Garlene: Guarded Recursion in Lean

Sergei Stepanenko
Aarhus University
Aarhus, Denmark
sergei.stepanenko@cs.au.dk

Patrick Bahr
IT University of Copenhagen
Copenhagen, Denmark
paba@itu.dk

Rasmus Ejlers Møgelberg
IT University of Copenhagen
Copenhagen, Denmark
mogel@itu.dk

Abstract

Extending the recursion principles of a formal system is an enticing but dangerous endeavor with a well-documented history of leading to consistency bugs. Nakano's guarded recursion is an elegant, type-based approach to soundly extend type theory with a powerful recursion principle. This makes guarded recursion useful for many applications, from programming with infinite structures such as streams to reasoning about advanced programming language features using synthetic guarded domain theory. Sadly, support for guarded recursion in mainstream interactive theorem provers remains limited: Existing implementations are experimental, restricted to fragments of its standard model, or difficult to use for larger developments.

In this paper, we present an implementation of guarded recursion as an embedded language in Lean consisting of a simply typed lambda calculus for definitions and a higher-order logic for reasoning. Using Lean's excellent support for metaprogramming, our language allows users to write guarded recursive definitions in an intuitive syntax and to prove properties about them using a dedicated proof mode. We give our language a presheaf model, which we use to prove the soundness of our language and to allow users to export guarded recursive definitions and their theorems into standard Lean developments. To demonstrate the usefulness of our language, we present several case studies for programming and reasoning with guarded recursion.

CCS Concepts: • **Theory of computation** → **Higher order logic**; **Type theory**; *Program verification.*

Keywords: guarded recursion, topos of trees, Lean, deep embedding, proof mode

ACM Reference Format:
Sergei Stepanenko, Patrick Bahr, and Rasmus Ejlers Møgelberg. 2027. Garlene: Guarded Recursion in Lean. In *Proceedings of the 16th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '27), January 11–12, 2027, Mexico City, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3871849.3871895



This work is licensed under a Creative Commons Attribution 4.0 International License.

CPP '27, Mexico City, Mexico
© 2027 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3146-4/2027/01
https://doi.org/10.1145/3871849.3871895

1 Introduction

Guarded recursion [28] is a powerful framework for programming and reasoning with recursion in type theory. The central gadgets of guarded recursion are 1) the 'later' modality mapping a type A to the type $\blacktriangleright A$ of values of type A delayed by one time step, 2) a fixed point operator $\mathsf{fix} : (\blacktriangleright A \to A) \to A$, and 3) guarded recursive types, *i.e.*, solutions to equations such as

$$D \cong 1 + 1 + \blacktriangleright D + \blacktriangleright (D \to D) \tag{1}$$

where the recursion variable D occurs only under the modality $\blacktriangleright$, but is allowed to occur also in negative positions. One way to think of guarded recursion is as an abstract approach to step-indexing [1], and this view is made precise in the standard model of guarded recursion, the topos of trees [5], which models types as presheaves over ω, the ordered natural numbers. One important use case of guarded recursion is for denotational semantics of programming languages with recursion [26, 30], also in combination with other features such as probabilistic choice [32] or non-determinism [27]. This program is often referred to as *synthetic guarded domain theory*.

Unfortunately, due to limited support for guarded recursion in proof assistants (see Section 2 below for a detailed discussion), none of the above-mentioned applications have been formalized in proof assistants. One of the main challenges for doing so is that the language they are expressed in – Guarded Type Theory [3] – uses Fitch style [9] for programming and reasoning about the $\blacktriangleright$ modality, and such languages are not easily implemented as shallow embeddings in type theory, because typechecking requires access to the context of expressions.

This paper presents Garlene, a framework for guarded recursion in Lean 4 [14], based on a deep embedding. Deep embeddings have a reputation for being expensive to work with, but two design decisions make ours feasible while preserving expressivity. The first is the decision to not implement the full dependent type theory used in previous applications. Rather, we implement a simply typed guarded λ-calculus, with a higher-order logic (also with guarded recursion) over it. This has the benefit that typechecking never needs to normalize terms, and we perform normalization only inside proofs. The second is that typing and logical derivability are valued in Lean's definitionally proof-irrelevant propositional universe Prop, so that derivations are never unfolded or compared. Taken together, these two aspects make quotation cheap: A Garlene program is typechecked once, and the

produced proof is not unfolded when the program is referred to in other programs.

Garlene is built in two layers. The first layer is the λ-calculus and logic, together with the tooling needed to work with a deep embedding: a typechecker for the λ-calculus, and a proof mode with a set of tactics for the logic, which allows one to prove statements internal to our calculus. In both the calculus and the logic, $\blacktriangleright$ is eliminated in Fitch style: by adv in terms and by the corresponding LATER-ELIM rule in proofs. The second layer is a model of both the calculus and the logic in the topos of trees, formalized in Lean using Mathlib [34]. The model serves two purposes: to prove the calculus and the logic sound, and to serve as a bridge between Garlene and ordinary Lean. Theorems proved in Garlene can be exported to Lean by interpretation into the model, and theorems proved about the model can be imported into Garlene. To facilitate this bridge between Garlene and Lean, our calculus and logic are both expandable: Any presheaf in the topos of trees can be used as a type in Garlene, and likewise any term and proof.

We illustrate the use of Garlene by two case studies. The first is the guarded delay monad Delay defined as the guarded recursive type satisfying $\mathrm{Delay}\, A \cong A + \blacktriangleright(\mathrm{Delay}\, A)$. This is a guarded variant of Capretta's delay monad [6] and is used in synthetic guarded domain theory as a monad for recursion. We show that this is a monad, and that Delay A is the free delay algebra on A, where a delay algebra is a type B with an operation $\blacktriangleright B \to B$. We show that as a consequence, Delay defines a monad on the topos of trees in the sense of Mathlib's category theory library. The proof uses both directions of the bridge between Garlene and Lean. The second is a programming-language case study, a sound and adequate denotational semantics for a typed λ-calculus with fixed points into a guarded universal domain satisfying equation (1) above; adequacy is an ordinary Lean theorem about the operational semantics of the calculus, obtained from equations between denotations proved in Garlene. The resulting proof scripts match ordinary Lean tactic proofs in size and shape; Figure 2 shows a representative one.

In summary, our contributions are:

1. an implementation in Lean of a Fitch-style λ-calculus with guarded recursion and a higher-order logic over it. The calculus is a restriction of Guarded Type Theory [3] to simple types and a single global clock. The logic is a new adaptation of guarded type theory.
2. a Kripke-style proof mode built from Lean tactics, in which guarded proofs are ordinary Lean tactic proofs.
3. two case studies: the delay monad, and a sound semantics for a typed λ-calculus with fixed points in a guarded universal domain, proved adequate for the well-typed fragment of the language.
4. implementation techniques and design decisions for working with deeply embedded calculi in Lean.

The implementation is written in Lean 4; all code shown in the paper is taken verbatim from it.

Overview. Section 2 discusses related work. Section 3 gives an informal overview of Garlene using the example of the guarded delay monad. The following sections then treat the calculus (Section 4), the logic (Section 5), the proof mode (Section 5.3) and the case studies (Section 6). Finally, Section 7 summarizes this work and provides some notes on future directions for this project.

Data Availability Statement. The Lean mechanization accompanying this work is available on Zenodo [33] and on GitHub at https://github.com/Kaptch/Garlene.

2 Related Work

Implementations of Guarded Type Theory. Guarded Cubical Agda [35] is an extension of Cubical Agda [36] with guarded recursion. It implements Clocked Cubical Type Theory [23], which combines guarded recursion with cubical type theory [10]. One of the benefits of this combination is that cubical type theory allows for extensionality principles to compute. In Garlene, these extensionality principles are stated using the proof-irrelevant universe of propositions, and so do not need to compute. The modality $\blacktriangleright$, as well as the notion of ticks used for Fitch-style programming with $\blacktriangleright$, are native to Guarded Cubical Agda, and the Fitch-style typing rules are built into the typechecker. This means that there is no distinction between the guarded type theory and native Cubical Agda like the distinction between Garlene and Lean in this work. Instead, Guarded Cubical Agda implements a version of guarded recursion where $\blacktriangleright$ is indexed by clocks. In this setting, working in a context with no free clock variables can be thought of as working in ordinary Cubical Agda, and working in a context with one free clock variable corresponds to working in a cubical version of the topos of trees. One can pass between the two by weakening in one direction, and by quantification over clocks in the other. Guarded Cubical Agda is available in Agda as an experimental extension enabled by the `--guarded` option, and has been used for a few verification projects [16]. However, the guarded fixed point operator and clock quantification are not primitive and have to be postulated, so results proved in the guarded setting, and their transfer to ordinary Cubical Agda, rest on postulated axioms. In Garlene, by contrast, export to Lean is justified by a model formalized in Lean.

Istari [11, 24] is a tactic-based proof assistant for an extensional Martin-Löf type theory that natively supports guarded recursion: A later modality (called 'future'), guarded recursive types, and a guarded fixed point combinator are built in and justified by a step-indexed semantics. In contrast to Guarded Cubical Agda, the fixed point combinator is primitive rather than postulated, and extensionality principles come from the extensional type theory rather than from

cubical type theory. However, Istari is a separate proof assistant with its own tactic language and libraries, whereas Garlene makes guarded definitions and theorems about them available in Lean.

Sikkel and BiSikkel [7, 8] implement multimode type theories [17], including guarded type theories, as deep embeddings in Agda together with a semantic interpretation, and are in this respect the closest to our work. Our approach differs in two ways. First, BiSikkel allows only for direct manipulation of deeply embedded syntax, whereas Garlene includes an elaboration from user syntax, and terms can moreover be constructed interactively by tactics. This gives a development experience closer to that of ordinary Lean proofs. Second, by using proof-irrelevant propositions we can carry typing and provability derivations in terms without ever inspecting them. This represents a major improvement in efficiency and allows for larger applications.

Logics with Later Modality. Iris [13, 20] provides the later modality, Löb induction (the logical correspondent of fix), and guarded recursive types and predicates. The Iris framework and MoSeL [21, 22] have become one of the biggest logics embedded in a proof assistant. This is the closest existing practice to ours, and our proof mode follows the same general approach: Tactics manipulate a reflected sequent, and the kernel checks the result. One difference lies in how the language is constructed: Iris uses a shallow embedding of terms manipulating the denotational model. While this allows a more direct implementation, it means that the Fitch-style modal languages that we use for programming and reasoning are unavailable. In addition to that, the denotational models are different: Iris uses a category of (complete) bounded bisected ultrametric spaces, which embeds fully faithfully into the topos of trees. Although we do not use objects that lie outside this category in this work, we may do so in future extensions. For example, comprehension types $\{x : A \mid \phi(x)\}$ generally do not live in the subcategory, but do live in the topos of trees. Future extensions may also change indexing to ordinals larger than ω for programming with finite powersets or reasoning about termination. Although a similar extension exists for Iris [31], it is unknown how to model general guarded recursive types in these, as is possible in extended versions of the topos of trees [25].

Frameworks for Synthetic Mathematics in Lean. The SynthLean framework [29] supports synthetic reasoning in Lean with interpretations into natural models of dependent type theory. It has the same overall architecture as our system: Surface syntax is elaborated into a deep embedding and then interpreted in a semantic model. SynthLean could in principle be instantiated with the topos of trees model, but offers no support for Fitch-style programming with modalities as it relies on a single context.

```
gtype Delay (A : TYPE) := ν X. [A] ⊕ ►X

gdef Delay.map (A B : TYPE) :
    (A → B) → [Delay A] → [Delay B] :=
  fix μ. λ f. λ d.
    [Delay.MK B]ₛ (case ([Delay.PROJ A]ₛ d)
      (λ a. inl (f a))
      (λ w. inr (delay (((adv 1 μ) f) (adv 1 w)))))
```

Figure 1. The guarded delay type and its functor action.

3 Overview of Garlene

In this section we provide a short demo of Garlene. We follow one example, namely the guarded delay monad, through the whole system: from its type declaration and programs (Figure 1), through proofs about these programs (Figure 2), to an extracted Lean theorem. To this end, we informally introduce Garlene as we go along, deferring the formal presentation of the language and its logic until Section 4 and Section 5, respectively. All code shown below is taken verbatim from the Lean implementation with only minor formatting changes.

3.1 A Guarded Type

We start with the type declaration in Figure 1. The command `gtype` declares a guarded recursive type. Here `Delay` is parametrized by `A : TYPE`, where `TYPE` is the type of Garlene types. The type `Delay A` is defined as a fixed point: `ν X` binds the recursion variable `X`, and `[A] ⊕ ►X` says that a value of type `Delay A` is either a value of type `A`, available now, or another `Delay A`, available only after taking a step. The latter is expressed by the later modality ►. A value of type `►T` is a value of type `T` that becomes available one step in the future. Unrolling the definition, an element of type `Delay A` is a value of type `A` after some finite number of steps, or a computation that never returns a value; this is the guarded version [30] of Capretta's delay monad [6]. Note that `X` occurs only under ►. This is what makes the fixed point well-defined, and it is the only requirement: There is no positivity requirement.

Two auxiliary notations appear already in this example and are used later. First, `A` is a Lean variable, and square brackets, as in `[A]` or `[Delay A]`, splice a Lean expression denoting a type into the syntax of Garlene. Second, `[c]ₛ` refers to a previously defined guarded program `c` of Garlene. We call this quotation (*cf*. Section 4.2).

Besides the type itself, the command generates two constants for folding and unfolding the fixed point,

```
Delay.MK A   : [A] ⊕ ►[Delay A] → [Delay A]
Delay.PROJ A : [Delay A] → [A] ⊕ ►[Delay A]
```

together with two equations stating that they are mutually inverse, named `Delay.MK_PROJ` and `Delay.PROJ_MK`. These two constants and two equations are the only interface to the type, and the fixed point is not unfolded definitionally.

```
gtheorem Delay.map_id (A : TYPE) :
    ∀ d : [Delay A].
      (([Delay.map A A]ₛ (λ x : A. x)) d = d) := by
    glöb IH
    gintro d
    gunfold Delay.map; gfix
    gcases ([Delay.PROJ A]ₛ d) with (⟨v, h⟩ | ⟨v, h⟩)
    · grewrite h; gsimpl; grewrite ← h
      gapply (Delay.MK_PROJ A)
    · grewrite h
      gsimpl
      gassert Htlf of ((delay _) = delay (adv 1 v))
      · gnext
        gadv IH as G
        gapply G
      grewrite Htlf
      grewrite (Delay.delay_eta A) v
      grewrite ← h
      gapply (Delay.MK_PROJ A)
```

Figure 2. The functor identity law for Delay.map. The two top-level bullet blocks are the inl and inr cases of the scrutinee. The highlighted line is the point where the proof state shown in Section 3.3 is taken.

3.2 Guarded Terms

Programs are written with the command `gdef`, which typechecks the body once, at definition time, and makes it available as a constant in Garlene.

Next, we define a function `Delay.map`, which is our first guarded recursive definition. `fix μ` binds the name of the recursive function μ, but at type `▶((A → B) → [Delay A] → [Delay B])`: The function being defined is available only one step in the future. The body proceeds by case analysis on `[Delay.PROJ A]ₛ d`. If the value is available now, we call `f` immediately. Otherwise we have `w : ▶[Delay A]`, and we have to make a recursive call on it, which can only be done a step later. This is where the two remaining primitives, `delay` and `adv`, come into play. They are the Fitch-style introduction and elimination forms for the later modality. `delay e` constructs a value of type `▶T` from `e : T`. Inside `e` we are one step in the future, and only there may `adv` be applied to values of type `▶T` from outside, such as μ and `w`, to get hold of the values they promise. The numeral in `adv 1` counts how many `delay`s we look out through; in the examples it is always `1`. Since `adv` can be applied to μ only under a `delay`, every recursive call is made after a step, and `Delay.map` is productive by construction; there is no separate termination or guardedness check. Section 4.1 makes this precise using a Kripke-style *stack of contexts*, where each `delay` opens a new frame and `adv n` reaches back `n` frames.

3.3 Proofs: Normalization and Tactics

Properties of programs are stated with `gtheorem` and proven in a higher-order logic over the calculus, which we present in Section 5. Figure 2 shows the functor identity law for `Delay.map`. The statement is again written in the syntax of Garlene. The proof is an ordinary Lean `by`-block. Every step of the proof script uses a tactic provided by Garlene for manipulating proof goals in Garlene's logic. We also get goals between steps, error messages, and hover information, which is delaborated to hide implementation details and look more readable than a raw goal representation. Garlene's tactics are prefixed with `g`, and most of them behave like their Lean counterparts: `gintro`, `gcases`, `grewrite`, `gapply`. Garlene's proof mode and its tactics are described in Section 5.3.

The proof is by Löb induction, initiated by the tactic `glöb IH`. If `P` is the current goal, `glöb IH` introduces an assumption `IH : lift (delay P)`, *i.e.*, `P` one step later, which we may use to prove `P`. We will write the composition of `lift` and `delay` as `▷`, which is a propositional version of `▶`. After unfolding the definition of `Delay.map` with `gunfold` and its fixed point with `gfix`, the proof branches on the value of `[Delay.PROJ A]ₛ d`. The interesting case is `[Delay.PROJ A]ₛ d = inr v` for some `v : ▶ [Delay A]`. In this case, `[Delay.map A A]ₛ (λ x : A. x) d` unfolds to `[Delay.MK A]ₛ (inr (delay (... (adv 1 v))))`, where the elided part is the unfolded recursive call. So it suffices to prove that `delay (... (adv 1 v)) = v`, an equality between terms of delayed type. By η-unfolding of the right-hand side, this is the goal `Htlf`. To prove such an equality between delayed data, it suffices to prove that the data delivered by each side in the next step are equal. The tactic `gnext` takes this step by moving the goal under the later modality, and `gadv IH as G` makes the induction hypothesis available as `G` without the later modality. Lean displays the goal at the highlighted line of Figure 2 as follows (we omit the unfolded body of `Delay.map`):

```
d : Delay A
v : ▶ [Delay A]
IH : lift (delay (∀ d. ([map A A]ₛ (λ x. x)) d = d))
h : [PROJ A]ₛ d = inr v
————————————————————
G : ∀ d. ([map A A]ₛ (λ x. x)) d = d
⊢ ((fix μ. ...) (λ x. x)) (adv 1 v) = adv 1 v
```

The horizontal line separates the two time frames: The hypotheses above it were introduced before the step, `G` after it. The goal then follows by application of `G`. The rest of the proof is a chain of rewritings: with `Htlf`, with the η-law for `delay`, and back with `h`, after which the generated equation `Delay.MK_PROJ` closes the goal, similar to the `inl` branch.

3.4 Denotation

So far, all definitions and proofs happened inside Garlene. What connects Garlene to the rest of Lean is the model: Types of Garlene are interpreted as objects of the topos of trees $\mathcal{I}$,

programs as morphisms and propositions as morphisms into the subobject classifier (Section 4.3). Soundness (Section 5.2) then says that every theorem proven in the proof mode holds in the model. For `Delay.map_id` this gives an ordinary Lean theorem about morphisms of $\mathcal{I}$:

```
theorem Delay.map_id_denotation (A : TYPE.{u}) :
    denoteHom (box([Delay.map A A]ₛ [idfun A]ₛ))
      = 1 ([[Delay A]]ₜ : ℐ.{u})
```

Here $[\![$`Delay A`$]\!]_t$ is the presheaf interpreting the type `Delay A`, `denoteHom` interprets a closed program of function type as a morphism between the interpretations of its domain and codomain, `box(...)` is the anonymous form of quotation, which packages a closed Garlene term, and `idfun A` is the identity function defined with `gdef`. The statement mentions no Garlene proposition, only its interpretation. In the same way, the monad laws of `Delay`, all proven in the proof mode, assemble into an instance `DELAY.monad : CategoryTheory.Monad` $\mathcal{I}$ of Mathlib's monads on the topos of trees (Section 6.1).

The model also works in the other direction: A morphism of $\mathcal{I}$ can be used as a constant in Garlene (using `ax`), and two programs with equal interpretations are equal in the logic. This is how, for instance, the Mathlib morphisms in the statement of the monad laws enter Garlene. For closed propositions there is, moreover, a small extraction interface (Section 5.2). A proposition is *valid* if its interpretation holds in the model; every theorem of the proof mode is valid by soundness; a theorem `P → Q` and a valid `P` give a valid `Q`; a valid $\triangleright P$ gives a valid P (validity quantifies over all steps at once, so a $\triangleright$ may be stripped, even though $\triangleright P \longrightarrow P$ is not a theorem of the logic); and a valid `pure` φ, for a Lean proposition φ, gives φ itself. The last two rules are how results leave the guarded model: The steps taken during a proof are discarded, and Section 6.3 uses this to turn a guarded adequacy proof into a statement about an operational semantics that mentions neither Garlene nor the topos and is stated using Lean's proposition universe.

4 The Garlene Core Calculus

In this section, we give a precise account of Garlene's core calculus, a simply typed λ-calculus with products, sums, the later modality, and a type Ω of propositions. In addition, Garlene has two type formers Δ and ax for referring to objects from the surrounding Lean context:

$$\text{types} \quad \tau, \sigma ::= \Delta\, T \mid \tau \times \sigma \mid \tau \oplus \sigma \mid \tau \to \sigma \mid \blacktriangleright \tau \mid \Omega \mid \mathsf{ax}\ I$$

$\Delta\, T$ embeds a Lean type T as a discrete Garlene type, and ax I embeds a presheaf I of the model as a Garlene type. Expressions include both terms and propositions:

$$\begin{aligned}\text{expr.}\quad e, \Phi ::= {} & \delta(a) \mid e \odot e' \mid \mathsf{pure}\ e \mid \mathsf{var}\ (p,q) \mid \mathsf{lam}_\tau\ e \\ \mid {} & \mathsf{app}_\tau\ e\ e' \mid \mathsf{delay}\ e \mid \mathsf{adv}_n\ e \mid \mathsf{fix}_{\blacktriangleright\tau}\ e \mid \langle e, e' \rangle \\ \mid {} & \mathsf{proj}\ e\ d \mid \mathsf{inl}_\tau\ e \mid \mathsf{inr}_\tau\ e \mid \mathsf{case}_{\tau,\sigma}\ e\ f\ g \\ \mid {} & \Phi \wedge \Phi' \mid \Phi \vee \Phi' \mid \Phi \to \Phi' \mid \forall_\tau\, \Phi \mid \exists_\tau\, \Phi \\ \mid {} & \mathsf{lift}\ e \mid \top \mid \bot \mid e =_\tau e' \mid \mathsf{ax}_\tau\ f\end{aligned}$$

The syntax uses de Bruijn indices, which are pairs of the form $\mathsf{var}\ (p,q)$ to account for the Kripke-style typing of Garlene, which we describe in Section 4.1. Four of the expression formers refer to Lean. The expression $\delta(a)$ embeds a Lean value $a : T$ at the discrete type $\Delta\, T$, which together with $\odot : \Delta\,(S \to T) \to \Delta\, S \to \Delta\, T$ gives discrete types an applicative structure. The expression `pure` e turns an embedded Lean proposition into a Garlene proposition, and $\mathsf{ax}_\tau\ f$ uses a global element $f : \mathbf{1} \longrightarrow [\![\tau]\!]$ of the model as a closed constant of type τ. The remaining formers are standard, with the exception of delay, adv_n, and fix, which interact with the context in a way that we describe in Section 4.1. lift e turns a delayed proposition $e : \blacktriangleright\Omega$ into a proposition now, allowing us to obtain a propositional version of the later modality (Section 5).

4.1 Type System

Figure 3 shows the typing rules of Garlene, which form a Kripke-style type system [12, 19] whose contexts are not lists of types but rather *stacks of frames* (or alternatively, *stacks of contexts*). A stack $\vec{\Gamma}$ is a list $\Gamma_0; \ldots; \Gamma_n$ of frames Γ_i, and in turn a frame Γ is a list $\tau_0, \ldots, \tau_m$ of types. The Kripke-style type system is a presentation of the Fitch-style type system of Clocked Type Theory [3] on which Garlene is based. The ";" separators between frames correspond to the ticks in Clocked Type Theory, or the padlocks in the Fitch-style type system of Clouston [9]. They represent the 'steps' of Section 3. From here on we use the terminology 'tick' rather than 'step'.

Since we use de Bruijn variables, we append both the stacks and individual frames from the left for consistency. Consequently, de Bruijn variables are pairs (p,q), where p accounts for the frames to the left of the variable's frame, and q refers to the entry τ_q in that frame. We write $\vec{\Gamma}$, $\vec{\Delta}$ for contexts and Γ, Δ, Ψ for single frames. The context of any derivable judgment has at least one frame, and the closed context is the single empty frame denoted $\cdot$. This is the reason for the side condition $|\vec{\Gamma}| > 0$ in several rules. The expression `delay` e is typechecked with a new empty frame pushed onto the context, which means that inside a delay we are one tick in the future, and the variables of the outer frames are one tick in the past. The expression $\mathsf{adv}_n\ e$ typechecks $e : \blacktriangleright\tau$ in the context with the innermost n frames removed and produces a value of type τ. It reaches back n ticks for a delayed value and uses it now. Expressions with binders, such as lam or $\forall$, introduce a variable in the innermost frame. Finally, fix e binds the recursive occurrence in the innermost

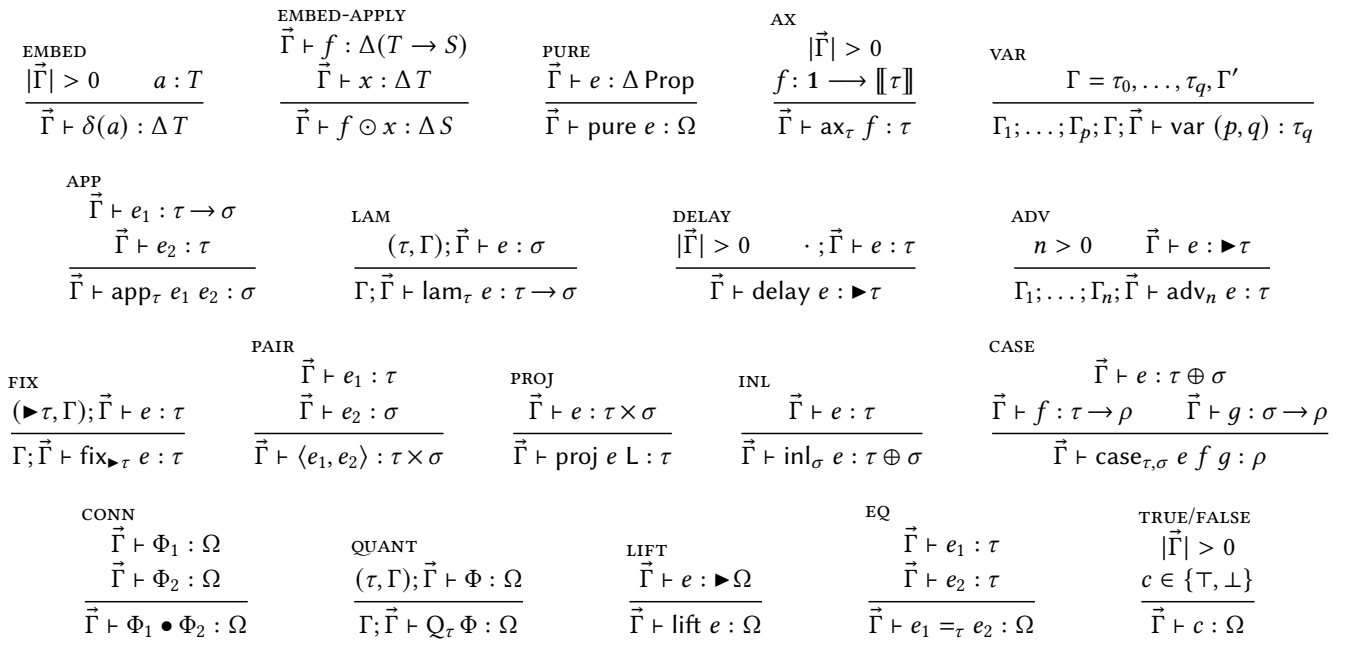


Figure 3. Typing rules. A context $\vec{\Gamma}$ is a stack of frames Γ. PROJ is shown for L; PROJ for R returns σ; INR is dual to INL. In CONN, $\bullet$ ranges over $\wedge$, $\vee$, $\to$; in QUANT, Q ranges over $\forall$, $\exists$. In AX, $f : \mathbf{1} \longrightarrow [\![\tau]\!]$ is a global element of the model. Outside this figure, we omit the type annotations of app, inl, inr and case. Well-typedness side conditions are omitted except where they matter.

frame at type $\blacktriangleright\tau$. Consequently, the recursive occurrence can be used only at type $\blacktriangleright\tau$, typically by applying adv to it under a delay, as shown in Section 3.

Fitch-style programming with delay and adv gives an intuitive way of programming with the $\blacktriangleright$ modality. The cost of this is that the theory of substitutions is more complex, as we describe in the following section.

4.2 Weakening and Substitution

Renamings and substitutions are defined by the following grammars:

$$\begin{array}{lll} \text{renamings} & \rho & ::= \mathsf{id} \mid \rho \circ \rho' \mid \rho\uparrow \mid \rho^+ \mid \rho\Uparrow^n \mid \rho^\triangleright \\ \text{frame subst.} & \theta & ::= \varepsilon \mid e \cdot \theta \\ \text{stacked subst.} & \Theta & ::= \theta \mid \theta ::_n \Theta \end{array}$$

Apart from identity and composition renamings, there are two dimensions, with two operations in each. Within a frame, one can weaken past an entry ($\uparrow$) or lift a renaming under a binder ($^+$); across frames, one can weaken past n frames ($\Uparrow^n$) or lift a renaming under a delay ($^\triangleright$). Substitutions are structured in the same way and inherit the notation. A frame substitution θ populates a single frame, while a stacked substitution Θ populates a stack of frames. The target of a frame substitution is a stack Δ consisting of a single frame. Lifting under a binder is not a primitive substitution but a derived construction, $\theta^+ = \mathsf{var}\,(0,0) \cdot \theta\langle\mathsf{id}\uparrow\rangle$, where the renaming acts on θ entrywise; Θ^+ applies this to the innermost frame of Θ and leaves the remaining frames unchanged. Figure 4 defines well-formedness judgments for renamings and substitutions, as well as their respective actions $e\langle\rho\rangle$ and $e[\Theta]$ on a term e.

The substitution action $e[\Theta]$ will appear in the provability and equality judgments of the logic in Section 5, but only in the special case where $\Theta = e' \cdot \varepsilon$, which we write as $e[e']$ instead of $e[e' \cdot \varepsilon]$. Most clauses of the renaming and substitution actions are congruences, and Figure 4 shows the cases that cross a tick or a variable binder. Binders and delay lift the renaming into the new slot or frame. For $\mathsf{adv}_n\ e$, the number n is replaced by the number of frames onto which the renaming maps the n frames that adv crosses, which we call the offset $\rho(n)$, and the action continues on e with the part of the renaming that lies beyond those frames, which we call the cut $\rho\restriction_n$.[1] Offset and cut extend to substitutions. $\Theta(n)$ is the sum of the frame counts m of the first n components $::_m$ of Θ, and $\Theta\restriction_n$ is Θ with those components removed. Stacked substitutions are the K-substitutions of Hu et al. [19] in a simply typed setting: The frame $::_n$ is their modal extension $\sigma; \Uparrow^n$, and the clause for adv in Figure 4 is their rule for pushing a substitution under unbox. Unlike their calculus, where K-substitutions are part of the syntax and typing and conversion are defined mutually, our renamings and substitutions are functions on raw terms, and preservation of typing is a lemma rather than a rule.

[1] Our cut and offset are the truncation $\sigma \mid n$ and the truncation offset $O(\sigma, n)$ of Hu et al. [19]; what they call the modal offset is our n in adv_n.

Renamings and substitutions enjoy the expected properties. Typing is preserved: From $\vec{\Delta} \vdash e : \tau$ and $\rho : \vec{\Gamma} \Rightarrow \vec{\Delta}$ we obtain $\vec{\Gamma} \vdash e\langle\rho\rangle : \tau$, which every rule that moves a term between contexts relies on. Renamings and substitutions are trivial on closed terms: A closed, well-typed term is a fixed point of every well-typed renaming and substitution, so a quoted constant may be placed at any offset, which we exploit for quotation below. Index arithmetic occurs only in the two clauses for adv, and every other clause passes the renaming either unchanged or lifted. So, renamings re-index variables, adjust the numbers in adv_n, and leave every other former unchanged. Renamings also satisfy the expected identity and composition laws, so the administrative renamings produced by tactics compose and cancel by simplification with these laws, and renamings do not appear in user-displayed goals. Cut and offset satisfy coherence laws similar to those of the truncations of Hu et al. [19]: $\rho\restriction_{a+b} = (\rho\restriction_a)\restriction_b$ and $\rho(a) + (\rho\restriction_a)(b) = \rho(a{+}b)$.

Typechecking and Quotation. Garlene's syntax is defined as an inductive type `EXPR`, which is extrinsically typed by a Lean predicate `TYPED : CTX → EXPR → TYPE → Prop`. A Garlene term defined via `gdef` is typechecked by the Garlene implementation, which produces a proof of `TYPED [.] e` τ. Such a closed Garlene term e of type τ is then packaged as a Lean value of type `SYNT` τ. Since `TYPED` is a proposition, the derivation is proof-irrelevant: Derivations are never compared, and two packages with the same term are equal. This provides a mechanism to reuse previously defined Garlene programs. More concretely, when users refer to a previously defined program c, the elaborator inserts it as $[c]_{\mathrm{s}}^{k,m}$, which re-embeds c : `SYNT` τ into a context of k frames whose outermost frame has m slots, as the weakening $c\langle\uparrow^m\Uparrow^{k-1}\rangle$. This allows tactics to re-index quotations freely and to compare them by their offsets. Until the user explicitly unfolds c, all administrative weakenings and substitutions produced by tactics simply re-index quotations around c. In the surface syntax a quotation is written `[c]`$_{\mathrm{s}}$ for a Lean value `c : SYNT` τ, and `box(e)`, its anonymous form, which quotes a closed term in place. Only closed terms are quoted; a definition that needs free variables takes them as arguments. Quoting open terms would require quotations to carry a renaming or substitution.

4.3 Semantics

The model is the topos of trees $\mathcal{I}$ [5], the presheaves over ω. Recall that the objects of $\mathcal{I}$ are natural number indexed sets, together with maps $X(n+1) \to X(n)$ for all n. Types are interpreted by objects $[\![\tau]\!]$, contexts by objects $[\![\vec{\Gamma}]\!]$, and a partial function

$$\texttt{expr_interp}\ \vec{\Gamma}\ e\ \tau : \texttt{Part}\,([\![\vec{\Gamma}]\!] \longrightarrow [\![\tau]\!])$$

interprets terms, where `Part` is Mathlib's type of partial values. The interpretation is partial because typing is extrinsic: It is a function on raw syntax, and a separate lemma states that it is defined whenever $\vec{\Gamma} \vdash e : \tau$. The later modality is interpreted by the $\blacktriangleright$ functor defined as $\blacktriangleright X(0) = 1$, $\blacktriangleright X(n+1) = X(n)$, ΔT by the constant presheaf on T, and Ω by the subobject classifier. The formers that refer to Lean are interpreted directly: $\delta(a)$ by the constant map at a, $\odot$ by the application map of the constant presheaf, pure and lift by maps into the subobject classifier, and $\mathsf{ax}_\tau\ f$ by f precomposed with the terminal map $[\![\vec{\Gamma}]\!] \longrightarrow \mathbf{1}$. The frame structure of contexts is interpreted through the left adjoint $\blacktriangleleft$ of the later functor, the *earlier* functor: A frame is the product of the interpretations of its types, and each frame boundary is one application of $\blacktriangleleft$, so $[\![\Gamma; \vec{\Gamma}]\!] = [\![\Gamma]\!] \times \blacktriangleleft\ [\![\vec{\Gamma}]\!]$. The former delay is then interpreted by transposition along $\blacktriangleleft \dashv \blacktriangleright$, turning $\blacktriangleleft [\![\vec{\Gamma}]\!] \longrightarrow [\![\tau]\!]$ into $[\![\vec{\Gamma}]\!] \longrightarrow \blacktriangleright [\![\tau]\!]$. The former adv_n projects the context onto its outer frames and transposes in the other direction; for $n > 1$ the surplus boundaries are first collapsed by iterating force : $\blacktriangleleft X \longrightarrow X$, the transpose of next : $X \longrightarrow \blacktriangleright X$. The same transpose interprets the use of a variable across a frame boundary. The interpretation of fix is a morphism $X^{\blacktriangleright X} \longrightarrow X$ defined by recursion on the stage: At stage 0 the object $\blacktriangleright X$ is a singleton, so the fixed point is determined trivially, and at stage $n+1$ it is obtained from stage n.

4.4 Guarded Recursive Types

The topos of trees is a presheaf category, and so has a Hofmann–Streicher universe $\mathcal{U}$ [18]. As observed by Birkedal and Møgelberg [4], guarded recursive types can be encoded semantically as fixed points of maps $\blacktriangleright\mathcal{U} \to \mathcal{U}$. To allow users to program such maps, we include the universe into the Garlene calculus as `UNIV` := ax $\mathcal{U}$, and add codes for type formers as shown in Figure 5. For example, given a Garlene type A, the code `Delay.code` A : `SYNT UNIV` of the guarded delay monad of Section 3 can be defined as

$$\mathsf{fix}\ X.\ \mathsf{SUM}\ \langle \mathsf{CODE}\ A, \mathsf{LATER}\ (\mathsf{delay}\ (\mathsf{adv}_1\ X))\rangle. \quad (2)$$

The body of this fixed point has type $\blacktriangleright$`UNIV` $\to$ `UNIV` because the only code former that involves the later modality, `LATER`, takes a *delayed* code as argument, and X occurs only as $\mathsf{adv}_1\ X$ under a delay.

Since the Garlene calculus is simply typed, we cannot treat arbitrary terms of the universe as types, as is done in dependently typed calculi. However, *closed* terms of type `UNIV` can be treated as types, using the bijective correspondence mapping global elements A of $\mathcal{U}$ to small objects $\mathsf{El}(A)$ in the topos of trees. In particular, given a closed code c : `SYNT UNIV`, we can construct a Garlene type ax $\mathsf{El}([\![c]\!])$. Moreover, the code formers of Figure 5 are interpreted as morphisms in the topos of trees corresponding to their type

$$\textsc{id}\ \frac{|\vec{\Gamma}| > 0}{\mathrm{id} : \vec{\Gamma} \Rightarrow \vec{\Gamma}} \qquad \textsc{comp}\ \frac{\rho_1 : \vec{\Delta} \Rightarrow \vec{\Psi} \quad \rho_2 : \vec{\Gamma} \Rightarrow \vec{\Delta}}{\rho_1 \circ \rho_2 : \vec{\Gamma} \Rightarrow \vec{\Psi}} \qquad \textsc{local-weaken}\ \frac{\rho : \Gamma; \vec{\Gamma} \Rightarrow \Delta; \vec{\Delta}}{\rho\uparrow : (\tau, \Gamma); \vec{\Gamma} \Rightarrow \Delta; \vec{\Delta}}$$

$$\textsc{cons}\ \frac{\rho : \Gamma; \vec{\Gamma} \Rightarrow \Delta; \vec{\Delta}}{\rho^+ : (\tau, \Gamma); \vec{\Gamma} \Rightarrow (\tau, \Delta); \vec{\Delta}} \qquad \textsc{global-lift}\ \frac{\rho : \vec{\Gamma} \Rightarrow \vec{\Delta}}{\rho^\triangleright : \cdot; \vec{\Gamma} \Rightarrow \cdot; \vec{\Delta}}$$

$$\textsc{global-shift}\ \frac{\rho : \vec{\Gamma} \Rightarrow \vec{\Delta}}{\rho\Uparrow^n : \Psi_1; \ldots; \Psi_n; \vec{\Gamma} \Rightarrow \vec{\Delta}} \qquad \textsc{snil}\ \frac{|\vec{\Gamma}| > 0}{\varepsilon : \vec{\Gamma} \Rrightarrow \cdot} \qquad \textsc{scons}\ \frac{\vec{\Gamma} \vdash e : \tau \quad \theta : \vec{\Gamma} \Rrightarrow \Delta}{e \cdot \theta : \vec{\Gamma} \Rrightarrow \tau, \Delta} \qquad \textsc{frame}\ \frac{n > 0 \qquad \Theta : \vec{\Gamma} \Rrightarrow \vec{\Delta} \quad \theta : \Psi_1; \ldots; \Psi_n; \vec{\Gamma} \Rrightarrow \Delta}{\theta ::_n \Theta : \Psi_1; \ldots; \Psi_n; \vec{\Gamma} \Rrightarrow \Delta; \vec{\Delta}}$$

$$(\mathsf{delay}\ e)\langle\rho\rangle = \mathsf{delay}\ e\langle\rho^\triangleright\rangle \qquad (\mathsf{adv}_n\ e)\langle\rho\rangle = \mathsf{adv}_{\rho(n)}\ e\langle(\rho\uparrow_n)\rangle \qquad (\mathsf{fix}\ e)\langle\rho\rangle = \mathsf{fix}\ e\langle\rho^+\rangle$$
$$(\mathsf{delay}\ e)[\Theta] = \mathsf{delay}\ e[(\varepsilon ::_1 \Theta)] \qquad (\mathsf{adv}_n\ e)[\Theta] = \mathsf{adv}_{\Theta(n)}\ e[(\Theta\uparrow_n)] \qquad (\mathsf{lam}_\tau\ e)[\Theta] = \mathsf{lam}_\tau\ e[\Theta^+]$$

Figure 4. Typing of renamings (ID to GLOBAL-SHIFT) and of substitutions (SNIL, SCONS, FRAME), and representative clauses of the two actions, namely those that cross a tick and one binder case for each. In GLOBAL-SHIFT and FRAME, n is the number of added frames $\Psi_1, \ldots, \Psi_n$. A frame substitution θ and a stacked one Θ use the same judgment.

```
UNIV.DISCRETE T     : SYNT UNIV
UNIV.CODE τ         : SYNT UNIV
UNIV.PROD, UNIV.SUM : SYNT (UNIV × UNIV ⟶ UNIV)
UNIV.LATER          : SYNT (▶ UNIV ⟶ UNIV)
UNIV.ARR            : SYNT (UNIV × UNIV ⟶ UNIV)
```

Figure 5. Code formers.

formers, as captured by decoding lemmas, *e.g.*,

$$\mathsf{El}(\llbracket \mathsf{ARR}\ \langle [c]_s, [d]_s \rangle \rrbracket) = \mathsf{El}(\llbracket c \rrbracket) \to \mathsf{El}(\llbracket d \rrbracket)$$
$$\mathsf{El}(\llbracket \mathsf{LATER}\ (\mathsf{delay}\ [c]_s) \rrbracket) = \blacktriangleright \mathsf{El}(\llbracket c \rrbracket)$$

These can be used to prove that types $N := \mathsf{ax}\ \mathsf{El}(\llbracket \mathtt{N.code} \rrbracket)$ defined using fixed points `N.code` of maps on the universe, such as Equation (2) above, satisfy the corresponding type equation $\llbracket N \rrbracket = \llbracket F[N] \rrbracket$, where F is the body before its translation to codes. Using this, one can define terms

```
N.MK   : SYNT (F[N] ⟶ N)
N.PROJ : SYNT (N ⟶ F[N])
```

and prove two equations `N.MK_PROJ` and `N.PROJ_MK` stating that they are mutually inverse.

To simplify defining guarded recursive types, we provide the command `gtype`, which automates these steps: From a declaration such as

```
gtype Delay (A : TYPE) := ν X. [A] ⊕ ▶X
```

it generates the code, the type, the type equation, `MK`, `PROJ` and their equations. The body is written using the same type syntax as the signatures of `gdef` and supports type constructors such as core calculus types `[τ]`, discrete types `Δ T`, products, sums, function types and ▶. The recursion variable must occur under ▶, but, unlike in Lean's inductive types, it need not occur positively, as in the domain of Section 6.3, which follows Paviotti et al. [30]:

```
gtype Dom := ν X. Δ Unit ⊕ Δ Unit ⊕ ▶X ⊕ ▶(X ⟶ X)
```

Its two summands `Δ Unit` are the unit value and an error. The `gtype` command supports only a single recursion variable, but this is not a limitation of the calculus: Since a code is an ordinary fixed point at type `UNIV`, codes for other guarded recursive types can be written directly in the calculus.

5 The Logic

In this section we describe the logic of Garlene, its proof rules, and its proof mode implemented in Lean.

5.1 Rules

A Garlene proposition Φ is a term of type Ω. Quantifiers range over all types, including Ω and $\tau \to \Omega$, so our logic is higher-order. The provability judgment $\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi$ references a hypothesis context $\vec{\Psi}$, which, like the variable context $\vec{\Gamma}$, is structured into a stack of frames. The stack represents a separation of logical assumptions into time frames, like in the proof example of Section 3.3, and these time frames are synchronized with the time frames for variables in rules. This means that every rule will maintain the invariants that $|\vec{\Gamma}| = |\vec{\Psi}|$, and that every hypothesis is a well-typed proposition in the context up to the frame where it was introduced. We write this as the following judgment:

$$\Gamma_0; \ldots; \Gamma_n \vdash \Psi_0; \ldots; \Psi_n \triangleq \forall k.\, \forall \Phi \in \Psi_k.\, \Gamma_k; \ldots; \Gamma_n \vdash \Phi : \Omega$$

In particular, a hypothesis can only mention variables defined in or before its own time frame. This synchronization of contexts is automatic in dependent type theories like Clocked Type Theory [3], where both variables and logical assumptions live in the same context. Our logic was inspired by Clocked Type Theory, but to the best of our knowledge, this is the first Fitch-style logic over a Fitch-style type theory with separate contexts for variables and hypotheses.

Figure 6 shows the proof rules of our logic, with well-typedness side conditions omitted. A Garlene theorem is declared using the command `gtheorem`, which elaborates to an ordinary Lean theorem whose statement encodes $\cdot \mid \cdot \vdash \Phi$, *i.e.*, Φ is a closed term of type Ω that is provable in the empty hypothesis context.

The rules of the logic are those of natural deduction for higher-order logic, adapted to the frame structure of contexts at two points. First, the rule ASM uses a hypothesis from frame n as $\Phi\langle\Uparrow^n\rangle$. That is, hypotheses cross frames by weakening, and what was provable n ticks ago remains provable now. As with adv_n, building the weakening into the rule avoids explicit structural rules for the stack [19]. Second, the rules that introduce a variable, namely FORALL-INTRO, EXISTS-ELIM and SUM-ELIM, extend the innermost variable frame and weaken the innermost hypothesis frame with $\uparrow$ accordingly. In addition, SUM-ELIM provides each branch with the equation $e = \mathsf{inl}\ x$, respectively $e = \mathsf{inr}\ x$; this is the hypothesis h that gcases displays in Figure 2. The rules INL-INJ and INL-INR-DISJ state that the injections into sum types are injective and disjoint.

Later. We write $\triangleright\Phi$ for $\mathsf{lift}\ (\mathsf{delay}\ \Phi)$. The rule LIFT-INTRO moves into a fresh frame: In order to prove $\triangleright\Phi$ now, we must prove Φ one tick later, with the whole context still available but shifted by one frame. LATER-ELIM is the counterpart of ADV: If $\mathsf{lift}\ e$ holds n ticks earlier, then $\mathsf{adv}_n\ e$ holds now. For $e = \mathsf{delay}\ \Phi$, BETA-DELAY (from Figure 7) reduces $\mathsf{adv}_n\ e$ to a weakening of Φ. Monotonicity of $\triangleright$, its commutation with conjunction, and $\triangleright\forall x.\, \Phi \vdash \forall x.\ \triangleright\, \Phi$ are derivable from LIFT-INTRO and LATER-ELIM. Commutation with disjunction (LATER-OR) is a separate rule, which holds in the model. The rule DELAY-EQ states that to prove equality of delayed terms it suffices to prove equality of values produced later. Together with ETA-DELAY, it gives extensionality for terms of type $\blacktriangleright\tau$: $t_1 = t_2$ follows from $\triangleright(\mathsf{adv}_1\ t_1 = \mathsf{adv}_1\ t_2)$.

Löb Induction. Guarded recursion at the level of proofs is the rule LOEB-IND: In order to prove Φ, one may hypothesize $\triangleright(\Phi\langle\Uparrow^1\rangle)$, that is, the statement itself one tick later.

Lean. Two rules connect the logic to Lean. From a Lean proof of P, the rule PURE-INTRO proves $\ulcorner P \urcorner$, which abbreviates pure $\delta(P)$. The rule FORALL-INTRO-POINTS proves $\forall_{\Delta\, T}\, \Phi$ from a Lean proof of $\Phi[\delta(a)]$ for every $a : T$; it is sound because $\Delta\, T$ is interpreted as a constant presheaf.

Equality. Propositional equality is backed by a separate judgment $\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau$ of judgmental equality. It enters the logic through EQ-DEF, which is the reflexivity rule for propositional equality. Propositional equality is eliminated by EQ-ELIM, *i.e.*, by Leibniz's rule through a substitution $\Phi[e]$. Figure 7 shows the guarded fragment of the rules. Altogether they make the judgment an equivalence relation, provide β-laws for each type former and η-laws for functions, products and the later modality, and include a congruence rule for every term former, including delay, adv and fix. In the β-rule for $\blacktriangleright$, the weakening $(\Uparrow^{n-1})^{\blacktriangleright}\uparrow^{|\Gamma_1|}$ carries the term e to the expanded context, and the side condition of the rule mirrors the typing of adv. The congruence rules, including under ticks, allow our meta-level normalizer to rewrite anywhere in a term (Section 5.3). Judgmental equality is sound with respect to the model, and derivability of an equation implies that both sides are well-typed.

Two rules go beyond the β, η and congruence rules: UNFOLD, which unfolds term-level fixed points, and AX, which equates two terms whose interpretations in the model coincide. We use the latter rule in three places: for the equations between MK and PROJ generated by `gtype`; for function extensionality (`gfunext`), which we prove in the model; and for equating two quoted programs with equal denotations, which is how we reason about Mathlib morphisms inside Garlene (Section 5.2, Section 6.1). Everything else is derived within the logic.

5.2 Soundness and Extraction

Sequents are interpreted in the internal logic of $\mathcal{I}$. A proposition over $\vec{\Gamma}$ denotes a morphism $[\![\vec{\Gamma}]\!] \longrightarrow \Omega$. A hypothesis in frame Ψ_d is a proposition over $\Gamma_d; \ldots; \Gamma_n$; it is moved to $[\![\vec{\Gamma}]\!]$ by applying $\blacktriangleleft$ and composing with a map $\blacktriangleleft\ \Omega \longrightarrow \Omega$, d times, where the map shifts truth values by a stage. A hypothesis stack denotes the conjunction of its hypotheses. Since these are defined using the interpretation of terms, they are partial functions, but well-formedness of the judgment $\vec{\Gamma} \vdash \vec{\Psi}$ implies that $[\![\vec{\Psi}]\!]$ is well-defined. Internal entailment $P \vdash_{\mathcal{I}} Q$ holds when, at every stage n and for every element γ of $[\![\vec{\Gamma}]\!]$ at that stage, the sieve $P\,\gamma$ is contained in the sieve $Q\,\gamma$.

Theorem 5.1 (Soundness). *If* $\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi$ *and* $\vec{\Gamma} \vdash \vec{\Psi}$ *then* $[\![\vec{\Psi}]\!] \vdash_{\mathcal{I}} [\![\Phi]\!]$.

The proof is by induction on derivations. The case of EQ-DEF uses soundness of the judgmental equality. As a corollary we obtain consistency: $\bot$ is not derivable in the empty context.

Soundness is used to export Garlene theorems into Lean. When exporting closed theorems, we simply use the fact that $\vec{\Gamma} \vdash \vec{\Psi}$ holds vacuously when $\vec{\Gamma}$ and $\vec{\Psi}$ consist of the empty frame $\cdot$. A closed proposition P is *semantically valid* when

$$\mathtt{Valid}\ P \triangleq [\![P]\!] \text{ is defined and } \top \vdash_{\mathcal{I}} [\![P]\!].$$

Validity satisfies the following rules:

$$\frac{\cdot \mid \cdot \vdash P}{\mathtt{Valid}\ P}\ \text{OF-GOAL} \qquad \frac{\mathtt{Valid}\ (\ulcorner \varphi \urcorner)}{\varphi}\ \text{PURE} \qquad \frac{\mathtt{Valid}\ (\triangleright P)}{\mathtt{Valid}\ P}\ \text{LATER}$$

The rule OF-GOAL is soundness at the singleton empty frame, LATER allows one to peel $\triangleright$ off a valid proposition, and PURE is the exit: A valid $\ulcorner \varphi \urcorner$ yields the Lean proposition φ itself. The rule LATER may look suspicious, since $\triangleright\Phi \vdash \Phi$ is not provable in the logic. It is sound nevertheless, because

$$\textsc{asm}\ \frac{\Phi \in \Psi}{\vec{\Gamma} \mid \Psi_1; \ldots; \Psi_n; \Psi; \vec{\Psi} \vdash \Phi\langle \Uparrow^n \rangle} \qquad \textsc{true-intro}\ \frac{}{\vec{\Gamma} \mid \vec{\Psi} \vdash \top} \qquad \textsc{false-elim}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \bot}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi}$$

$$\textsc{and-intro}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1 \qquad \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_2}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1 \wedge \Phi_2} \qquad \textsc{and-elim}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1 \wedge \Phi_2}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_i}$$

$$\textsc{or-intro}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_i}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1 \vee \Phi_2} \qquad \textsc{or-elim}\ \frac{\vec{\Gamma} \mid (P, \Psi); \vec{\Psi} \vdash \Phi \qquad \vec{\Gamma} \mid (Q, \Psi); \vec{\Psi} \vdash \Phi}{\vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash P \vee Q \qquad \vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \Phi}$$

$$\textsc{impl-intro}\ \frac{\vec{\Gamma} \mid (\Phi_1, \Psi); \vec{\Psi} \vdash \Phi_2}{\vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \Phi_1 \to \Phi_2} \qquad \textsc{impl-elim}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1 \to \Phi_2 \qquad \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_1}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi_2}$$

$$\textsc{forall-intro}\ \frac{(\tau, \Gamma); \vec{\Gamma} \mid \Psi\langle\uparrow\rangle; \vec{\Psi} \vdash \Phi}{\Gamma; \vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \forall_\tau \Phi} \qquad \textsc{forall-intro-points}\ \frac{\forall a : T. \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi[\delta(a)]}{\vec{\Gamma} \mid \vec{\Psi} \vdash \forall_{\Delta T} \Phi}$$

$$\textsc{forall-elim}\ \frac{\Gamma; \vec{\Gamma} \mid \vec{\Psi} \vdash \forall_\tau \Phi \qquad \Gamma; \vec{\Gamma} \vdash e : \tau}{\Gamma; \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi[e]} \qquad \textsc{exists-intro}\ \frac{\Gamma; \vec{\Gamma} \vdash e : \tau \qquad \Gamma; \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi[e]}{\Gamma; \vec{\Gamma} \mid \vec{\Psi} \vdash \exists_\tau \Phi}$$

$$\textsc{exists-elim}\ \frac{\Gamma; \vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \exists_\tau \Phi \qquad (\tau, \Gamma); \vec{\Gamma} \mid (\Phi, \Psi\langle\uparrow\rangle); \vec{\Psi} \vdash Q\langle\uparrow\rangle}{\Gamma; \vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash Q}$$

$$\textsc{sum-elim}\ \frac{\Gamma; \vec{\Gamma} \vdash e : \tau \oplus \sigma \qquad (\tau, \Gamma); \vec{\Gamma} \mid (e\langle\uparrow\rangle = \mathsf{inl}\ x, \Psi\langle\uparrow\rangle); \vec{\Psi} \vdash \Phi\langle\uparrow\rangle \qquad (\sigma, \Gamma); \vec{\Gamma} \mid (e\langle\uparrow\rangle = \mathsf{inr}\ x, \Psi\langle\uparrow\rangle); \vec{\Psi} \vdash \Phi\langle\uparrow\rangle}{\Gamma; \vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \Phi}$$

$$\textsc{lift-intro}\ \frac{\cdot; \vec{\Gamma} \mid \cdot; \vec{\Psi} \vdash \Phi}{\vec{\Gamma} \mid \vec{\Psi} \vdash \triangleright\Phi}$$

$$\textsc{later-elim}\ \frac{n > 0 \qquad \vec{\Gamma} \mid \vec{\Psi} \vdash \mathsf{lift}\ e}{\Gamma_1; \ldots; \Gamma_n; \vec{\Gamma} \mid \Psi_1; \ldots; \Psi_n; \vec{\Psi} \vdash \mathsf{adv}_n\ e} \qquad \textsc{later-or}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \triangleright(P \vee Q)}{\vec{\Gamma} \mid \vec{\Psi} \vdash \triangleright P \vee \triangleright Q}$$

$$\textsc{loeb-ind}\ \frac{\vec{\Gamma} \mid (\triangleright(\Phi\langle\Uparrow^1\rangle), \Psi); \vec{\Psi} \vdash \Phi}{\vec{\Gamma} \mid \Psi; \vec{\Psi} \vdash \Phi} \qquad \textsc{delay-eq}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \triangleright(e_1 =_\tau e_2)}{\vec{\Gamma} \mid \vec{\Psi} \vdash \mathsf{delay}\ e_1 =_{\blacktriangleright\tau} \mathsf{delay}\ e_2}$$

$$\textsc{eq-def}\ \frac{\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau}{\vec{\Gamma} \mid \vec{\Psi} \vdash e_1 =_\tau e_2} \qquad \textsc{eq-elim}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash e_1 =_\tau e_2 \qquad \vec{\Gamma} \mid \vec{\Psi} \vdash \Phi[e_1]}{\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi[e_2]}$$

$$\textsc{pure-intro}\ \frac{P \text{ holds in Lean}}{\vec{\Gamma} \mid \vec{\Psi} \vdash \ulcorner P \urcorner} \qquad \textsc{inl-inj}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \mathsf{inl}\ a = \mathsf{inl}\ a'}{\vec{\Gamma} \mid \vec{\Psi} \vdash a = a'} \qquad \textsc{inl-inr-disj}\ \frac{\vec{\Gamma} \mid \vec{\Psi} \vdash \mathsf{inl}\ a = \mathsf{inr}\ b}{\vec{\Gamma} \mid \vec{\Psi} \vdash \bot}$$

Figure 6. The provability judgment $\vec{\Gamma} \mid \vec{\Psi} \vdash \Phi$. In AND-ELIM and OR-INTRO, $i \in \{1, 2\}$; INL-INJ has a dual variant for inr; $\triangleright\Phi$ abbreviates lift (delay Φ); $\ulcorner P \urcorner$ abbreviates pure $\delta(P)$. Well-typedness side conditions are omitted except where they matter.

$$\textsc{beta-delay}\ \frac{n > 0 \qquad m = |\Gamma_1| \qquad \cdot; \vec{\Gamma} \vdash e : \tau}{\Gamma_1; \ldots; \Gamma_n; \vec{\Gamma} \vdash \mathsf{adv}_n\ (\mathsf{delay}\ e) \equiv e\langle(\Uparrow^{n-1})^\triangleright \uparrow^m\rangle : \tau} \qquad \textsc{eta-delay}\ \frac{\vec{\Gamma} \vdash e : \blacktriangleright\tau}{\vec{\Gamma} \vdash e \equiv \mathsf{delay}\ (\mathsf{adv}_1\ e) : \blacktriangleright\tau}$$

$$\textsc{unfold}\ \frac{(\blacktriangleright\tau, \Gamma); \vec{\Gamma} \vdash e : \tau}{\Gamma; \vec{\Gamma} \vdash \mathsf{fix}\ e \equiv e[\mathsf{delay}\ (\mathsf{fix}\ e)\langle\Uparrow^1\rangle] : \tau} \qquad \textsc{cong-delay}\ \frac{|\vec{\Gamma}| > 0 \qquad \cdot; \vec{\Gamma} \vdash e \equiv e' : \tau}{\vec{\Gamma} \vdash \mathsf{delay}\ e \equiv \mathsf{delay}\ e' : \blacktriangleright\tau}$$

$$\textsc{cong-adv}\ \frac{n > 0 \qquad \vec{\Gamma} \vdash e \equiv e' : \blacktriangleright\tau}{\Gamma_1; \ldots; \Gamma_n; \vec{\Gamma} \vdash \mathsf{adv}_n\ e \equiv \mathsf{adv}_n\ e' : \tau}$$

$$\textsc{cong-fix}\ \frac{(\blacktriangleright\tau, \Gamma); \vec{\Gamma} \vdash e \equiv e' : \tau}{\Gamma; \vec{\Gamma} \vdash \mathsf{fix}\ e \equiv \mathsf{fix}\ e' : \tau} \qquad \textsc{beta-embed-apply}\ \frac{|\vec{\Gamma}| > 0}{\vec{\Gamma} \vdash \delta(f) \odot \delta(x) \equiv \delta(f\ x) : \Delta S}$$

$$\textsc{ax}\ \frac{\vec{\Gamma} \vdash e : \tau \qquad \vec{\Gamma} \vdash e' : \tau \qquad [\![e]\!] = [\![e']\!]}{\vec{\Gamma} \vdash e \equiv e' : \tau}$$

Figure 7. The equational judgment $\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau$. Not shown are the equivalence rules, the β- and η-rules for functions and products, the β-rules for sums, and the pointwise congruence rules; see Appendix A for more rules. In BETA-EMBED-APPLY, $f : T \to S$ and $x : T$ are Lean values. In AX, the third premise is equality of the two interpretations in the model.

validity quantifies over all stages: $\triangleright P$ at stage $n + 1$ is P at stage n, and therefore a valid $\triangleright P$ gives a valid P. The ticks accumulated in the proof are discarded on the way out, and an extracted theorem does not mention the deep embedding of the syntax or the provability judgment. In Section 6.3 we use these rules to obtain an adequacy theorem stated in Lean.

5.3 The Proof Mode

Working directly with a deep embedding requires explicit manipulations of syntax. For instance, introducing a universally quantified variable requires weakening the hypotheses to account for the new variable. Garlene's proof mode provides a set of tactics that take care of these operations behind the scenes, allowing users to work with named variables and hypotheses, as in Figure 2.

Simplification. The simplifier contracts β-redexes using Garlene's equational theory. For a term e of type τ, it returns a term e' together with a derivation of $\vec{\Gamma} \vdash e \equiv e' : \tau$. The reductions include function application, product projections, case expressions, and cancellation of adv against delay; fixed points are not unfolded by the simplifier, so that repeated simplification terminates. The Garlene simplifier does not use η-laws; they are applied as lemmas, such as `Delay.delay_eta` in Figure 2. We leave other reduction strategies for future work. The tactic gsimpl repeats simplification until no reduction applies outside the bodies of fixed points. Unfolding a fixed point is implemented by a separate tactic: gfix applies UNFOLD of Figure 7 to exposed fixed points, that is, those not under delay, adv, a quotation or another fixed point, and then simplifies, so that the adv_1 in the body cancels the delay around the unfolded definitions.

These reductions, and rules that extend the context, introduce weakenings and substitutions. We push them to variables and quotations using the syntactic laws of Section 4.2. We also use this mechanism with the other tactics that produce weakenings and substitutions: those that extend the context or enter a frame and weaken the hypotheses (*e.g.*, gintro), and those that instantiate bound variables by substitution (*e.g.*, gapply). Quotations are handled as described in Section 4.2.

Rewriting. One of the most used tactics in our framework, grewrite h, replaces an occurrence of one side of an equation h in the goal by the other side, like Lean's rw. The step is justified by EQ-ELIM, which needs a *motive*: the goal with the occurrence abstracted into a fresh variable. In Garlene, constructing the motive is slightly different from non-modal languages, because the occurrence may be under delay and adv, and the fresh variable must be referred to across frames.

Consider grewrite HtIf in Figure 2. If we write a for the recursive call, HtIf states $\mathsf{delay}\ a = \mathsf{delay}\ (\mathsf{adv}_1\ v)$. The tactic abstracts the occurrence of delay a in the goal, producing a *motive*: a proposition in the context extended by a fresh variable. In named notation, the resulting motive is

$$M(z) := [\mathsf{Delay.MK}\ A]_{\mathsf{s}}\ (\mathsf{inr}\ z) = d$$

where z names the fresh variable $\mathsf{var}\ (0, 0)$. The original goal is $M[\mathsf{delay}\ a]$ and the new goal is $M[\mathsf{delay}\ (\mathsf{adv}_1\ v)]$. The tactic reduces the former to the latter, so it applies EQ-ELIM to the new goal with the equation $\mathsf{delay}\ (\mathsf{adv}_1\ v) = \mathsf{delay}\ a$.

Here the occurrence is at the top level. Had we rewritten a itself, under the delay, the motive would be

$$[\mathsf{Delay.MK}\ A]_{\mathsf{s}}\ (\mathsf{inr}\ (\mathsf{delay}\ \mathsf{var}\ (1, 0))) = d,$$

with the fresh variable one frame back, because it is located under delay. The tactic implementation traverses the term to find the occurrence of a term to rewrite, while keeping a tick depth k, initially 0, and a renaming ρ, initially $\uparrow$, which is the weakening by the fresh variable as seen from the current position. Under delay, k increases by one and ρ becomes $\rho^{\triangleright}$. Under adv_n with $n \le k$, the offset becomes $\rho(n)$, and the traversal continues at depth $k - n$ with renaming $\rho\uparrow_n$. When $n > k$, the operand belongs to a frame preceding the fresh variable, so the traversal does not enter it. A matching subterm is replaced by $\mathsf{var}\ (k, 0)$. Non-binding constructors are traversed structurally; binders (λ, $\forall$, $\exists$, fix) are not, so an occurrence under a binder is rewritten after introducing its variable, *e.g.*, with gintro. Subterms containing no replacement are renamed by ρ. Finally, the tactic typechecks the motive and proves that substituting the two sides recovers the original and rewritten propositions. When arguments of an equation are omitted, a structural search tries the target and then its subterms, taking the first match that determines all remaining arguments. Quotations are compared by their SYNT values.

Logical Tactics. The tactic glöb IH applies LOEB-IND by adding the hypothesis $\mathsf{IH} : \triangleright P$, where P is the current goal. The tactic gnext transforms the goal $\triangleright Q$ into Q in a new frame (LIFT-INTRO), and also applies DELAY-EQ if the goal is $\mathsf{delay}\ a = \mathsf{delay}\ b$. In a later frame, gadv H as G adds $\mathsf{G} : P$ for a hypothesis $\mathsf{H} : \triangleright P$, by LATER-ELIM followed by BETA-DELAY. Introduction, case analysis and intermediate assertions use gintro, gcases and gassert; the case patterns follow Lean's rcases, with branch equations for sum-typed terms. The tactic gpoints a for a goal $\forall_{\Delta\, T}\, \Phi$ introduces a Lean variable $a : T$ and leaves $\Phi[\delta(a)]$ as the Garlene goal, with the Garlene contexts unchanged. The tactic gembed applies PURE-INTRO, reducing a Garlene goal $\ulcorner P \urcorner$ to the Lean goal P.

Implementation. Garlene's provability judgment is implemented as an inductive family `PROVES : CTX → PCTX → EXPR → Prop`, which in turn is used by a wrapper with additional indices that record variable and hypothesis names for elaboration and display. The tactics are Lean metaprograms using Qq [15] to construct typed Lean expressions. They share operations for looking up hypotheses, weakening them into the current context, and constructing the corresponding derivations. Typing side conditions are discharged by the typechecker and arithmetic conditions by decision procedures. Lean's kernel checks the assembled proof term. The tactics are therefore sound, but not complete with respect to the rules of Figure 6.

6 Case Studies

We apply Garlene to two case studies: We first return to the delay monad of Section 3 to prove its monad laws and to characterize it as a free delay algebra. We then interpret a λ-calculus with fixed points in a guarded recursive domain and prove soundness and adequacy. None of these results are new [30], but they illustrate the expressive power of Garlene.

6.1 The Guarded Delay Monad

We extend the example of Section 3 with definitions for the left inclusion $\mathsf{ret} : A \to \mathsf{Delay}\, A$, the right inclusion $\mathsf{step} : \blacktriangleright(\mathsf{Delay}\, A) \to \mathsf{Delay}\, A$, and the monadic bind operation $\mathsf{bind} : (A \to \mathsf{Delay}\, B) \to \mathsf{Delay}\, A \to \mathsf{Delay}\, B$. We prove the functor laws for map and the monad laws for bind and ret by unfolding their guarded definitions. The left unit law is simply the definition of bind on ret, while right unit, associativity, and preservation of composition by map use Löb induction and case analysis as in Figure 2.

Interpreting these operations on Delay gives us a monad on $\mathcal{I}$. Showing that Delay satisfies the monad laws uses both connections between Garlene and the model: ax and soundness, in a round trip. Consider, for example, naturality of the unit. A morphism in the topos of trees, $f\colon X \longrightarrow Y$, is identified with its transpose $\mathbf{1} \longrightarrow Y^X = [\![\mathsf{ax}\, X \to \mathsf{ax}\, Y]\!]$, as the closed term $\overline{f} \triangleq \mathsf{ax}_{\mathsf{ax}\, X \to \mathsf{ax}\, Y}\ f$. We must show $[\![\mathsf{map}\ \overline{f}]\!] \circ [\![\mathsf{ret}_X]\!] = [\![\mathsf{ret}_Y]\!] \circ f$, where the subscript records the type argument $\mathsf{ax}\, X$ or $\mathsf{ax}\, Y$ of ret. At this point, we treat $\overline{f}$ as an opaque constant in Garlene. We then prove the closed equation $\mathsf{ret}_Y \circ \overline{f} = \mathsf{map}\ \overline{f} \circ \mathsf{ret}_X$ between functions $\mathsf{ax}\, X \to \mathsf{Delay}\ (\mathsf{ax}\, Y)$ in Garlene: By function extensionality it suffices to show $\mathsf{ret}_Y\ (\overline{f}\ x) = \mathsf{map}\ \overline{f}\ (\mathsf{ret}_X\ x)$, which is the equation of map on ret obtained by unfolding the fixed point. Soundness exports the equation as an equality of denotations using two facts: the denotation of a composite is the composite of the denotations, and $[\![\overline{f}]\!] = f$. The other fields of the Mathlib CategoryTheory.Monad on $\mathcal{I}$ are obtained in the same way.

6.2 Delay Is the Free Delay Algebra

A *delay algebra* is an object B with a map $s : \blacktriangleright B \to B$. For $f : A \to B$, guarded recursion defines an extension $\mathsf{ext}\ s\ f : \mathsf{Delay}\, A \to B$ satisfying

$$\begin{aligned} \mathsf{ext}\ s\ f\ (\mathsf{ret}\ a) &= f\ a \\ \mathsf{ext}\ s\ f\ (\mathsf{step}\ w) &= s\ (\mathsf{delay}\ (\mathsf{ext}\ s\ f\ (\mathsf{adv}_1\ w))). \end{aligned}$$

These equations can be used to express $\mathsf{ext}\ s\ f : \mathsf{Delay}\, A \to B$ as a fixed point. For uniqueness, suppose h is another map satisfying the above equations. We prove $\forall d.\ h\ d = \mathsf{ext}\ s\ f\ d$ by Löb induction and case analysis on PROJ d. The return case follows from the first equation. The step case reduces to equality of delayed recursive calls, where gadv makes the induction hypothesis available. Function extensionality then gives $h = \mathsf{ext}\ s\ f$. Freeness provides an alternative route to showing that Delay is a monad.

6.3 A λ-Calculus with Fixed Points

In this example, we define a model for a simply typed call-by-name λ-calculus with unit and fixed points, and prove it computationally adequate. Since Garlene does not have inductive types, we represent syntax, typing judgments and the operational semantics using Lean types. Writing λ_{fix} for this object language, we define an inductive family $\mathsf{Lam}\, n$ of λ_{fix} terms in the scope of n variables, an inductive type of λ_{fix} types $a ::= \mathsf{unit} \mid a \to b$, and a typing judgment $\mathsf{Typing}\ \Gamma\ e\ a$. We write $e \to e'$ for one reduction step and $e \twoheadrightarrow e'$ for its reflexive-transitive closure. The grammar and operational semantics of λ_{fix} are shown below. $i : \mathsf{Fin}\, n$ is a de Bruijn index, the bodies under binders are terms of type $\mathsf{Lam}\ (n+1)$ and $t[u]$ substitutes u for the variable 0 of t.

$$e ::= \mathsf{unit} \mid \mathsf{var}\ i \mid \mathsf{app}\ e\ e' \mid \mathsf{lam}\ e \mid \mathsf{fix}\ e$$

$$\begin{aligned} &\mathsf{app}\ (\mathsf{lam}\ t)\ u \to t[u] \\ &\mathsf{app}\ t\ u \to \mathsf{app}\ t'\ u \quad \text{if } t \to t' \\ &\mathsf{fix}\ t \to t[\mathsf{fix}\ t] \end{aligned}$$

We interpret λ_{fix} in the guarded type Dom of Section 4.4. Its four summands represent the unit value, an error, delayed computations, and delayed functions. We write unit, error, thunk w and lam g for the four injections into Dom. An environment $\mathsf{Env}\, n$ contains n elements of Dom. The interpretation $\mathsf{interp}_n : \mathsf{Lam}\, n \to \mathsf{SYNT}\ (\mathsf{Env}\, n \to \mathsf{Dom})$ is defined by recursion on the λ_{fix} syntax in Lean, using guarded recursion to interpret λ_{fix} fixed points. The interpretation is given by the following clauses, omitting quotations, where $\rho : \mathsf{Env}\, n$ is an environment and ρ_i is its i-th entry:

$$\begin{aligned} &\mathsf{interp}_n\ \mathsf{unit}\ \rho &&= \mathsf{unit} \\ &\mathsf{interp}_n\ (\mathsf{var}\ i)\ \rho &&= \rho_i \\ &\mathsf{interp}_n\ (\mathsf{app}\ t\ u)\ \rho &&= \mathsf{apply}\ (\mathsf{interp}_n\ t\ \rho)\ (\mathsf{interp}_n\ u\ \rho) \\ &\mathsf{interp}_n\ (\mathsf{lam}\ t)\ \rho &&= \mathsf{lam}\ (\mathsf{delay}\ (\lambda v.\ \mathsf{interp}_{n+1}\ t\ \langle \rho, v\rangle)) \\ &\mathsf{interp}_n\ (\mathsf{fix}\ t)\ \rho &&= \mathsf{fix}\ y. \\ &\qquad \mathsf{thunk}\ (\mathsf{delay}\ (\mathsf{interp}_{n+1}\ t\ \langle \rho, \mathsf{adv}_1\ y\rangle))). \end{aligned}$$

Application $\mathsf{apply} : \mathsf{Dom} \to \mathsf{Dom} \to \mathsf{Dom}$ is defined in Garlene by guarded recursion. It returns error on error and unit, and otherwise satisfies

$$\begin{aligned} \mathsf{apply}\ (\mathsf{lam}\ g)\ x &= \mathsf{thunk}\ (\mathsf{delay}\ (\mathsf{adv}_1\ g\ x)) \\ \mathsf{apply}\ (\mathsf{thunk}\ w)\ x &= \mathsf{thunk}\ (\mathsf{delay}\ (\mathsf{apply}\ (\mathsf{adv}_1\ w)\ x)). \end{aligned}$$

The soundness property states that each λ_{fix} reduction $e \to e'$ corresponds to one thunk in the model:

$$\cdot \mid \cdot \vdash \forall \rho.\ \mathsf{interp}_n\ e\ \rho = \mathsf{thunk}\ (\mathsf{delay}\ (\mathsf{interp}_n\ e'\ \rho)).$$

The proof is by induction on the reduction derivation, using substitution lemmas for the interpretation. In other words, we build a Garlene proof in the proof mode by Lean induction over the step relation.

Adequacy. To prove adequacy, we relate closed λ_{fix} terms to the model. Given a relation R between λ_{fix} terms and elements of Dom, we define Exp $R\ e\ p$, which requires a λ_{fix} reduct related to each value exposed by p. Omitting quotations and λ_{fix} term embeddings, we define

$$\mathsf{Exp}\ R\ e\ p \triangleq \mathsf{wp}\ p\ (\lambda v.\ \exists w.\ \ulcorner e \twoheadrightarrow w \urcorner \wedge R\ w\ v).$$

The weakest precondition wp is defined by guarded recursion. It applies its postcondition to unit and functions, is false on error, and postpones the condition by one tick at a thunk:

$$\mathsf{wp}\ (\mathsf{thunk}\ t)\ \Phi = \triangleright\big(\mathsf{wp}\ (\mathsf{adv}_1\ t)\ \Phi\big).$$

We define a logical relation Val_a by recursion on λ_{fix} types and use it as R in the Exp relation. At unit, it requires $w =$ unit and $p =$ unit. At an arrow type, p must be lam g; applications to related arguments must be related one tick later:

$$\mathsf{Val}_{a\to b}\ w\ p \triangleq \exists g.\ p = \mathsf{lam}\ g \wedge$$
$$\forall y.\ \triangleright\ \big(\forall q.\ \mathsf{Exp}\ \mathsf{Val}_a\ y\ q \to \mathsf{Exp}\ \mathsf{Val}_b\ (\mathsf{app}\ w\ y)\ (\mathsf{adv}_1\ g\ q)\big)$$

The relation SubstOk $\Gamma\ \sigma\ \rho$ between a closing substitution $\sigma : \mathsf{Fin}\, n \to \mathsf{Lam}\, 0$ and an environment $\rho : \mathsf{Env}\, n$ is defined by recursion on n; for each variable i it relates $\sigma\, i$ to the i-th entry of ρ by Exp $\mathsf{Val}_{\Gamma\, i}$. The fundamental lemma relates every well-typed term to its interpretation under related substitutions and environments.

Lemma 6.1 (Fundamental lemma)**.** *For every typing context* Γ *of length* n*, term* $e : \mathsf{Lam}\, n$ *and type* a *with* Typing $\Gamma\ e\ a$*, and every closing substitution* $\sigma : \mathsf{Fin}\, n \to \mathsf{Lam}\, 0$*,*

$$\cdot \mid \cdot \vdash \forall \rho : \mathsf{Env}\, n.\ \mathsf{SubstOk}\ \Gamma\ \sigma\ \rho \to \mathsf{Exp}\ \mathsf{Val}_a\ (e[\sigma])\ (\mathsf{interp}_n\ e\ \rho).$$

We prove it by induction on typing derivations in Lean, carrying out each case in the proof mode. In the fixed point case, the typing induction hypothesis applies to the body, while Löb induction assumes the property of the recursive term one tick later. Unfolding exposes a thunk; under it, gadv gives the relation needed for the recursive environment entry.

We use denote e for the closed interpretation of e in Dom, and step r for thunk (delay $[r]_{\mathsf{s}}$) where $r :$ Dom. The Lean predicate Runs $k\ p\ q$ states that p is q delayed by k applications of step:

$$\mathsf{EqG}\ p\ q \triangleq (\cdot \mid \cdot \vdash [p]_{\mathsf{s}} = [q]_{\mathsf{s}}) \qquad \mathsf{Runs}\ 0\ p\ q \triangleq \mathsf{EqG}\ p\ q$$
$$\mathsf{Runs}\ (k{+}1)\ p\ q \triangleq \exists r.\ \mathsf{EqG}\ p\ (\mathsf{step}\ r) \wedge \mathsf{Runs}\ k\ r\ q.$$

Adequacy then states that if the denotation of a term terminates, also the term itself terminates.

Theorem 6.2 (Adequacy)**.** *For every closed term* e *of type* unit *and every* $k \in \mathbb{N}$*,*

$$\mathsf{Runs}\ k\ (\mathsf{denote}\ e)\ \mathsf{unit} \implies e \twoheadrightarrow \mathsf{unit}.$$

The proof uses the validity rules described in Section 5.2. Soundness of Garlene makes the closed instance of the fundamental lemma valid, and by induction on k we transport validity of the fundamental lemma across the equations given by Runs. At unit, the logical relation implies the valid proposition $\ulcorner e \twoheadrightarrow \mathsf{unit} \urcorner$, which we can extract to Lean.

Adequacy allows us to relate proofs about the denotational semantics of λ_{fix} back to the operational semantics of λ_{fix}, that is, to the relation $\twoheadrightarrow$ defined in Lean.

7 Conclusion and Future Work

Garlene aims to serve as a practical tool for programming and reasoning with guarded recursion. Learning from previous work on embedding multimode type theories in Agda [7, 8], we focused on providing intuitive syntax and good performance that scales to larger applications. Our experience – documented in the two case studies above – suggests that Garlene is capable of mechanizing larger formal developments using guarded recursion. For example, checking the λ_{fix} development (about 1,600 lines) with Lean 4.33.1 takes about 30 seconds on a machine with an Intel Core Ultra 5 225U (12 cores and 16 GB of RAM) when Lean uses all available cores (its default), and about 60 seconds when it is restricted to a single thread. We hope that Garlene will find use for mechanizing existing and future work that uses guarded type theory as metalanguage.

We also plan to explore the use of Garlene as a language for programming with and reasoning about coinductive types. One way to do this could be to expand Garlene from the single-clock version to multiple clocks [2]. Another would be to use the denotational model, exploiting the fact that many coinductive types can be expressed as sets of global elements of guarded recursive types in the topos of trees. For these applications, it will be interesting to extend the model of guarded recursion by increasing the indexing to ordinals larger than ω. This should also allow for programming with and reasoning about coinductive types whose definitions involve constructions such as finite or countable powersets or finite distributions [25]. Such a model could also be used to reason about liveness properties as in Transfinite Iris [31].

Finally, apart from presenting Garlene, this paper also describes key implementation techniques that enabled the performance and usability characteristics of Garlene. We hope that these insights can help future work on embedding domain-specific calculi in general-purpose proof assistants.

Use of LLMs. Large language models were used to fill out some of the proofs in the Lean mechanization, primarily the case-heavy renaming and substitutions proofs used for simplification, and for debugging tactics. All relevant theorem statements were written by hand, so we do not expect risk of the LLMs introducing errors.

Acknowledgments

This work was supported by grant number 2032-00134B from the Independent Research Fund Denmark. We thank David Thrane Christiansen for helpful discussions.

References

[1] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. doi:10.1145/504709.504712

[2] Robert Atkey and Conor McBride. 2013. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, Boston, Massachusetts, USA, 197–208. ISBN 978-1-4503-2326-0. doi:10.1145/2500365.2500597

[3] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. doi:10.1109/LICS.2017.8005097

[4] Lars Birkedal and Rasmus Ejlers Møgelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE Computer Society, 213–222. doi:10.1109/LICS.2013.27

[5] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 55–64. doi:10.1109/LICS.2011.16

[6] Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). doi:10.2168/LMCS-1(2:1)2005

[7] Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2022. Sikkel: Multimode Simple Type Theory as an Agda Library. In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022 (EPTCS, Vol. 360)*, Jeremy Gibbons and Max S. New (Eds.). 93–112. doi:10.4204/EPTCS.360.5

[8] Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. 2025. BiSikkel: A Multimode Logical Framework in Agda. *Proc. ACM Program. Lang.* 9, POPL (2025), 210–240. doi:10.1145/3704844

[9] Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 258–275. doi:10.1007/978-3-319-89366-2_14

[10] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4, 10 (2017), 3127–3170. http://collegepublications.co.uk/ifcolog/?00019

[11] Karl Crary. 2026. The Istari Proof Assistant. Retrieved November 18, 2026 from https://istarilogic.org/ Version 1.3c.

[12] Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. doi:10.1145/382780.382785

[13] Markus de Medeiros, Sergei Stepanenko, Zongyuan Liu, Oliver Soeser, Fernando Leal, Alvin Tang, Max Vistrup, Ralf Jung, Mario Carneiro, Joseph Tassarotti, Michael Sammler, and Lars Birkedal. 2026. Iris in Lean. (2026). arXiv:2609.24252 [cs.LO] doi:10.48550/arXiv.2609.24252

[14] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. doi:10.1007/978-3-030-79876-5_37

[15] Gabriel Ebner. 2021. Qq: Quotation and Antiquotation for Lean 4. Lean 4 library. https://github.com/leanprover-community/quote4 Accessed 2026-08-30.

[16] Eric Giovannini, Tingting Ding, and Max S. New. 2025. Denotational Semantics of Gradual Typing using Synthetic Guarded Domain Theory. *Proc. ACM Program. Lang.* 9, POPL (2025), 772–801. doi:10.1145/3704863

[17] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2021. Multimodal Dependent Type Theory. *Log. Methods Comput. Sci.* 17, 3 (2021). doi:10.46298/LMCS-17(3:11)2021

[18] Martin Hofmann and Thomas Streicher. 1997. Lifting Grothendieck Universes. Unpublished note. https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf

[19] Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by evaluation for modal dependent type theory. *J. Funct. Program.* 33 (2023). doi:10.1017/S0956796823000060

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

[21] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. doi:10.1145/3236772

[22] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. 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, 205–217. doi:10.1145/3009837.3009855

[23] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. 2022. Greatest HITs: Higher inductive types in coinductive definitions via induction under clocks. In *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 42:1–42:13. doi:10.1145/3531130.3533359

[24] Runming Li, Yue Yao, and Robert Harper. 2026. Mechanizing Synthetic Tait Computability in Istari. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '26)*. Association for Computing Machinery, New York, NY, USA, 231–247. ISBN 979-8-4007-2341-4. doi:10.1145/3779031.3779085

[25] Rasmus Ejlers Møgelberg. 2026. Multi-Clocked Guarded Recursion Beyond ω. In *31st International Conference on Types for Proofs and Programs (TYPES 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 384)*, Fredrik Nordvall Forsberg and James McKinna (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:22. ISBN 978-3-95977-441-3. doi:10.4230/LIPIcs.TYPES.2025.11

[26] Rasmus Ejlers Møgelberg and Marco Paviotti. 2019. Denotational semantics of recursive types in synthetic guarded domain theory. *Math. Struct. Comput. Sci.* 29, 3 (2019), 465–510. doi:10.1017/S0960129518000087

[27] Rasmus Ejlers Møgelberg and Andrea Vezzosi. 2021. Two Guarded Recursive Powerdomains for Applicative Simulation. In *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021 (EPTCS, Vol. 351)*, Ana Sokolova (Ed.). 200–217. doi:10.4204/EPTCS.351.13

[28] Hiroshi Nakano. 2000. A Modality for Recursion. In *15th Annual IEEE Symposium on Logic in Computer Science, Santa Barbara, California, USA, June 26-29, 2000*. IEEE Computer Society, 255–266. doi:10.1109/LICS.2000.855774

[29] Wojciech Nawrocki, Joseph Hua, Mario Carneiro, Yiming Xu, Spencer Woolfson, Shuge Rong, Sina Hazratpour, and Steve Awodey. 2026. A Certifying Proof Assistant for Synthetic Mathematics in Lean. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2026, Rennes, France, January 12-13,*

2026, Kathrin Stark, Yannick Zakowski, Nikhil Swamy, and Nicolas Tabareau (Eds.). ACM, 88–103. doi:10.1145/3779031.3779087

[30] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. 2015. A Model of PCF in Guarded Type Theory. In *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015 (Electronic Notes in Theoretical Computer Science, Vol. 319)*, Dan R. Ghica (Ed.). Elsevier, 333–349. doi:10.1016/J.ENTCS.2015.12.020

[31] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: resolving an existential dilemma of step-indexed separation logic. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 80–95. doi:10.1145/3453483.3454031

[32] Philipp Stassen, Rasmus Ejlers Møgelberg, Maaike Zwart, Alejandro Aguirre, and Lars Birkedal. 2025. Modelling Recursion and Probabilistic Choice in Guarded Type Theory. *Proc. ACM Program. Lang.* 9, POPL (2025), 1417–1445. doi:10.1145/3704884

[33] Sergei Stepanenko, Patrick Bahr, and Rasmus Ejlers Møgelberg. 2026. *Garlene: Guarded Recursion in Lean.* doi:10.5281/zenodo.24506493

[34] The mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. ACM, New Orleans, LA, USA. doi:10.1145/3372885.3373824

[35] Niccolò Veltri and Andrea Vezzosi. 2023. Formalizing CCS and π-calculus in Guarded Cubical Agda. *J. Log. Algebraic Methods Program.* 131 (2023), 100846. doi:10.1016/J.JLAMP.2022.100846

[36] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 87:1–87:29. doi:10.1145/3341691

A Equational Rules

Figure 7 shows only the most important rules of the equational judgment $\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau$. Figure 8 below gives a more comprehensive set of rules, which only elides standard congruence rules.

$$\textsc{rfl}\ \frac{\vec{\Gamma} \vdash e : \tau}{\vec{\Gamma} \vdash e \equiv e : \tau} \qquad \textsc{sym}\ \frac{\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau}{\vec{\Gamma} \vdash e_2 \equiv e_1 : \tau} \qquad \textsc{tran}\ \frac{\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau \qquad \vec{\Gamma} \vdash e_2 \equiv e_3 : \tau}{\vec{\Gamma} \vdash e_1 \equiv e_3 : \tau}$$

$$\textsc{beta-embed-apply}\ \frac{|\vec{\Gamma}| > 0}{\vec{\Gamma} \vdash \delta(f) \odot \delta(x) \equiv \delta(f\ x) : \Delta\, S}$$

$$\textsc{beta-lam}\ \frac{\Gamma; \vec{\Gamma} \vdash e' : \tau \qquad (\tau, \Gamma); \vec{\Gamma} \vdash e : \sigma}{\Gamma; \vec{\Gamma} \vdash \mathsf{app}\ (\mathsf{lam}_\tau\ e)\ e' \equiv e[e'] : \sigma} \qquad \textsc{eta-lam}\ \frac{\vec{\Gamma} \vdash e : \tau \to \sigma}{\vec{\Gamma} \vdash e \equiv \mathsf{lam}_\tau\ (\mathsf{app}\ e\langle\uparrow\rangle\ (\mathsf{var}\ (0,0))) : \tau \to \sigma}$$

$$\textsc{beta-delay}\ \frac{n > 0 \qquad m = |\Gamma_1| \qquad \cdot; \vec{\Gamma} \vdash e : \tau}{\Gamma_1; \ldots; \Gamma_n; \vec{\Gamma} \vdash \mathsf{adv}_n\ (\mathsf{delay}\ e) \equiv e\langle(\Uparrow^{n-1})^{\triangleright}\uparrow^m\rangle : \tau} \qquad \textsc{eta-delay}\ \frac{\vec{\Gamma} \vdash e : \blacktriangleright\tau}{\vec{\Gamma} \vdash e \equiv \mathsf{delay}\ (\mathsf{adv}_1\ e) : \blacktriangleright\tau}$$

$$\textsc{unfold}\ \frac{(\blacktriangleright\tau, \Gamma); \vec{\Gamma} \vdash e : \tau}{\Gamma; \vec{\Gamma} \vdash \mathsf{fix}\ e \equiv e[\mathsf{delay}\ (\mathsf{fix}\ e)\langle\Uparrow^1\rangle] : \tau}$$

$$\textsc{beta-prod}\ \frac{\vec{\Gamma} \vdash e : \tau \qquad \vec{\Gamma} \vdash e' : \sigma}{\vec{\Gamma} \vdash \mathsf{proj}\ \langle e, e'\rangle\ \mathsf{L} \equiv e : \tau} \qquad \textsc{eta-prod}\ \frac{\vec{\Gamma} \vdash e : \tau \times \sigma}{\vec{\Gamma} \vdash e \equiv \langle \mathsf{proj}\ e\ \mathsf{L}, \mathsf{proj}\ e\ \mathsf{R}\rangle : \tau \times \sigma}$$

$$\textsc{beta-case}\ \frac{\vec{\Gamma} \vdash a : \tau \qquad \vec{\Gamma} \vdash f : \tau \to \rho \qquad \vec{\Gamma} \vdash g : \sigma \to \rho}{\vec{\Gamma} \vdash \mathsf{case}\ (\mathsf{inl}\ a)\ f\ g \equiv \mathsf{app}\ f\ a : \rho}$$

$$\textsc{cong-lam}\ \frac{(\tau, \Gamma); \vec{\Gamma} \vdash e \equiv e' : \sigma}{\Gamma; \vec{\Gamma} \vdash \mathsf{lam}_\tau\ e \equiv \mathsf{lam}_\tau\ e' : \tau \to \sigma} \qquad \textsc{cong-forall}\ \frac{(\tau, \Gamma); \vec{\Gamma} \vdash \Phi \equiv \Phi' : \Omega}{\Gamma; \vec{\Gamma} \vdash \forall_\tau\, \Phi \equiv \forall_\tau\, \Phi' : \Omega}$$

$$\textsc{cong-fix}\ \frac{(\blacktriangleright\tau, \Gamma); \vec{\Gamma} \vdash e \equiv e' : \tau}{\Gamma; \vec{\Gamma} \vdash \mathsf{fix}\ e \equiv \mathsf{fix}\ e' : \tau} \qquad \textsc{cong-delay}\ \frac{|\vec{\Gamma}| > 0 \qquad \cdot; \vec{\Gamma} \vdash e \equiv e' : \tau}{\vec{\Gamma} \vdash \mathsf{delay}\ e \equiv \mathsf{delay}\ e' : \blacktriangleright\tau}$$

$$\textsc{cong-adv}\ \frac{n > 0 \qquad \vec{\Gamma} \vdash e \equiv e' : \blacktriangleright\tau}{\Gamma_1; \ldots; \Gamma_n; \vec{\Gamma} \vdash \mathsf{adv}_n\ e \equiv \mathsf{adv}_n\ e' : \tau} \qquad \textsc{ax}\ \frac{\vec{\Gamma} \vdash e : \tau \qquad \vec{\Gamma} \vdash e' : \tau \qquad [\![e]\!] = [\![e']\!]}{\vec{\Gamma} \vdash e \equiv e' : \tau}$$

Figure 8. The equational judgment $\vec{\Gamma} \vdash e_1 \equiv e_2 : \tau$. BETA-PROD and BETA-CASE are shown for L and inl; the R/inr versions are dual, and CONG-FORALL has an $\exists$ dual. Not shown: pointwise congruence rules in an unchanged context for app, $\odot$, pure, pairing, projections, injections, case, the connectives, equality, and lift. In BETA-EMBED-APPLY, $f : T \to S$ and $x : T$ are Lean values. In AX, the third premise is equality of the two interpretations in the model.

Received 2026-09-10; accepted 2026-11-10