2012].

GHC [Spector-Zabusky et al. 2019]. However, hs-to-coq's pure translation cannot be used for cost analysis so existing work using this tool has been restricted to functional correctness.

Abel et al. [2005] and Dylus et al. [2019] respectively translate Haskell to monadic embeddings in Agda and Coq, based on the call-by-name translation by Moggi [1991]. This is enough to model Haskell's partiality, but not its lazy cost semantics.

Liquid Haskell augments Haskell with refinement types [Vazou 2016] to enable formal verification, and it has been applied to cost analysis [Handley et al. 2020]. The major difference is that our work does not require an explicit notion of laziness, as discussed earlier in this section. Furthermore, Handley et al. [2020] verify Haskell programs written explicitly in the tick monad; to analyze arbitrary Haskell programs, some monadic translation is necessary.

Nondeterminism and Dual Logics. Our optimistic and pessimistic specifications are examples of predicate transformer semantics. They date back to Dijkstra [1975], forming the basis of much work on the verification of effectful programs in type theory [Nanevski et al. 2008; Swamy et al. 2013; Swierstra 2009; Swierstra and Baanen 2019]. Our predicate transformer semantics are two conventional effect observations [Maillard et al. 2019] from the clairvoyance monad—a variant of the powerset monad—to the specification monads respectively for angelic and demonic nondeterminism.

The duality between pessimistic and optimistic specifications is also the duality of Hoare logic [Hoare 1969] and reverse Hoare logic [de Vries and Koutavas 2011; O'Hearn 2020]. Those logics use sets of states to approximate program behavior. In Hoare logic, the postcondition over-approximates the set of reachable states; in reverse Hoare logic, the postcondition under-approximates the set of reachable states. Here, we show that abstractions for angelic and demonic nondeterminism give rise, rather simply, to logics of over- and under-approximations of time consumption. The notion of approximation underlying our logics is formally defined as follows: a set of cost-value pairs A underapproximates a set of pairs B if, for every $(v, c) \in A$, there exists $(w, d) \in B$ which "costs less and is more defined", *i.e.,* such that $d \leq c$ and $v \leq w$. Thus, sets of states are ordered by inclusion in Hoare logic, whereas sets of cost-value pairs follow a more elaborate order structure in our dual logic, based on the view that those pairs themselves are approximations of the actual behavior of lazy programs.

9 CONCLUSION AND FUTURE WORK

In this paper, we present a novel and simple framework for formally reasoning about costs of lazy functional programs. The framework is based on a new model of lazy evaluation: clairvoyant call-by-value [Hackett and Hutton 2019], which makes use of nondeterminism to avoid modeling mutable higher-order state in classic models of laziness [Launchbury 1993].

Our framework includes a simple clairvoyance monad and a translation from a typed calculus to programs in this monad that captures the semantics of clairvoyant call-by-value. Compared with the denotational semantics of Hackett and Hutton, our translation deals with typed programs, does not rely on domain theory, and accounts for the cost of every nondeterministic execution. We also develop dual logics *over-* and *under-approximations* similar to those of de Vries and Koutavas [2011]; Hoare [1969]; O'Hearn [2020] that enable local and modular formal reasoning of computation costs. We show the effectiveness of our approach via several small case studies.

In future work, we would like to apply this methodology to existing programs written in Haskell. In particular, we would like to augment tools like hs-to-coq [Spector-Zabusky et al. 2018] so that they can automatically translate Haskell programs to the clairvoyance monad and explore techniques in Coq that can be used to automate reasoning in those logics.

ACKNOWLEDGMENTS

We thank Koen Claessen, who recommended to one author the work of Hackett and Hutton [2019] during ICFP'20, and Deepak Garg, who recommended to another author the work of Danielsson [2008] during POPL'21. We thank the anonymous reviewers of ICFP'21, whose feedback helped improve the paper. We also thank our shepherd Leonidas Lampropoulos.

This material is based upon work supported by the National Science Foundation under Grant No. 1521539, and Grant No. 2006535. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell programs using constructive type theory. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2005, Tallinn, Estonia, September 30, 2005*, Daan Leijen (Ed.). ACM, 62–73. https://doi.org/10.1145/1088348.1088355

Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua Cohen, and Stephanie Weirich. 2021. Ready, Set, Verify! Applying hs-to-coq to real-world Haskell code. *Journal of Functional Programming* 31 (2021), e5. https://doi.org/10.1017/S0956796820000283

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *J. Autom. Reason.* 62, 3 (2019), 331–365. https://doi.org/10.1007/s10817-017-9431-7

Jan Christiansen. 2011. Sloth - A Tool for Checking Minimal-Strictness. In *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings (Lecture Notes in Computer Science)*, Ricardo Rocha and John Launchbury (Eds.), Vol. 6539. Springer, 160–174. https://doi.org/10.1007/978-3-642-18378-2_14

Jan Christiansen and Daniel Seidel. 2011. Minimally strict polymorphic functions. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, Peter Schneider-Kamp and Michael Hanus (Eds.). ACM, 53–64. https://doi.org/10.1145/2003476.2003487

Karl Crary and Stephanie Weirich. 2000. Resource Bound Certification. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, Mark N. Wegman and Thomas W. Reps (Eds.). ACM, 184–198. https://doi.org/10.1145/325694.325716

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

Edsko de Vries and Vasileios Koutavas. 2011. Reverse Hoare Logic. In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings (Lecture Notes in Computer Science)*, Gilles Barthe, Alberto Pardo, and Gerardo Schneider (Eds.), Vol. 7041. Springer, 155–171. https://doi.org/10.1007/978-3-642-24690-6_12

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

Sandra Dylus, Jan Christiansen, and Finn Teegen. 2019. One Monad to Prove Them All. *Art Sci. Eng. Program.* 3, 3 (2019), 8. https://doi.org/10.22152/programming-journal.org/2019/3/8

Manuel Eberl. 2021. *Asymptotic Reasoning in a Proof Assistant.* Dissertation. Technical University of Munich, Munich, Germany. http://nbn-resolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20210121-1554821-1-2

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

Daniel P. Friedman and David S. Wise. 1974. *Unwinding Structured Recursions into Iterations.* Technical Report TR19. Indiana University. https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR19

Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs. (Vérification mécanisée de la correction et complexité asymptotique de programmes).* Ph.D. Dissertation. Inria, Paris, France. https://tel.archives-ouvertes.fr/tel-02437532

Armaël Guéneau, Arthur Charguéraud, and François Pottier. 2018. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018,*

Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science), Amal Ahmed (Ed.), Vol. 10801. Springer, 533–560. https://doi.org/10.1007/978-3-319-89884-1_19

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

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

Peter Henderson and James H. Morris, Jr. 1976. A Lazy Evaluator. In *Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, USA, January 1976*, Susan L. Graham, Robert M. Graham, Michael A. Harrison, William I. Grosky, and Jeffrey D. Ullman (Eds.). ACM Press, 95–103. https://doi.org/10.1145/800168.811543

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. https://doi.org/10.1145/363235.363259

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

John Hughes. 1986. A Novel Representation of Lists and its Application to the Function "reverse". *Inf. Process. Lett.* 22, 3 (1986), 141–144. https://doi.org/10.1016/0020-0190(86)90059-1

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

Mark B. Josephs. 1989. The Semantics of Lazy Functional Languages. *Theor. Comput. Sci.* 68, 1 (1989), 105–111. https://doi.org/10.1016/0304-3975(89)90122-9

Ugo Dal Lago. 2011. A Short Introduction to Implicit Computational Complexity. In *Lectures on Logic and Computation - ESSLLI 2010 Copenhagen, Denmark, August 2010, ESSLLI 2011, Ljubljana, Slovenia, August 2011, Selected Lecture Notes (Lecture Notes in Computer Science)*, Nick Bezhanishvili and Valentin Goranko (Eds.), Vol. 7388. Springer, 89–109. https://doi.org/10.1007/978-3-642-31485-8_3

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

Yao Li, Li-yao Xia, and Stephanie Weirich. 2021. Reasoning about the garden of forking paths (artifact). https://doi.org/10.5281/zenodo.4771438 GitHub repository at: https://github.com/lastland/ClairvoyanceMonad.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. 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, 333–343. https://doi.org/10.1145/199448.199528

Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. 2017. Contract-based resource verification for higher-order functions with memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 330–343. http://dl.acm.org/citation.cfm?id=3009874

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra monads for all. *Proc. ACM Program. Lang.* 3, ICFP (2019), 104:1–104:29. https://doi.org/10.1145/3341708

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, LA, USA, March 29 - April 1, 1995 (Electronic Notes in Theoretical Computer Science)*, Stephen D. Brookes, Michael G. Main, Austin Melton, and Michael W. Mislove (Eds.), Vol. 1. Elsevier, 370–392. https://doi.org/10.1016/S1571-0661(04)00022-2

Coq development team. 2021. *The Coq proof assistant.* http://coq.inria.fr Version 8.13.2.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

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

Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911. https://doi.org/10.1017/S0956796808006953

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.

Tomas Petricek. 2012. Evaluation strategies for monadic computations. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012 (EPTCS)*, James Chapman and Paul Blain Levy (Eds.), Vol. 76. 68–89. https://doi.org/10.4204/EPTCS.76.7

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

Dana S. Scott. 1976. Data Types as Lattices. *SIAM J. Comput.* 5, 3 (1976), 522–587. https://doi.org/10.1137/0205037

Antal Spector-Zabusky, Joachim Breitner, Yao Li, and Stephanie Weirich. 2019. Embracing a mechanized formalization gap. *CoRR* abs/1910.11724 (2019). arXiv:1910.11724 http://arxiv.org/abs/1910.11724

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

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying higher-order programs with the Dijkstra monad. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 387–398. https://doi.org/10.1145/2491956.2491978

Wouter Swierstra. 2009. A Hoare Logic for the State Monad. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.), Vol. 5674. Springer, 440–451. https://doi.org/10.1007/978-3-642-03359-9_30

Wouter Swierstra and Tim Baanen. 2019. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.* 3, ICFP (2019), 103:1–103:26. https://doi.org/10.1145/3341707

D. A. Turner. 2004. Total Functional Programming. *J. Univers. Comput. Sci.* 10, 7 (2004), 751–768. https://doi.org/10.3217/jucs-010-07-0751

Tarmo Uustalu. 2002. Monad Translating Inductive and Coinductive Types. In *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers (Lecture Notes in Computer Science)*, Herman Geuvers and Freek Wiedijk (Eds.), Vol. 2646. Springer, 299–315. https://doi.org/10.1007/3-540-39185-1_17

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. 1992. Comprehending Monads. *Math. Struct. Comput. Sci.* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

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