Mechanizing Synthetic Tait Computability in Istari

Runming Li
Carnegie Mellon University
Pittsburgh, USA
runmingl@cs.cmu.edu

Yue Yao
Carnegie Mellon University
Pittsburgh, USA
yueyao@cs.cmu.edu

Robert Harper
Carnegie Mellon University
Pittsburgh, USA
rwh@cs.cmu.edu

Abstract

Categorical gluing is a powerful technique for proving meta-theorems of type theories such as canonicity and normalization. Synthetic Tait Computability (STC) provides an abstract treatment of the complex gluing models by internalizing the gluing category into a modal dependent type theory with a phase distinction. This work presents a mechanization of STC in the Istari proof assistant. Istari is a Martin-Löf-style extensional type theory with equality reflection, which avoids much of the explicit transport reasoning typically found in intensional proof assistants. This work develops a reusable library for synthetic phase distinction, including modalities, extension types, and strict glue types, and applies it to two case studies: (1) a canonicity model for dependent type theory with dependent products and booleans with large elimination, and (2) a Kripke canonicity model for the cost-aware logical framework. Our results demonstrate that the core STC constructions can be formalized essentially verbatim in Istari, preserving the elegance of the on-paper arguments while ensuring machine-checked correctness.

CCS Concepts: • **Theory of computation → Type theory**; **Categorical semantics**.

Keywords: gluing, synthetic Tait computability, logical relations, extensional type theory, Istari, equality reflection, meta-theory, cost-aware logical framework

ACM Reference Format:
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), January 12–13, 2026, Rennes, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779031.3779085

1 Introduction

The past decade has seen significant advancements in the meta-theory of programming languages, particularly for dependent type theories, due to the use of the categorical *gluing* technique [30] in programming languages. Traditionally, meta-theorems of programming languages such as canonicity and normalization are proved using syntactic *logical relations* arguments *à la* Tait [91]. These are families of predicates or relations defined by induction on the structure of types, with respect to an operational semantics or a reduction system. When the syntax of a programming language is in more semantic and algebraic presentations, such as locally cartesian closed categories for dependent type theories, the gluing technique provides a categorical and *proof-relevant* generalization to syntactic logical relations. Concretely, the gluing technique constructs a *gluing* model over the syntactic model of the programming language, along a suitable functor to a semantic category, such as the category of sets. For example, gluing along the global sections functor yields a proof-relevant logical relations model that establishes canonicity [21, 53, 70], the property that all closed terms of boolean type are either true or false. This technique has been applied to prove various canonicity and normalization results, including simply-typed λ-calculus [33, 90], System F [9], dependent type theory [8, 27, 58], and univalent type theory [80].

1.1 Gluing

This subsection provides a brief overview of the gluing technique. For reasons of space, we assume familiarity with basic notions from category theory such as hom-functors and the Yoneda embedding, and refer the reader to classic textbooks such as Awodey [16], Riehl [76]. Consider a category $\mathcal{T}$ that represents the syntax of an object language: its objects are types, and its morphisms are judgmental equivalence classes of terms. In this setting, a canonicity theorem may be formulated as follows:

Theorem 1.1 (Canonicity)**.** *For every closed term of boolean type, represented as a morphism $b : 1_{\mathcal{T}} \to \mathsf{bool}_{\mathcal{T}}$ in $\mathcal{T}$, either $b = \mathsf{true}_{\mathcal{T}}$ or $b = \mathsf{false}_{\mathcal{T}}$.*

To ensure the category representing the syntax has good categorical properties, we can embed $\mathcal{T}$ into presheaf category $\mathbf{PSh}(\mathcal{T})$ via the Yoneda embedding: $よ : \mathcal{T} \hookrightarrow \mathbf{PSh}(\mathcal{T})$. The proof strategy is to construct a model of the object language in the Artin gluing [22] $\mathcal{G} = \mathbf{Set} \downarrow \Gamma$, the comma category over the global sections functor:

$$\Gamma : \mathbf{PSh}(\mathcal{T}) \to \mathbf{Set}$$
$$\Gamma(A) = \mathsf{Hom}(よ(1_{\mathcal{T}}), A)$$

under the identification $よ(1_{\mathcal{T}}) = \mathbf{1}_{\mathbf{PSh}(\mathcal{T})}$ because Yoneda preserves limits. To be explicit, an object in $\mathcal{G}$ is a triple

CPP '26, Rennes, France

ACM ISBN 979-8-4007-2341-4/2026/01
https://doi.org/10.1145/3779031.3779085

(S, A, f) where $S \in \mathbf{Set}$, $A \in \mathbf{PSh}(\mathcal{T})$, and $f : S \to \Gamma(A)$; a morphism from (S, A, f) to (S', A', f') is a pair of morphisms $(h : S \to S', t : A \to A')$ such that the diagram as shown below commutes. An object in $\mathcal{G}$ in the form of $(S, \text{ɣ}(B), f)$ is to be thought of as a family of sets S_b indexed by morphisms $\text{ɣ}(b) : \text{ɣ}(1_{\mathcal{T}}) \to \text{ɣ}(B)$, *i.e.* a proof-relevant predicate on closed terms of type B.

$$\begin{array}{ccc} S & \xrightarrow{h} & S' \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle f'} \\ \Gamma(A) & \xrightarrow{\Gamma(t)} & \Gamma(A') \end{array}$$

A functorial model of the object language in $\mathcal{G}$ in the sense of Lawvere [60] is a functor $\iota : \mathbf{PSh}(\mathcal{T}) \to \mathcal{G}$ that preserves necessary structures, such as finite products and exponential objects for simply-typed λ-calculus. An analogue of the fundamental theorem of logical relations in this setting is that the functorial model ι is a section of the projection functor $\pi : \mathcal{G} \to \mathbf{PSh}(\mathcal{T})$, *i.e.* $\pi \circ \iota = \mathrm{Id}_{\mathbf{PSh}(\mathcal{T})}$ as depicted below. The action of π on morphism (h, t) is $\pi(h, t) = t : A \to A'$.

$$\begin{array}{ccccc} \mathcal{T} & \xrightarrow{\text{ɣ}} & \mathbf{PSh}(\mathcal{T}) & \xrightarrow{\iota} & \mathcal{G} \\ & & & & \downarrow{\scriptstyle \pi} \\ & & & \searrow^{\mathrm{Id}} & \\ & & & & \mathbf{PSh}(\mathcal{T}) \end{array}$$

Suppose the action of ι on the boolean type $\iota(\text{ɣ}(\mathsf{bool}_{\mathcal{T}})) = \mathsf{BOOL}_{\mathcal{G}}$ is defined as

$$\mathsf{BOOL}_{\mathcal{G}} = \left(\begin{array}{c} \{0,1\} \\ \downarrow {\scriptstyle f(0)=\text{ɣ}(\mathsf{false}_{\mathcal{T}})} \\ \downarrow {\scriptstyle f(1)=\text{ɣ}(\mathsf{true}_{\mathcal{T}})} \\ \Gamma(\text{ɣ}(\mathsf{bool}_{\mathcal{T}})) \end{array} \right).$$

Take any morphism $b : 1_{\mathcal{T}} \to \mathsf{bool}_{\mathcal{T}}$, which represents a closed term of boolean type in the object language. By the definition of ι, the morphism $\iota(\text{ɣ}(b)) : 1_{\mathcal{G}} \to \mathsf{BOOL}_{\mathcal{G}}$ in the gluing category $\mathcal{G}$ contains the following data $(h : \mathbf{1}_{\mathbf{Set}} \to \{0,1\}, t : \text{ɣ}(1_{\mathcal{T}}) \to \text{ɣ}(\mathsf{bool}_{\mathcal{T}}))$:

$$\begin{array}{ccc} \mathbf{1}_{\mathbf{Set}} & \xrightarrow{h} & \{0,1\} \\ \downarrow & & \downarrow {\scriptstyle f(0)=\text{ɣ}(\mathsf{false}_{\mathcal{T}})} \\ & & \downarrow {\scriptstyle f(1)=\text{ɣ}(\mathsf{true}_{\mathcal{T}})} \\ \Gamma(\text{ɣ}(1_{\mathcal{T}})) = \mathbf{1}_{\mathbf{Set}} & \xrightarrow{\Gamma(t)} & \Gamma(\text{ɣ}(\mathsf{bool}_{\mathcal{T}})) \end{array}$$

From $\pi \circ \iota = \mathrm{Id}$ it follows that $\pi(\iota(\text{ɣ}(b))) = \text{ɣ}(b)$, and hence $t = \text{ɣ}(b)$. Considering the function $h : \mathbf{1}_{\mathbf{Set}} \to \{0,1\}$, one obtains that h maps to either 0 or 1. Consequently, $\text{ɣ}(b)$ must be either $\text{ɣ}(\mathsf{true}_{\mathcal{T}})$ or $\text{ɣ}(\mathsf{false}_{\mathcal{T}})$. By the full faithfulness of the Yoneda embedding, it follows that b is either $\mathsf{true}_{\mathcal{T}}$ or $\mathsf{false}_{\mathcal{T}}$, thus establishing the canonicity theorem.

The heart of this proof is then to close the construction of ι under other type formers and term constructors of the object language, and check that ι is indeed a section of π in each case. For a tutorial construction of common type formers in the gluing category, we refer readers to Angiuli and Gratzer [14, §6.6].

Formally, this gluing proof already works if we glue directly over the syntactic category $\mathcal{T}$ and the global sections functor $\Gamma : \mathcal{T} \to \mathbf{Set}$. However, in anticipation of the type-theoretic reasoning in Section 2, working in the presheaf category $\mathbf{PSh}(\mathcal{T})$ allows us to leverage the internal language of presheaf categories to reason about the gluing construction in a type-theoretic manner.

1.2 Synthetic Tait Computability

The aforementioned gluing technique is a powerful and flexible tool for proving meta-theoretic properties of programming languages, but the construction of the gluing model can be quite involved (especially for those that prove normalization), with many tedious details to check. For example, depending on the exact presentation of the syntactic category $\mathcal{T}$, there may be many subtle but boring naturality conditions to verify.

Sterling's *synthetic Tait computability* (STC) [82] is a recent development that aims to simplify the construction of gluing models by internalizing the gluing construction in a suitable modal dependent type theory. This technique has been successfully applied to a variety of type theories, including modal [36] and cubical [84] type theories, ML-like module calculus [88], dependent type theory with controlled unfolding [39], higher-order effect handlers [95, 97], and dependent call-by-push-value [63].

The key idea of STC is to introduce a *synthetic phase distinction* between *syntax* and *semantics*. The idea of phase distinctions is originally due to the development of ML modules [47] in which a *static* phase isolates the static, compile-time constructs from dynamic, runtime constructs, where dynamic constructs can depend on static ones but not vice versa. The meta-theory of programming languages displays similar structures: the *semantics* depends on the *syntax*, but not vice versa, and a *syntactic* phase can be used to isolate the syntactic constructs from the semantic construction. This isolation of syntax is formally analogous to the projection functor $\pi : \mathcal{G} \to \mathbf{PSh}(\mathcal{T})$ in the gluing construction, which forgets the semantic information. The innovation of STC is to view the internal language of the gluing category $\mathcal{G}$ as a modal extensional dependent type theory with a *syntactic* phase, in which one can write down the definitions of the gluing model such as $\mathsf{BOOL}_{\mathcal{G}}$ and the fact that $\pi(\mathsf{BOOL}_{\mathcal{G}}) = \text{ɣ}(\mathsf{bool}_{\mathcal{T}})$ using type-theoretic constructs, without having to explicitly construct the objects and morphisms in the gluing category.

Synthetic Phase Distinction. Synthetically in type theory, a *syntactic* phase is an intuitionistic, type-theoretic proposition syn that, when assumed in the context, isolates the

syntax from the semantics. Categorically syn may be interpreted as a subterminal object $(\mathbf{0}_{\mathbf{Set}}, \text{ѣ}(\mathbf{1}_{\mathcal{T}}), !)$ in $\mathcal{G}$, which, when assumed in the context, erases the semantic component from **Set** and leaves only the syntactic component from $\mathcal{T}$. A proposition like syn immediately induces a pair of idempotent monadic modalities [78].

Open Modality. The open modality $\bigcirc A = \text{syn} \to A$ is the reader monad for the phase syn with monadic unit $\eta_\bigcirc : A \to \bigcirc A$ defined as $\eta_\bigcirc = \lambda a.\lambda z.a$. Intuitively, open modality introduces a syntactic assumption to the context, thereby isolating the syntax. Formally there is an equivalence of categories between $\mathcal{T}$ and the slice category $\mathcal{G}/\text{syn}$, where the projection functor π is equivalent to exponentiating by syn. The condition that $\pi(\text{BOOL}_{\mathcal{G}}) = \text{ѣ}(\text{bool}_{\mathcal{T}})$ can then be expressed in the internal language as an open equation $\bigcirc(\text{BOOL} = \text{bool})$.

Closed Modality. The closed modality $\bullet A = A \vee \text{syn}$ is the join between A and syn, which can be defined categorically as the pushout along the projection maps of $A \times \text{syn}$, or equivalently a quotient inductive type that equates all elements under the syntactic phase as follows:

$$\begin{array}{ccc} A \times \text{syn} & \xrightarrow{\pi_2} & \text{syn} \\ \downarrow{\scriptstyle \pi_1} & & \downarrow{\scriptstyle \star} \\ A & \xrightarrow[\eta_\bullet]{} & \bullet A \end{array}$$

$$\begin{array}{l} \textbf{data} \bullet (A : \mathcal{U}) : \mathcal{U} \textbf{ where} \\ \quad \eta_\bullet : A \to \bullet A \\ \quad \star : \text{syn} \to \bullet A \\ \quad \text{law} : (a : A)\ (z : \text{syn}) \to \\ \qquad \eta_\bullet\ a = \star\ z \end{array}$$

Intuitively, closed modality marks a type as purely semantic by identifying all its elements under the syntactic phase: $\bigcirc\bullet A$ is contractible, *i.e.* has exactly one element up to equality. Categorically in $\mathcal{G}$, the construction of open and closed modalities can be understood as an exponential object and a pushout respectively as follows.

$$\bigcirc\begin{pmatrix} S \\ \downarrow f \\ \Gamma(A) \end{pmatrix} = \begin{pmatrix} \Gamma(A) \\ \downarrow \text{id} \\ \Gamma(A) \end{pmatrix} \qquad \bullet\begin{pmatrix} S \\ \downarrow f \\ \Gamma(A) \end{pmatrix} = \begin{pmatrix} S \\ \downarrow ! \\ \mathbf{1}_{\mathbf{Set}} \end{pmatrix}$$

Other Dependent Type Formers. The Artin gluing $\mathcal{G} = \mathbf{Set} \downarrow \Gamma$ is an elementary topos [52], meaning that the internal language of $\mathcal{G}$ supports common dependent type formers, such as dependent products and sums, binary coproducts, extensional equality types, and a hierarchy of universes. We refer readers to Yang [95, §5.3.3] for an exposition of other type formers in $\mathcal{G}$.

1.3 Formalization of Synthetic Tait Computability

This work addresses the formalization of synthetic Tait computability in a proof assistant. The STC approach to metatheory is particularly attractive for mechanization: the internal language of the gluing category $\mathcal{G}$ is a dependent type theory that closely resembles the languages of many existing dependently-typed proof assistants. By extending a given proof assistant with the modal constructions required for STC, the definitions and proofs of STC can be expressed directly within that system. The on-paper proofs using STC typically deal with a large number of equalities coming from both the syntax of the object language and the meta-level modality framework. Nevertheless, such arguments are concise and elegant: for example, the canonicity proof for a core dependent type theory occupies less than a page. The following key factors contribute to this concision, and the subsequent discussion explains how they are addressed in the formalization of this work.

Equations in the Syntax. In an algebraic presentation, the equational theory of the object language is typically encoded as propositional equalities. Because the language of STC has extensional equality types, these propositional equalities can be internalized as judgmental equalities using *equality reflection*. Therefore reasoning about equalities is hidden in the background (as part of the typing derivation). In a proof assistant like Agda where equality reflection is not available, these propositional equalities must be explicitly transported in the proofs, leading to large terms with transports and equalities between those terms that obscure the main ideas of the proofs. While there are efforts to turn limited forms of propositional equalities into judgmental ones via rewrite rules in Agda [24], this approach is not as general as equality reflection. For this reason, the present mechanization is carried out in Istari [29], a Martin-Löf-style [67] extensional proof assistant with a computational semantics, which provides equality reflection natively.

Phase Distinction. A notable feature of synthetic phase distinction is that a term could have different types depending on the phase. For example, since $\bigcirc(\text{BOOL} = \text{bool})$, any term $b : \text{BOOL}$ is also $b : \text{bool}$ under the syntactic phase. This kind of implicit coercion is used heavily as a convenient device in on-paper STC proofs. The logic of Istari is a type-assignment system in the sense of Nuprl, in which a term may be assigned different types. This allows the implicit coercions to be used exactly as in on-paper STC proofs.

Extension Types. In STC, proof obligations like $\bigcirc(\text{BOOL} = \text{bool})$ are achieved by using *extension types* [77], $\{A \mid \text{syn} \hookrightarrow a_0\}$, the collection of elements of A that restrict to a_0 under the syntactic phase syn. Extension types are not natively supported in most proof assistants, but one option is to encode them as Σ-types $\Sigma_{a:A}\bigcirc(a = a_0)$. The Σ-type encoding, however, does not provide implicit coercion that is heavily used in on-paper STC proofs: if $a : \{A \mid \text{syn} \hookrightarrow a_0\}$ then $a : A$. On the other hand, extension types align well with subset types [7, 72], which Istari supports natively and gives the desired implicit coercion.

1.4 Contributions

This work presents a formalization of synthetic Tait computability in the Istari proof assistant, with the following contributions:

1. A library of synthetic phase distinction in Istari, including modalities, extension types, strict glue types, and other constructs necessary for STC;
2. Formalizations of two STC case studies in Istari:
   a. A canonicity gluing model for a dependent type theory with a base type of booleans supporting large elimination and dependent products [82, §4.4], corresponding to unary logical relations;
   b. A canonicity gluing model for the cost-aware logical framework [63, 71], a dependent call-by-push-value [61] language with a phase distinction for cost analysis, corresponding to unary Kripke logical relations [59].

Synopsis. The remainder of the paper is organized as follows. Section 2 provides a brief refresher on an example proof using synthetic Tait computability. Section 3 presents a tutorial on the Istari proof assistant and its underlying type theory. Section 4 introduces the library for synthetic phase distinction and illustrates the mechanization of STC through two case studies. Section 5 discusses related work on the formalization of gluing arguments, and Section 6 outlines possible directions for future research.

2 A Refresher on STC

This section provides a brief refresher on synthetic Tait computability by considering canonicity for a core dependent type theory with booleans, one of the case studies mechanized in Section 4. The discussion begins with an introduction of the technical devices required for STC.

2.1 Extension Types

Originally developed in the context of homotopy type theory [77] and cubical type theory [92], where the "phase" is dimension formula for cubes, extension types $\{A \mid \mathsf{syn} \hookrightarrow a_0\}$ classify the elements of a type A that are equal to a distinguished element a_0 under the influence of syn. In the context of STC, this construction provides a succinct and elegant formulation of the condition $\pi \circ \iota = \mathrm{Id}_{\mathcal{T}}$. Implicit coercions are employed to simplify notation: if $a : \{A \mid \mathsf{syn} \hookrightarrow a_0\}$ then $a : A$ and $\bigcirc(a = a_0)$, and conversely. The standard inference rules for extension types are presented in Fig. 1.

$$\textsc{Formation}\quad \frac{A : \mathcal{U} \qquad \mathsf{syn} \to (a_0 : A)}{\{A \mid \mathsf{syn} \hookrightarrow a_0\} : \mathcal{U}} \qquad \textsc{Introduction \& Elimination}\quad \frac{a : A \qquad \mathsf{syn} \to (a = a_0 : A)}{a : \{A \mid \mathsf{syn} \hookrightarrow a_0\}}$$

Figure 1. Inference rules for extension types

2.2 Strict Glue Types

The core idea of STC invites the existence of a strict glue type $(a : A) \ltimes B(a)$, where a syntactic component A is glued to a semantic component B just like a Σ type. The key difference is that it needs to be governed by the syntactic phase, so that open equations of the kind $\bigcirc((a : A) \ltimes B(a) = A)$ hold. In order for such equations to hold, the glue type needs to have the syntactic component A be *open-modal* and the semantic component B be *closed-modal* [78].

Definition 2.1. A type A is *open-modal* if its interpretation in the gluing category is a constant function, *i.e.* an object in the form of $(\Gamma(A), A, \mathrm{id})$. In the internal language, this means A is isomorphic to $\bigcirc A$: $A \cong \bigcirc A$.

Definition 2.2. A type B is *closed-modal* if its interpretation in the gluing category is trivial on the open part, *i.e.* an object in the form of $(B, \lrcorner(\mathbf{1}_{\mathcal{T}}), !)$. In the internal language, this means B is isomorphic to $\bullet B$: $B \cong \bullet B$. The present mechanization uses an equivalent definition [78] that is easier to work with: B is closed-modal if under syn, the type B is contractible, *i.e.* $\bigcirc B \cong \mathbf{1}$. Immediately $\bullet B$ for any type B is closed-modal; $\{A \mid \mathsf{syn} \hookrightarrow a_0\}$ is also closed-modal because syntactically it collapses to a singleton a_0.

In the gluing category, the glue type $(a : A) \ltimes B(a)$ is interpreted exactly as *gluing* the interpretations of A and B.

$$A = \begin{pmatrix} \Gamma(A) \\ \downarrow{\scriptstyle \mathrm{id}} \\ \Gamma(A) \end{pmatrix} \qquad B(a) = \begin{pmatrix} B(a) \\ \downarrow{\scriptstyle !} \\ \Gamma(\lrcorner(\mathbf{1}_{\mathcal{T}})) = \mathbf{1}_{\mathbf{Set}} \end{pmatrix}$$

$$(a : A) \ltimes B(a) = \begin{pmatrix} \coprod_{a \in \Gamma(A)} B(a) \\ \downarrow{\scriptstyle \pi_1} \\ \Gamma(A) \end{pmatrix}$$

Notationally we write $[\mathsf{syn} \hookrightarrow a \mid b]$ for an element of the glue type $(a : A) \ltimes B(a)$, equipped with projections $\pi_\circ$ and $\pi_\bullet$. The associated β- and η-equations hold as expected. The most significant equations are

$$\bigcirc((a : A) \ltimes B(a) = A) \quad \text{and} \quad \bigcirc([\mathsf{syn} \hookrightarrow a \mid b] = a).$$

These open equations are critical for the fundamental theorems of logical relations, where the open modality plays the role of projecting out the syntactic part. A complete set of inference rules for the strict glue type is presented in Fig. 2. The use of strict glue types in STC is standard [39, 83, 89, 95, 97] and can be justified by the realignment/strictification axiom in a Grothendieck topos [20, 37, 73, 82, 89]. Unlike Sterling and Harper [88] who use realignment directly to construct computability argument, this work axiomatizes strict glue types to emphasize the geometric intuition of the syntax/semantics distinction. Conceptually, this is closely related to the glue types employed in other forms of phase

distinction [43], where the corresponding open equations are justified by univalence.

$$\textsc{Formation \& Type-eq-syn}\quad \frac{A : \mathsf{syn} \to \mathcal{U} \qquad B : ((z : \mathsf{syn}) \to A\ z) \to \mathcal{U} \qquad (x : ((z : \mathsf{syn}) \to A\ z)) \to (B(x) \cong \mathbf{1})}{(x : A) \ltimes B(x) : \mathcal{U} \qquad (z : \mathsf{syn}) \to (x : A) \ltimes B(x) = A\ z : \mathcal{U}}$$

$$\textsc{Introduction}\quad \frac{a : (z : \mathsf{syn}) \to A\ z \qquad b : B(a)}{[\mathsf{syn} \hookrightarrow a \mid b] : (x : A) \ltimes B(x)}$$

$$\textsc{Elimination-open \& Elimination-closed}\quad \frac{g : (x : A) \ltimes B(x)}{\pi_\circ\ g : (z : \mathsf{syn}) \to A\ z \qquad \pi_\bullet\ g : B(\pi_\circ\ g)}$$

$$\textsc{Uniqueness}\quad \frac{g : (x : A) \ltimes B(x)}{g = [\mathsf{syn} \hookrightarrow \pi_\circ\ g \mid \pi_\bullet\ g] : (x : A) \ltimes B(x)}$$

$$\textsc{Computation-open \& Computation-closed}\quad \frac{a : (z : \mathsf{syn}) \to A\ z \qquad b : B(a)}{\pi_\circ\ [\mathsf{syn} \hookrightarrow a \mid b] = a : (z : \mathsf{syn}) \to A\ z \qquad \pi_\bullet\ [\mathsf{syn} \hookrightarrow a \mid b] = b : B(a)}$$

$$\textsc{Term-eq-syn}\quad \frac{g : (x : A) \ltimes B(x)}{(z : \mathsf{syn}) \to (\pi_\circ\ g)\ z = g : A\ z}$$

Figure 2. Inference rules for strict glue types

2.3 Syntax

The presentation of a type theory can be given succinctly as a signature in a logical framework following the *judgments as types* principle [46, 82]. For example, the signature of a dependent type theory with booleans, large elimination, and dependent products is given below in a semantic logical framework [45, 82, 93, 96], that is, in the internal language of locally Cartesian closed categories (LCCC). The adequacy of this presentation is ensured by Gratzer and Sterling [38] on defining dependent type theories in LCCCs. The syntactic category $\mathcal{T}$ is then the free LCCC generated by the constants of the signature, quotiented by the equalities between constants. By construction, all syntax is open-modal; in the mechanization this is enforced by taking the signature of the syntax under the $\bigcirc$ modality.

This second-order algebraic presentation of syntax in the sense of Martin Löf's logical framework [45, 56, 72] is particularly convenient and ergonomic for mechanization, in direct contrast to the inductive characterizations of syntax, which often require lengthy and intricate reasoning about bindings and substitutions as exemplified in many prior mechanization efforts for programming language meta-theory [2, 17], as well as the use of quotients, which are not available in many proof assistants, to represent judgmental equality.

$$\begin{aligned}
\mathsf{tp} &: \mathcal{U}\\
\mathsf{tm} &: \mathsf{tp} \to \mathcal{U}\\
\mathsf{bool} &: \mathsf{tp}\\
\mathsf{true} &: \mathsf{tm}(\mathsf{bool})\\
\mathsf{false} &: \mathsf{tm}(\mathsf{bool})\\
\mathsf{if} &: (C : \mathsf{tm}(\mathsf{bool}) \to \mathsf{tp}) \to (b : \mathsf{tm}(\mathsf{bool})) \to\\
&\quad \mathsf{tm}(C(\mathsf{true})) \to \mathsf{tm}(C(\mathsf{false})) \to \mathsf{tm}(C(b))\\
\mathsf{if}_{\beta_1} &: \mathsf{if}\ C\ \mathsf{true}\ t\ f = t\\
\mathsf{if}_{\beta_2} &: \mathsf{if}\ C\ \mathsf{false}\ t\ f = f\\
\mathsf{pi} &: (A : \mathsf{tp}) \to (\mathsf{tm}(A) \to \mathsf{tp}) \to \mathsf{tp}\\
\mathsf{lam} &: ((x : \mathsf{tm}(A)) \to \mathsf{tm}(B(x))) \to \mathsf{tm}(\mathsf{pi}\ A\ B)\\
\mathsf{app} &: \mathsf{tm}(\mathsf{pi}\ A\ B) \to (x : \mathsf{tm}(A)) \to \mathsf{tm}(B(x))\\
\mathsf{pi}_\beta &: \mathsf{app}\ (\mathsf{lam}\ f)\ a = f\ a\\
\mathsf{pi}_\eta &: \mathsf{lam}\ (\mathsf{app}\ e) = e
\end{aligned}$$

2.4 Canonicity Model

In specifying the functorial semantics $\iota : \mathbf{PSh}(\mathcal{T}) \to \mathcal{G}$, the main task is to define the images of the constants produced by the functor ι. More precisely, the goal is to define:

$$\begin{aligned}
\mathsf{TP} &: \{\mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tp}\}\\
\mathsf{TM} &: \{\mathsf{TP} \to \mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tm}\}\\
\mathsf{BOOL} &: \{\mathsf{TP} \mid \mathsf{syn} \hookrightarrow \mathsf{bool}\}\\
\mathsf{TRUE} &: \{\mathsf{TM}(\mathsf{BOOL}) \mid \mathsf{syn} \hookrightarrow \mathsf{true}\}\\
&\ldots
\end{aligned}$$

Notation. Throughout this paper, lowercase red terms denote syntactic components, whereas uppercase BLUE terms denote their semantic counterparts under the image of ι.

2.4.1 Semantics of Judgments.

The semantics of tp is given by the collection of all syntactic types in the language, each equipped with the corresponding collection of semantics of terms of that type, *cf*. the proof-relevant logical relations definition for universes [27, 80]. The glue type is used to assemble all data into a single structure.[1]

$$\begin{aligned}
\mathsf{TP} &: \{\mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tp}\}\\
\mathsf{TP} &= (A : \mathsf{tp}) \ltimes \{\mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tm}(A)\}
\end{aligned}$$

This definition satisfies the extension type condition $\bigcirc(\mathsf{TP} = \mathsf{tp})$ by the equation of glue type. The semantics of tm is then obtained by projecting the corresponding term collection for each type $A : \mathsf{TP}$.

$$\begin{aligned}
\mathsf{TM} &: \{\mathsf{TP} \to \mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tm}\}\\
\mathsf{TM}\ A &= \pi_\bullet A
\end{aligned}$$

[1] For simplicity, universe levels are omitted in the presentation, although they are fully accounted for in the formalization.

This definition also satisfies the extension type condition ○(TM = tm) by the use of extension type in TP.

2.4.2 Semantics of Booleans. In proving canonicity, the goal is to show that every closed term of boolean type is either true or false. To this end, the semantics of bool is defined by gluing the syntactic boolean terms with a semantic component that classifies the canonical boolean values.

$$\begin{aligned}&\mathsf{BOOL} : \{\mathsf{TP} \mid \mathsf{syn} \hookrightarrow \mathsf{bool}\}\\&\mathsf{BOOL} = [\mathsf{syn} \hookrightarrow \mathsf{bool} \mid\\&\quad (b : \mathsf{tm}(\mathsf{bool})) \ltimes \bullet(b = \mathsf{true} + b = \mathsf{false})]\end{aligned}$$

To type-check this definition, three conditions need to be verified:

1. First, ○(BOOL = bool) is required, which follows directly from the judgmental equations of terms of glue types.
2. Second, the semantic component of BOOL restricts to tm(bool) under syn by the glue type equation, as required by TP.
3. Third, the use of the closed modality in the predicate $\bullet(b = \mathsf{true} + b = \mathsf{false})$ guarantees that the semantic component of the glue type is closed-modal, as required by its formation rule.

The semantics of terms of type bool are injections, and the semantics of if is a case analysis on the disjunction in the semantic part of BOOL.

$$\begin{aligned}&\mathsf{TRUE} : \{\mathsf{TM}(\mathsf{BOOL}) \mid \mathsf{syn} \hookrightarrow \mathsf{true}\}\\&\mathsf{TRUE} = [\mathsf{syn} \hookrightarrow \mathsf{true} \mid \eta_\bullet(\mathsf{inl}(\checkmark))]\end{aligned}$$

$$\begin{aligned}&\mathsf{IF} : \{(C : \mathsf{TM}(\mathsf{BOOL}) \to \mathsf{TP}) \to (b : \mathsf{TM}(\mathsf{BOOL})) \to\\&\quad \mathsf{TM}(C(\mathsf{TRUE})) \to \mathsf{TM}(C(\mathsf{FALSE})) \to \mathsf{TM}(C(b)) \mid\\&\quad \mathsf{syn} \hookrightarrow \mathsf{if}\}\\&\mathsf{IF}\ C\ b\ t\ f = \mathsf{case}\ \pi_\bullet b\ \mathsf{of}\\&\quad \eta_\bullet(\mathsf{inl}(\_)) \Rightarrow t\\&\quad \eta_\bullet(\mathsf{inr}(\_)) \Rightarrow f\\&\quad \star\ z \Rightarrow \mathsf{if}\ C\ b\ t\ f\end{aligned}$$

Note that non-trivial equality proof obligations in the definition of IF arise in order to ensure that the three branches coincide, as the construction defines a map out of a quotient.

2.4.3 Semantics of Dependent Products. A recurring pattern emerges in the definition of these constants: the heart of the proof is expressed concisely in type-theoretic terms (sometimes called the *realizers*), followed by accompanying English explanations (the typing derivations) to justify the non-trivial well-typedness conditions. Because the internal language is extensional, reasoning about equalities occurs at the judgmental level rather than in the surface syntax. This observation motivates the mechanization of STC in an extensional proof assistant: the definitions should remain concise as-is, and the type-checker can ensure that all well-typedness conditions are formally satisfied. In other words, in extensional type theory, the proof consists of not only the term and type, but also its typing derivation. For instance, if the mechanization accepts the following definition of PI, then all necessary conditions can be guaranteed.

$$\begin{aligned}&\mathsf{PI} : \{(A : \mathsf{TP}) \to (\mathsf{TM}(A) \to \mathsf{TP}) \to \mathsf{TP} \mid \mathsf{syn} \hookrightarrow \mathsf{pi}\}\\&\mathsf{PI}\ A\ B = [\mathsf{syn} \hookrightarrow \mathsf{pi}\ A\ B \mid\\&\quad (e : \mathsf{tm}(\mathsf{pi}\ A\ B)) \ltimes \{(a : \mathsf{TM}(A)) \to \mathsf{TM}(B(a)) \mid\\&\qquad \mathsf{syn} \hookrightarrow \mathsf{app}\ e\}]\end{aligned}$$

$$\begin{aligned}&\mathsf{LAM} : \{((x : \mathsf{TM}(A)) \to \mathsf{TM}(B(x))) \to \mathsf{TM}(\mathsf{PI}\ A\ B) \mid\\&\quad \mathsf{syn} \hookrightarrow \mathsf{lam}\}\\&\mathsf{LAM}\ f = [\mathsf{syn} \hookrightarrow \mathsf{lam}\ f \mid f]\end{aligned}$$

$$\begin{aligned}&\mathsf{APP} : \{\mathsf{TM}(\mathsf{PI}\ A\ B) \to (x : \mathsf{TM}(A)) \to \mathsf{TM}(B(x)) \mid\\&\quad \mathsf{syn} \hookrightarrow \mathsf{app}\}\\&\mathsf{APP}\ e\ a = (\pi_\bullet e)\ a\end{aligned}$$

It is worth noting that, for the definition of LAM to be well-typed, the equation $\mathsf{pi}_\beta : \mathsf{app}\ (\mathsf{lam}\ f)\ a = f\ a$ from the syntax must be used. This equation does not appear explicitly in the term because, again, the internal language is extensional; by equality reflection, it is turned into a judgmental equality and can thus be used directly in type-checking. The β- and η-equations must also be verified, which can be done straightforwardly using the equations for glue types.

$$\begin{aligned}&\mathsf{PI}_\beta : \{\mathsf{APP}\ (\mathsf{LAM}\ f)\ a = f\ a \mid \mathsf{syn} \hookrightarrow \mathsf{pi}_\beta\}\\&\mathsf{PI}_\beta = \checkmark\end{aligned}$$

$$\begin{aligned}&\mathsf{PI}_\eta : \{\mathsf{LAM}\ (\mathsf{APP}\ e) = e \mid \mathsf{syn} \hookrightarrow \mathsf{pi}_\eta\}\\&\mathsf{PI}_\eta = \checkmark\end{aligned}$$

2.5 What Is So Synthetic About STC?

Compared to *analytical* mathematics, *synthetic* methods emphasize the use of high-level axiomatizations and abstractions to express ideas directly, rather than constructing them from more primitive notions. It is important to highlight the key axiomatizations that STC uses and how they impact the mechanization.

Axiom 1. *There is an intuitionistic proposition* syn *that classifies the syntactic phase.*

Axiom 1 is crucial in STC to isolate and simplify the categorical formulation of the fundamental theorem of logical relations: $\pi \circ \iota = \mathrm{Id}_{\mathcal{T}}$ becomes type-theoretic extension-type conditions such as BOOL : {TP | syn ↪ bool}. In the mechanization, this axiom can be realized by postulating such a proposition and its associated properties, as is done in other formalizations of synthetic phase distinctions [43, 71].

Axiom 2. *Syntactically, there exists syntax* tp, tm, bool, true, false, if, pi, lam, app, *etc.*

Axiom 2 axiomatizes the syntax of the object type theory using higher-order syntax. Most notably, the syntax is *not* inductively defined; in fact, the entire synthetic development of STC avoids any inductive definitions. Therefore, internally, the STC proof cannot perform case analysis on the syntax.

This axiom is essential for the concise presentation of syntax in STC, as the use of higher-order abstract syntax simplifies bindings and substitutions significantly compared to first-order representations.

It is then a natural question to ask: where is the induction proof in STC? The answer is that the induction proof is *analytical* with respect to the *synthetic* development of STC, *i.e.* in the categorical interpretation as shown in Section 1.1. The category that represents the syntax, $\mathcal{T}$, is inductively constructed as the free LCCC generated by the signature of Axiom 2, and the functorial semantics $\iota : \mathbf{PSh}(\mathcal{T}) \to \mathcal{G}$ is defined by induction on this inductively defined category. In other words, the synthetic proofs TP, TM, BOOL, *etc.* are pieces of lemmas that can be used to construct the inductive, analytical proof.

The advantage of this approach is the level of abstraction: the analytical justification of Axiom 1 and Axiom 2 in terms of category theory only needs to be carried out once for all object languages, and only the type-theoretic, synthetic proofs need to be developed for each case study. The mechanization of this work, therefore, focuses exclusively on the synthetic part of STC, namely the core proofs of the logical relations argument.

3 The Istari Proof Assistant

Istari [29] is a recently developed, tactic-oriented proof assistant based on Martin-Löf extensional type theory [67], following the tradition of Lcf [69] and Nuprl [26]. It represents a significant extension of traditional Nuprl-style proof assistants, providing support for guarded recursion, impredicativity, and other features. Istari provides a Rocq-style user experience to build proof scripts using tactics. This section provides a brief overview of Istari and its type theory, focusing on aspects most relevant to the present mechanization.

In Istari terms and computations exist prior to their typing. Types in Istari represent partial equivalence relations (PERs) on terms. The judgment $\Gamma \vdash M = N : A$ asserts that the terms M and N are equal as elements of type A. The derived typing judgment $\Gamma \vdash M : A$ indicates that M is a reflexive instance of the equality $\Gamma \vdash M = M : A$. The logic of Istari is a *type-assignment system* [29], allowing users to assign types to terms through automatic and manual typing proofs. Some terms may be assigned multiple types, and others may not be assigned any type at all. Similarly, two terms may be equal at one type but distinct at another.

Istari is an *extensional* type theory, in which *judgmental* and *propositional* equality coincide, through the principle of *equality reflection* [66, 67]. Therefore, notationally we write $M = N : A$ for both judgmental equality and propositional equality. Consequently, a typing judgment $M : A$, being a reflexive instance of equality, also constitutes a proposition within the logic. Typing proofs may involve any mathematical facts, including previously established equations; as a result, type-checking is undecidable in general. In practice, the included type-checker uses known and previously proved typing relations to discharge most of the type-checking goals.

Computationally equal [29] terms may be converted to one another in a type-free manner. For instance, to establish $(\lambda\, x.M)\ N : A$, it suffices to show for the β-reduct $M[N/x] : A$. Computational equality is closely related to direct computation in Nuprl [7, 26, 49]. This principle aligns with the computational interpretation of terms and reflects the philosophy of meaning explanations from Martin-Löf type theory [67].

Equalities in Istari are dictated by the type, similar to those of observational type theory [10, 75]. Conceptually, Istari terms are programs corresponding to the computational content of proofs. Types classify the computational behavior of terms, and computationally equal terms are equal at that type. Concretely, Istari supports:

1. *Function extensionality*: two functions F and G are equal ($F = G : A \to B$) if and only if $M = N : A$ implies $F\,M = G\,N : B$.
2. *Uniqueness of identity proofs*: any proof of equality $M = N : A$ is equal to the trivial empty tuple $()$.

Equality at universes $\mathsf{U}\, i$ is *intensional*: for example, product types are equal $A \times B = A' \times B' : \mathsf{U}\, i$ if and only if their components are equal $A = A' : \mathsf{U}\, i$ and $B = B' : \mathsf{U}\, i$.

Istari supports a wide range of types beyond what is typically available in proof assistants based on dependent type theory. Most relevant to this work, Istari supports:

1. *Subtyping.* A type A is a *subtype* of B, written $A <: B$, if equality at type A implies equality at type B. Consequently, if $M : A$, then $M : B$. Subtyping is reflexive and transitive.
2. *Cumulative universes.* Istari supports a hierarchy of cumulative universes $\mathsf{U}\,0, \mathsf{U}\,1, \mathsf{U}\,2, \ldots$, where $\mathsf{U}\, i <: \mathsf{U}\,(1 + i)$ for every i.
3. *Intersection types.* A term M inhabits the intersection type $\mathsf{intersect}\ (x : A)\ .\ B$ if and only if, for every term N such that $N : A$, $M : B(N)$ holds. In other words, the same term M inhabits every member of the type family B. Intersection types are particularly convenient for handling universe levels. For instance, $A : \mathsf{intersect}\ (i : \mathsf{level})\ .\ \mathsf{U}\, i$ states that A is a type at any universe i.
4. *Subset types.* Istari provides support for working with proof irrelevance with a range of tools. A term M inhabits the subset type $\{x : A \mid P\}$ if $M : A$ and $P(M)$ is inhabited. The proof is *irrelevant* for the equality between elements of the subset type; in particular, $\{x : A \mid P\} <: A$.
5. *Guarded types.* Another tool for proof irrelevance is the guarded function type $A \xrightarrow{g} B$. To establish $M : A \xrightarrow{g} B$, it is sufficient to derive $M : B$, with $x : A$ as a "proof-irrelevant" assumption. Dually, to use $M : A \xrightarrow{g} B$, it suffices to use it as $M : B$ and to establish A using tactics. Guarded types

serve to encode *presuppositions* such as those in the strict glue types.

The meta-theory of Istari, including soundness and consistency, has been mechanized in Rocq [28]. The remainder of this section presents a few simple examples, both to elaborate on the preceding discussion and to illustrate additional characteristics of the system.

3.1 Istari by Example

Define **vec** $A\ n$ to be the type of n-element vectors of type A. For the purpose of demonstration, **vec** $A\ n$ is specified as the subset of lists whose computed length is n.

$$\textbf{vec}\ A\ n \triangleq \{x : \textbf{list}\ A \mid \textbf{length}(x) = n : \textbf{nat}\}$$

Following the Lcf tradition, the Istari proof system consists of a trusted kernel and an interface. The kernel maintains the current proof state, and the interface allows users to manipulate proof objects by invoking tactics and effectful functions on the kernel. For instance, the **vec** type can be defined in Istari as follows:

```
define /vec A n/
/ {x : list A | length(x) = n : nat}                    /
/ intersect i . forall (A : U i) (n : nat) . U i        /;
```

The command "define" introduces a new definition, specified by its name, parameters, raw term, and type. The type **vec** $A\ n$ is made universe polymorphic by introducing a universe level i through intersection; consequently, i does not appear among the parameters.

In the proof mode the goal is to prove the declared typing. This is discharged with three tactics.

```
inference. unfold /vec/. typecheck.
```

The inference tactic performs unification and fills in implicit arguments, such as the type of the variable i. The defined constant is then unfolded, and finally the type-checker resolves all remaining goals.

As a first example, consider the operation of appending two vectors v_1 and v_2.

```
define /append {A} v1 v2/
/ List.append v1 v2                                     /
/ intersect i m n . forall (A : U i)
    (v1 : vec A m) (v2 : vec A n) . vec A (m + n)       /;
```

The function takes in two vectors v_1 and v_2 of lengths m and n, respectively. Its definition is simply **List.append**, exactly the computation required to append two vectors, without any explicit reasoning about lengths. The length constraints are instead handled in the typing proof. The typing proof begins by destructing v_1 and v_2, which produces four assumptions and a new proof goal:

```
v1 v2 : list A
H1 (hidden) : List.length v1 = m : nat
H2 (hidden) : List.length v2 = n : nat
⊢ List.append v1 v2
  : {x : list A | List.length (x) = m + n : nat}
```

In addition to v_1, v_2 : **list** A, two *hidden* assumptions about their lengths are generated. A hidden assumption can only be used in proof-irrelevant proof goals, such as typing proofs, as is the case here.

The next tactic splitOf establishes inhabitation of a subset type by requiring two proofs: first, that the result is a **list** A; and second, that its length equals the sum of the lengths of the arguments. The first obligation is automatically discharged by the auto tactic. The second requires a lemma from the **List** library. A complete chain of tactics for this interaction is shown below.

```
inference. unfold /append, vec at all/.
introOf /i m n A v1 v2/.
destruct /v1/ /v1 H1/. destruct /v2/ /v2 H2/.
unhide.
splitOf » auto.
subst /m n/.
apply /List.length_append/.
```

As illustrated by this example, Istari provides a streamlined process with a clear separation between the computational content (**List.append**) and the corresponding correctness argument via subset types.

3.2 Equality Reasoning in Istari

Istari offers a variety of tools to alleviate the difficulties of equality reasoning, most notably through equality reflection. As a result, terms often remain close to their intended computational intuition. This section illustrates several of these tools in Istari, with emphasis on techniques used in the mechanization of STC.

3.2.1 Extensionality. Heterogeneous equality can be approached directly through extensionality from either side. Consider the associativity of **append**: for vectors v_1, v_2, v_3,

$$\textbf{append}\ (\textbf{append}\ v_1\ v_2)\ v_3\ =\ \textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3).$$

In an intensional type theory, such a statement is not immediately well-typed because the two sides have heterogeneous types and thus one must transport at least one side. By contrast, Istari permits such heterogeneous equalities as-is.

```
lemma assoc
/ forall i (A : U i) n1 n2 n3
    (v1 : vec A n1) (v2 : vec A n2) (v3 : vec A n3) .
    append (append v1 v2) v3 =
      append v1 (append v2 v3) : _                      /;
```

To complete this proof, it suffices to appeal to associativity of **List.append** and use tactics to reason about underlying equalities induced by subset types.

3.2.2 Origami with fold **and** unfold**.** The final example illustrates a technique extensively used in this work to manage coercions via the identity function. This technique, which we call *Origami*, resembles the use of transport in intensional type theory, but differs fundamentally in that it relies on the

computational content of the identity function. Consider the definition of a reverse function on vectors:

$$\textbf{reverse} : \textbf{vec}\ A\ n \to \textbf{vec}\ A\ n \triangleq \textbf{List.reverse}.$$

Now suppose the goal is to establish

$$\begin{aligned}&\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3))\\ =\ &\textbf{reverse}\ (\textbf{append}\ (\textbf{append}\ v_1\ v_2)\ v_3).\end{aligned}$$

One possible attempt is to directly apply the **assoc** lemma via the rewrite tactic, which replaces one side of an equality with the other. However, invoking the tactic directly confuses the type-checker and generates impossible proof obligations, such as $n_1 = n_1 + n_2$. One remedy is to first *fold* a coercion onto one side of the equation along the identity function, and later *unfold* it. In Istari, the coercion along $H : (A = B : \mathbf{U}\ i)$ can be defined as the following identity function:

```
define /coe H/
/ fn a . a                                                    /
/ intersect i (A B : U i) . A = B : U i → A → B               /;
```

Starting with the proof obligation

$$\begin{aligned}\vdash\ &\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3)) =\\ &\textbf{reverse}\ (\textbf{append}\ (\textbf{append}\ v_1\ v_2)\ v_3) : \_\end{aligned}$$

the first step is to fold the coercion $\textbf{coe}_H$ around the right-hand side using the tactic fold /**coe** H/, producing a homogeneous equality:

$$\begin{aligned}&H : \textbf{vec}\ A\ ((n_1 + n_2) + n_3) = \textbf{vec}\ A\ (n_1 + (n_2 + n_3)) : \mathsf{U}\ i\\ \vdash\ &\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3)) =\\ &\textbf{reverse}\ (\textbf{coe}\ H\ (\textbf{append}\ (\textbf{append}\ v_1\ v_2)\ v_3)) : \_\end{aligned}$$

After this, rewrite tactic can be applied without confusion, which results in:

$$\begin{aligned}&H : \textbf{vec}\ A\ ((n_1 + n_2) + n_3) = \textbf{vec}\ A\ (n_1 + (n_2 + n_3)) : \mathsf{U}\ i\\ \vdash\ &\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3)) =\\ &\textbf{reverse}\ (\textbf{coe}\ H\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3))) : \_\end{aligned}$$

The key distinction of coercion in Istari compared to its intensional analogue is the ability to unfold **coe**, which evaluates the identity function. By contrast, coercions in intensional type theory $\textbf{coe} : (A = B : \mathbf{U}\ i) \to A \to B$ do not become the identity function unless the equality in question is reflexivity, which almost always is not the case in a large proof. Unfolding reduces the goal to a reflexive instance, which can be discharged with the reflexivity tactic:

$$\begin{aligned}\vdash\ &\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3)) =\\ &\textbf{reverse}\ (\textbf{append}\ v_1\ (\textbf{append}\ v_2\ v_3)) : \_\end{aligned}$$

This technique is extensively applied in this work to facilitate type-checking during rewriting.

4 Mechanization

The mechanization of synthetic Tait computability in Istari begins with a library for synthetic phase distinction. This library allows on-paper definitions to be transcribed directly into the formalization, with type-checking generating exactly the expected proof obligations. These obligations are then discharged using tactics in Istari.

4.1 Library for Synthetic Phase Distinctions

Because Istari does not have a phase distinction built in, it is extended with a library of definitions and lemmas for phase, modalities, extension types, and strict glue types. This extension axiomatizes the relevant constants and equations as explained in Section 2.5, following the style of logical frameworks [45, 46]. In the mechanization, axioms are variables collected in a context. A subset of representative definitions is summarized in this subsection.

4.1.1 Phase. A phase is represented as a type syn with at most one element up to equality in Istari.

$$\begin{aligned}&\textbf{syn} : \mathsf{U}\ i\\ &\textbf{syn\_prop} : \text{forall}\ (z\ w : \textbf{syn})\ .\ z = w : \textbf{syn}\end{aligned}$$

4.1.2 Closed Modality. The closed modality ● includes two constructors, $\eta_\bullet$ and $\star$, along with an equation law that identifies them at the syntactic phase.

$$\begin{aligned}&\textbf{closed} : \mathsf{U}\ i \to \mathsf{U}\ i\\ &\textbf{eta} : A \to \textbf{closed}\ A\\ &\textbf{star} : \textbf{syn} \to \textbf{closed}\ A\\ &\textbf{law} : \text{forall}\ (a : A)\ (z : \textbf{syn})\ .\ \textbf{eta}\ a = \textbf{star}\ z : \textbf{closed}\ A\end{aligned}$$

As a pushout/quotient, the closed modality has an eliminator of the following type:

$$\begin{aligned}&\textbf{closed\_elim} : \text{forall}\ (C : \textbf{closed}\ A \to \mathsf{U}\ i)\\ &\quad (a : \textbf{closed}\ A)\\ &\quad (c_\eta : \text{forall}\ (a : A)\ .\ C(\textbf{eta}\ a))\\ &\quad (c_\star : \text{forall}\ (z : \textbf{syn})\ .\ C(\textbf{star}\ z))\ .\\ &\quad (eq : \text{forall}\ (a : A)\ (z : \textbf{syn})\ .\ c_\eta\ a = c_\star\ z : \_)\ \xrightarrow{g}\ C\ a\end{aligned}$$

Most notably, it is not a priori true that this eliminator is well-defined, because $c_\eta\ a$ and $c_\star\ z$ have disparate types. Only via the equation **law** can they be identified. In intensional proof assistants such as Agda, this term cannot be expressed directly; one must transport one side of the equation along **law**. In Istari, the eliminator can be written literally as above because, at that stage, it is merely a raw term. The type-checker generates the proof obligation requesting identification of the types, which is discharged by citing the equation **law** using tactics. This pattern is common when postulating quotient types in proof assistants. Cubical type theories [12, 25] can also handle this situation in a computationally well-behaved manner using heterogeneous path types, but equality reflection offers an even simpler solution in situations like this.

Another notable aspect of this definition is the use of the guarded function arrow $\xrightarrow{g}$ in the equality condition *eq*. As introduced in Section 3, guarded functions allow one to avoid explicit equality reasoning in the term; such facts can instead be established in the typing derivation using tactics. This

approach lets terms that use the eliminator be written more naturally, as in the definition of IF in Section 2. Using these definitions, lemmas such as the fact that the ● modality is closed-modal can then be proved.

4.1.3 Strict Glue Type.

Strict glue types as shown in Fig. 2 are similarly encoded in Istari. The following is a representative selection of definitions.

```
glue_type : forall (A : syn → U i) .
  forall (B : (forall (z : syn) . A z) → U i) .
  (forall a . closed_model (B a)) -g-> U i

glue : forall (a : forall (z : syn) . A z) . (B a) →
  glue_type A B

pi_open : glue_type A B → (forall (z : syn) . A z)
pi_closed : forall (g : glue_type A B) . B (pi_open g)

type_eq_syn : forall (z : syn) .
  glue_type A B = A z : U i
term_eq_syn : forall (z : syn) (g : glue_type A B) .
  (pi_open g) z = g : A z
```

4.1.4 Extension Type.

Extension types $\{A \mid \text{syn} \hookrightarrow a_0\}$ are implemented as subset types as follows:

```
define /ext A a0/
/ {x : A | forall (z : syn) . x = a0 z : A}          /
/ forall (A : U i) . (syn → A) → U i                 /;
```

The most useful lemma about extension types is that $\{A \mid \text{syn} \hookrightarrow a_0\}$ is a subtype of A, which is extensively used in our development to prove that if a : **ext** $A\ a_0$ then $a : A$. This lemma follows directly from subset types.

```
lemma ext_subtype
/ forall (A : U i) (a0 : syn → A) . ext A a0 <: A    /;
```

4.2 Definitions of STC in Istari

The remainder of the development consists largely of a straightforward transcription of the on-paper definitions from Section 2 into Istari, using tactics to prove each term has the correct type. This process is mechanical and raises few surprises, highlighting the effectiveness of Istari for mechanizing synthetic Tait computability. As expected, all proof obligations required by STC arise naturally and automatically as type-checking goals in Istari, and are then discharged using tactics in a mostly straightforward manner.

This process can be illustrated using the definition of LAM from Section 2. As a reminder, the definition is:

```
LAM : {((x : TM(A)) → TM(B(x))) → TM(PI A B) |
  syn ↪ lam}
LAM f = [syn ↪ lam f | f]
```

That is, the semantics of a lambda function whose type is TM(PI A B) is a syntactic lambda lam f and a semantic function space itself $f : ((x : \text{TM}(A)) \to \text{TM}(B(x)))$. This definition is expressed in Istari as follows:

```
define /LAM A B/
/ fn f . glue (fn z . lam f) f                       /
/ forall (A : TP) (B : TM(A) → TP) .
    ext ((forall (x : TM(A)). TM(B(x))) →
      TM(PI A B)) (fn z . lam)                       /;
```

The type-checker in Istari generates the following proof obligations for this definition:

```
z : syn
⊢ (fn f . glue (fn z . lam f) f) = lam
  : ((forall (a : tm(A)) . tm(B(a))) → tm(pi A B))
```

This condition, under the syntactic phase LAM is equal to lam, emerges as a type-checking goal when the type-checker reaches the rules for the extension type. This corresponds exactly to one of the critical conditions expected from the definition of STC and the gluing argument in general. The goal is advanced using the function extensionality tactic, yielding:

```
z : syn
f : (forall (a : tm(A)) . tm(B(a)))
⊢ glue (fn z . lam f) f = lam f : tm(pi A B)
```

The proof is concluded by citing the corresponding equation from the definition of the strict glue type. A further proof obligation arises from the extension type in the definition of PI: it must hold that app (lam f) $= f$, since every term of type pi A B is characterized by its application app. This appears as

```
z : syn
f : (forall (a : tm(A)) . tm(B(a)))
⊢ app (lam f) = f : (forall (a : tm(A)) . tm(B(a)))
```

This goal is discharged by citing the corresponding equation pi_β from syntax.

For each constant in the STC definition, the same process is followed: the on-paper term is transcribed verbatim into Istari and then type-checked using tactics. The default type-checker of Istari automatically discharges most proof obligations, and the remaining ones typically capture the essential content of the STC proof, the parts that would otherwise be justified informally in an on-paper development. These are then handled manually by citing the relevant equations and lemmas. As one might expect, extensional type theories such as Istari enable all definitions to be expressed essentially verbatim without modification, thereby minimizing the gap between the on-paper and formalized proofs.

4.3 Proof Engineering in Istari

This subsection discusses techniques used to streamline equality reasoning in Istari with concrete examples from the mechanization of STC.

4.3.1 Hinting the Type-checker with Origami.

A notable phenomenon of synthetic phase distinction is that a

term may inhabit different types depending on the phase. For instance, a term of type TP is also of type tp under the syntactic phase, by virtue of the identification ○(TP = tp). Consider, for example, the definition of PI:

PI : {(A : TP) → (TM(A) → TP) → TP | syn ↪ pi}
PI A B = [syn ↪ pi A B | ⋯]

To type-check the syntactic part of this definition, Istari will generate the following proof obligation:

z : **syn**
A : **TP**
B : **TM**(A) → **TP**
⊢ **pi** A B : **tp**

Because **pi** : forall (A : **tp**) . (**tm**(A) → **tp**) → **tp** and A : **TP**, the type-checker will further generate the sub-goal:

z : **syn** ⊢ **TP** = **tp** : U i

which can then be proved using the extension type property on **TP**. However, this proof obligation shows up often enough that it became tedious to discharge it manually. With the Origami technique described in Section 3.2.2 we can streamline it by inserting "type casts" that point the type-checker to a specific type equality, without affecting the extent of the term itself. To this end, define

cast z ≜ **coe** (H z), where H : ○(TP = tp).

Definition of **PI** can then be modified to impose **cast** on A:

define /**PI′** A B/
/ fn A B . **glue** (fn z . **pi** (**cast** z A) ⋯) ⋯ /
/ forall (A : **TP**) (B : **TM**(A) → **TP**).
ext ((forall (x : **TM**(A)) . **TM**(B(x))) →
TM(**PI** A B)) (fn z . **pi**) /;

The proof obligation **TP** = **tp** no longer arises during type-checking, as the type of **cast** z A is already **tp**. As in Section 3.2.2, this "type cast" is computationally just an identity function. It can subsequently be eliminated by unfolding relevant definitions, yielding

(fn A B . **glue** (fn z . **pi** A ⋯) ⋯)
: forall (A : **TP**) (B : **TM**(A) → **TP**) . ⋯ .

This recovers exactly the original on-paper definition of PI. Notably, the situation differs from coercions in intensional proof assistants, where computation of a coercion is possible only when the underlying equality is reflexivity, a condition rarely satisfied within a large proof. The definition of **cast** plays an additional role in the development: proof obligations of the following form often arise after converting **TM** to **tm**:

z : **syn**
A : **TP**
⊢ **tm** A : U i

This induces the familiar sub-goal **TP** = **tp** as before. In order to let the type-checker discharge this goal, the opposite strategy from Section 3.2.2 can be used: rather than eliminating a coercion from a definition, one introduces it within a proof by imposing **cast** on A via the fold tactic.

fold /**cast** z A/.

The fold tactic changes the goal to **tm** (**cast** z A) : U i, which the type-checker discharges automatically. This proof engineering technique becomes particularly significant in the presence of extension types. For instance, **TM**(A) is frequently used to denote the type of terms of type A. The type-checker has difficulty with this, because the type of **TM** is an extension type, yet it is applied as if it were a function with argument A.

TM : **ext** (**TP** → U i) (fn z . **tm**)

The type-checker tries to unify the extension type and a function type, creating an impossible goal:

⊢ (**TP** → U i) = **ext** (**TP** → U i) (fn z . **tm**) : U (1 + i)

As before, Origami can guide the type-checker to coerce between extension types and their original types. **coe** H in Section 3.2.2 can be similarly extended to enable passage from subtypes to super-types when H proves a subtyping. Citing **ext_subtype** from Section 4.1.4, we define

out ≜ **coe ext_subtype**.

The problematic **TM**(A) is instead written as (**out TM**) A, which type-checks without issue. **out** can be eliminated as before, recovering the on-paper definitions. This behavior contrasts with formulations of extension types in which **in** and **out** are primitive introduction and elimination forms [77, 98] that cannot be "computed away" on their own.[2]

4.3.2 Computational Equality. The semantics of large elimination for booleans, IF, together with its associated equation IF_{β_1}, provide a representative example of the usefulness of computational equality in Istari, as discussed in Section 3. Concretely, the semantics of BOOL is given by a binary sum, TRUE corresponds to a left injection, and IF is a case analysis. In verifying the equation

$$\mathsf{IF}_{\beta_1} : (\mathsf{IF}\ C\ \mathsf{TRUE}\ t\ f = t : \mathsf{TM}(C(\mathsf{TRUE})))$$

the following proof obligation arises:

C : **TM**(**BOOL**) → **TP**
t : **TM**(C(**TRUE**))
f : **TM**(C(**FALSE**))
⊢ (case (inl ()) of | inl _ . t | inr _ . f) : **TM**(C(**TRUE**))

To type-check this goal directly, the type-checker needs to reason about the impossibility of the second branch. With computational equality, it is possible to first run a type-free computation on the term and then type-check the resulting term. Using the reduce tactic of Istari, the term in question is simplified according to the type-free operational semantics:

(case (inl ()) of | inl _ . t | inr _ . f) ↦ t.

The new goal t : **TM**(C(**TRUE**)) is immediate.

[2] An analogy for this distinction is the difference between equal-recursive and iso-recursive types.

4.4 Case Studies

The effectiveness of this formalization is demonstrated by two case studies of STC applications.

Core Dependent Type Theory. The first case study is the canonicity gluing model of a dependent type theory with dependent product types and booleans with large elimination, exactly as presented in Section 2. Each constant in the proof of STC is transcribed exactly as on-paper from Sterling [82, 83] into Istari as terms and types, followed by type-checking using tactics. The terms themselves remain as concise as in the on-paper development. We report the rough number of lines of tactics needed for type-checking each constant below.

Lines of tactics	Definitions
0–100	TM, TP, TRUE, FALSE
100–200	PI, LAM, PI_β, BOOL
400–500	APP, PI_η
1200–1500	IF, IF_{β_1}, IF_{β_2}

In some cases, the tactic scripts are longer than expected because the modality framework is axiomatized inside Istari rather than being built in as primitives, so the type-checker does not have as good support for reasoning about, *e.g.*, strict glue types as it would have if they were built in. For example, unlike built-in dependent sums, strict glue types do not enjoy type-free computation, so manual tactics are needed to go through certain steps in PI_β. Similarly, having to work with axiomatized large elimination for the $\bullet$ modality in IF leads to longer proof scripts. Nevertheless, the overall length of the tactic scripts remains quite manageable, and this only impacts tactic scripts rather than the terms themselves.

Cost-Aware Logical Framework. Cost-aware logical framework (**calf**) [44, 71] is a dependent call-by-push-value language designed for synthetic cost analysis of algorithms [42, 57, 62, 100]. **calf** incorporates a phase distinction beh between cost and behavior, analogous to syn, to isolate the behavioral aspects of a program from its cost. Its call-by-push-value [61] structure includes a free-forgetful adjunction $\mathsf{F} \dashv \mathsf{U}$, with an underlying writer monad used for cost tracking, and a cost-charging computational effect $\mathsf{step}^c(e)$ that records c units of cost when executing the computation e.

The canonicity property of **calf** is formally established in Li and Harper [63], where a gluing model is developed using STC. **calf** is justified by a two-world Kripke semantics, *i.e.* presheaves over the poset $\mathbb{2} = \{\mathsf{beh} \to \top\}$. To be precise, the gluing category $\mathcal{G}$ is taken to be $\mathbf{PSh}(\mathbb{2}) \downarrow N_\rho$ where N_ρ is the global sections functor induced by the phase-separated interpretation ρ of **calf** into $\mathbf{PSh}(\mathbb{2})$ as follows:

$$\rho : \mathbb{2} \to \mathbf{PSh}(\mathcal{T}) \qquad N_\rho : \mathbf{PSh}(\mathcal{T}) \to \mathbf{PSh}(\mathbb{2})$$
$$\rho(\mathsf{beh}) = \mathbb{y}(\mathsf{beh}) \qquad N_\rho(X) = \mathsf{Hom}(\rho -, X)$$
$$\rho(\top) = \mathbb{y}(\mathbf{1}_{\mathcal{T}})$$

Synthetically this two-world Kripke structure is captured by axiomatizing the beh phase in the Istari mechanization, in a similar manner to the syn phase.

The STC proof then captures the Eilenberg-Moore [32] categorical structure for call-by-push-value [61] where the free functor F constructs free monad algebras and the forgetful functor U forgets the algebra structure. Every computation type in call-by-push-value is interpreted as a monad algebra CostAlg that satisfies the monad laws as well as an equation $\mathsf{step}_{\mathsf{beh}}$ that erases cost at the behavioral phase. For any computation type X, the syntax of **calf** forms exactly such an algebra $\mathsf{SynAlg}(X)$ under the syntactic phase syn.

record CostAlg **where**
  Car : $\mathcal{U}$
  step : $\mathbb{C} \to \mathsf{Car} \to \mathsf{Car}$
  step_0 : $\mathsf{step}\ 0\ x = x$
  step_+ : $\mathsf{step}\ c_1\ (\mathsf{step}\ c_2\ x) = \mathsf{step}\ (c_1 + c_2)\ x$
  $\mathsf{step}_{\mathsf{beh}}$: $\bigcirc_{\mathsf{beh}}(\mathsf{step}\ c\ x = x)$

The STC proof for computation types $\mathsf{tp}^\ominus$ is similar to that for types tp in Section 2.4.1, with the addition of verifying that every computation type in the gluing model forms a monad algebra, and that they collapse to the syntactic algebra SynAlg under the syntactic phase.

$\mathsf{TP}^\ominus$: $\{\mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{tp}^\ominus\}$
$\mathsf{TP}^\ominus = (X : \mathsf{tp}^\ominus) \ltimes \{\mathsf{CostAlg} \mid \mathsf{syn} \hookrightarrow \mathsf{SynAlg}(X)\}$

The desired canonicity property for **calf** is that every closed term of boolean free computation is equal to charging certain cost and returning either true or false.

Theorem 4.1 (Canonicity). *For every term* $e : \mathsf{tm}^\ominus(\mathsf{F}(\mathsf{bool}))$, *there exists* $c : \mathbb{C}$ *such that* $e = \mathsf{step}^c(\mathsf{ret}(\mathsf{true}))$ *or* $e = \mathsf{step}^c(\mathsf{ret}(\mathsf{false}))$.

This property is captured in the carrier type of F, which forms the free monad algebra.

$\mathsf{F} : \{\mathsf{TP}^+ \to \mathsf{TP}^\ominus \mid \mathsf{syn} \hookrightarrow \mathsf{F}\}$
$\mathsf{F}(A) = [\mathsf{syn} \hookrightarrow \mathsf{F}(A) \mid \mathsf{FreeAlg}(A)]$

$\mathsf{FreeAlg}(A).\mathsf{Car} : \{\mathcal{U} \mid \mathsf{syn} \hookrightarrow \mathsf{SynAlg}(\mathsf{F}(A)).\mathsf{Car}\}$
$\mathsf{FreeAlg}(A).\mathsf{Car} = (e : \mathsf{tm}^\ominus(\mathsf{F}(A))) \ltimes \bullet P$
  where
    $P = \Sigma_{a:\mathsf{TM}^+(A)} \Sigma_{bc:\bullet_{\mathsf{beh}}\mathbb{C}}$ case bc of
      $\star\ (b : \mathsf{beh}) \Rightarrow \bigcirc(e = \mathsf{ret}\ a)$
      $\eta_{\bullet_{\mathsf{beh}}}\ (c : \mathbb{C}) \Rightarrow \bigcirc(e = \mathsf{step}^c(\mathsf{ret}\ a))$

In other words, the (proof-relevant) predicate P associated with the F type is the collection of all values it can return ($a : \mathsf{TM}^+(A)$) and the cost charged ($bc : \bullet_{\mathsf{beh}}\mathbb{C}$). The case analysis on bc amounts to case analysis of the Kripke worlds, making sure that the canonicity property holds in both worlds. Instantiating A to be BOOL then yields the desired canonicity property. For more details about the construction of F and other definitions in the STC proof for **calf**, we refer the reader to Li and Harper [63] and the accompanying mechanization of this work where we formalize

representative definitions, namely the call-by-push-value adjunction F ⊣ U and the behavioral/cost phase distinction with effects $\mathsf{step}^c(e)$.

5 Related Work

This section discusses related work on prior formalizations of gluing, synthetic Tait computability, and logical relations, and on extensional proof assistants comparable to Istari.

5.1 Formalization of Gluing Argument

There has been significant recent progress on formalizing gluing arguments. For example, normalization gluing models for the simply-typed λ calculus have been mechanized in Cubical Agda and Rocq [1, 19]. Most notably, Kaposi and Pujet [55] formalized a canonicity gluing model for a dependent type theory presented as a category with families in Agda, using postulated constructs from observational type theory [10, 75]. This work differs in several respects.

Treatment of Syntax and Equalities. Both works begin with an algebraic signature of the syntax: theirs, a first-order category with families; ours, a higher-order abstract syntax presentation for a generalized algebraic theory. In such presentations, the equations of the syntax are expressed as propositional equalities. Both works are motivated by the observation that, for proof engineering purposes, it is advantageous to make as many of these equations hold judgmentally as possible. The approach Kaposi and Pujet [55] takes to *strictify* the equations is to instantiate the signature in a particular way, using quotient inductive-inductive types [54] and techniques from strict presheaves [74]. This construction ensures that all equations of the substitution calculus hold judgmentally, which considerably simplifies the subsequent gluing construction. Some equations, most notably β and η, remain propositional, so a small amount of transports and coercions are still required in their gluing proof. In the present work, the use of higher-order abstract syntax already avoids many equations about substitutions, most notably the naturality conditions. The remaining equations are uniformly turned into judgmental equalities by Istari's equality reflection, thereby avoiding the need for transports and bringing the formalization closer to the on-paper proofs. Consequently, this work does not make the effort to define the inductive syntax in type theory (in fact, the inductive syntax cannot be defined internally due to the use of higher-order abstract syntax), but rather works directly with the abstract signature of the syntax, without relying on initiality. The use of the modality framework does create some overhead in the number of equalities to manage, as explained in Section 1.3, but all such equalities are straightforward to handle with equality reflection as well. The drawback of our approach is that, in Kaposi and Pujet [55], judgmentally equal terms compute to the same thing automatically, while in our approach judgmentally equal terms are proved equal by manually invoking relevant equations using tactics.

Computational Content. Working synthetically exposes both advantages and limitations in the formalization. A primary limitation is that, although the proof is fully constructive, an evaluation algorithm is only external and cannot be extracted *internally* in Istari. This arises from the nature of synthetic Tait computability as explained in Section 2.5. By contrast, the algorithmic computational content of Kaposi and Pujet [55] is directly extractable internally in Agda. In exchange for this limitation internal language constructs can be flexibly reused across different object languages, which is exactly the *synthetic* advantage of STC. For example, although the gluing categories for the dependent type theory and **calf** differ, their internal languages share largely the same structures, allowing formalizations to use the same library of modal dependent type theory.

5.2 Formalization of Synthetic Tait Computability

The present work is not the first to consider mechanizing synthetic Tait computability. Sterling and Harper [88] anticipated that the main difficulty of the mechanization might lie in the treatment of phase, and suggested that definitional proof-irrelevance [34], as implemented in Agda and Rocq, could be used to implement the phase syn. The present work shows that this is not the main bottleneck: the fact that syn is propositional is used only once in the entire development, namely to show that ● is closed-modal. Huang [51] suggested using Cubical Agda cofibrations and glue types to simulate phases and the realignment axiom, but the semantics of cubical type theory is distant from the extensional internal language of the gluing category $\mathcal{G}$. Although Cubical Agda would allow a neat formulation of the ● modality as a higher inductive type, it is overkill because the gluing requires only set-level mathematics, and this work deliberately tries to avoid pervasive set-truncation and transports.

5.3 Mechanized Logical Relations

Besides the gluing construction and its synthetic variants, there is a rich literature on mechanizing logical relations and normalization-by-evaluation arguments, analytically rather than synthetically, for dependent, effectful, and/or linear type theories in Agda [3–5, 31] and Rocq [6, 35, 41, 65, 94, 99]. These mechanizations typically work with an operational semantics or a reduction system, and define logical relations over them. Such syntactic approaches are more flexible in the sense that they can directly manipulate arbitrary syntactic constructs. By contrast, the gluing approach applies most naturally to type theories given more semantic and categorical presentations, such as those with reduction-free equational theories. It is yet unclear how to apply gluing to languages that are primarily oriented around operational semantics. Another common challenge these mechanizations face is the

tedious reasoning about bindings and substitutions, though this can be mitigated by automated tools [79, 81]. The synthetic approach we take here avoids these challenges by using higher-order abstract syntax, in a style similar to mechanizing logical relations in Beluga [2, 23].

5.4 Other Related Proof Assistants

Istari [29] follows the computational type theory tradition of Martin-Löf type theory [67] and Nuprl [26], whose computational semantics inspired MetaPRL [48] and higher-dimensional proof assistants such as RedPRL [11, 13, 15]. Our development could in principle live in these systems, especially Nuprl, but we use Istari for its modern, stable implementation.

Outside this tradition, several type theories validate extensional principles. Observational type theory (OTT) [10, 75] validates function extensionality but not equality reflection, and its Agda embeddings support gluing arguments [55]. Sterling et al. [85, 86] propose a type theory with uniqueness of identity proofs using cubical syntax, currently without an implementation. Andromeda [18] implements an extensional type theory with equality reflection, but explores different proof-engineering choices than Istari.

6 Conclusion and Future Work

This work mechanizes synthetic Tait computability in the Istari proof assistant, which is based on extensional type theory. Our guiding principle throughout is that computer formalizations should track the on-paper proof as closely as possible, introducing as little extraneous technical machinery as possible. By taking full advantage of Istari's equality reflection and related features, the mechanization remains straightforward and largely preserves the concision and elegance of the original arguments, in line with the vision of Sterling [82], where STC reduces complex gluing constructions to simple type-theoretic reasoning.

This perspective addresses a common concern in the literature that such arguments are difficult to mechanize:

> Accordingly, these proofs [gluing] have a very extensional flavor, and as such are less amenable to implementation in a proof assistant based on intensional type theory. Moreover, the more sophisticated iterations [STC] rely on (multi)modal type theories as internal languages for feature-rich categories, for which mechanization is still in its infancy. Adjedj et al. [6]

Overall, we believe our development demonstrates the feasibility and benefits of mechanizing complex meta-theoretic arguments in an extensional proof assistant. The main price we pay for concise terms that closely resemble on-paper proofs is a certain amount of manual type-checking, a consequence of undecidable type-checking. In practice, however, we find that Istari's tactics and automation alleviate much of this burden to a manageable level. This work also serves as a proof of concept that extensional proof assistants with equality reflection provide a viable and promising alternative to intensional proof assistants for specific classes of applications, such as are demonstrated here.

6.1 Future Work

The present mechanization of synthetic Tait computability in Istari opens several avenues for future research:

Further Formalizations. The library and methods in this work should be able to extend to formalizations of more sophisticated STC proofs, such as:

1. binary homogeneous logical relations for parametricity of a module calculus [88];
2. binary heterogeneous logical relations for compiler, *e.g.* call-by-value to call-by-push-value compilation;
3. step-indexed logical relations for recursive types [87], ready for mechanization in Istari thanks to Istari's support for the future modality and guarded recursion;
4. normalization for dependent type theory [36, 39, 82, 84].

Mechanized Analytical Justification of STC. As mentioned in Sections 2.5 and 5, externalizing the internal language to the gluing category $\mathcal{G}$ is necessary to obtain the algorithmic content of STC proofs. Two possible approaches are:

1. Extend Istari's computational semantics to support a presheaf model, *i.e.* Kripke logical relations for the syntax-semantics phase distinction, so the internal mechanization is directly justified by Istari's semantics;
2. Mechanize the gluing categorical construction with respect to the internal language of STC in a proof assistant such as Agda, Rocq, or Lean, where significant category-theoretic formalizations already exist [40, 50, 68].

Data Availability Statement

The Istari mechanization described in this paper is available at Li et al. [64].

Acknowledgments

The authors thank Karl Crary, Jonathan Sterling, Harrison Grodin, and Matias Scharager for fruitful discussions and proofreading of this work, and the anonymous reviewers for their thoughtful comments. This material is based upon work supported by the National Science Foundation under grant numbers 2211996 and 2442461, and by the United States Air Force Office of Scientific Research under grant numbers FA9550-21-0009, FA9550-23-1-0434, and FA9550-21-1-0385 (Tristan Nguyen, program manager). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the United States Air Force Office of Scientific Research.

References

[1] The 1Lab Development Team. 2024. *The 1Lab Normalisation by evaluation.* https://1lab.dev/Cat.CartesianClosed.Free.html#normalisation-by-evaluation

[2] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. 2019. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming* 29 (2019), e19. doi:10.1017/S0956796819000170

[3] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proc. ACM Program. Lang.* 7, ICFP, Article 220 (Aug. 2023), 35 pages. doi:10.1145/3607862

[4] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. doi:10.1145/3158111

[5] Emmanuel Suárez Acevedo and Stephanie Weirich. 2023. Making Logical Relations More Relatable (Proof Pearl). arXiv:2309.15724 [cs.PL] https://arxiv.org/abs/2309.15724

[6] Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. doi:10.1145/3636501.3636951

[7] S.F. Allen, M. Bickford, R.L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. 2006. Innovations in computational type theory using Nuprl. *Journal of Applied Logic* 4, 4 (2006), 428–469. doi:10.1016/j.jal.2005.10.005 Towards Computer Aided Mathematics.

[8] Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. 2024. Internal Parametricity, without an Interval. *Proc. ACM Program. Lang.* 8, POPL, Article 78 (Jan. 2024), 30 pages. doi:10.1145/3632920

[9] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1996. Reduction-free Normalisation for System F. (1996). Available at https://people.cs.nott.ac.uk/psztxa/publ/f97.pdf.

[10] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now!. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (Freiburg, Germany) *(PLPV '07)*. Association for Computing Machinery, New York, NY, USA, 57–68. doi:10.1145/1292597.1292608

[11] Carlo Angiuli. 2019. *Computational Semantics of Cartesian Cubical Type Theory.* Ph. D. Dissertation. Carnegie Mellon University. https://carloangiuli.com/papers/thesis.pdf

[12] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. 2021. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science* 31, 4 (2021), 424–468. doi:10.1017/S0960129521000347

[13] Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, and Jonathan Sterling. 2018. The RedPRL Proof Assistant (Invited Paper). In Proceedings of the 13th International Workshop on *Logical Frameworks and Meta-Languages: Theory and Practice,* Oxford, UK, 7th July 2018 *(Electronic Proceedings in Theoretical Computer Science, Vol. 274)*, Frédéric Blanqui and Giselle Reis (Eds.). Open Publishing Association, 1–10. doi:10.4204/EPTCS.274.1

[14] Carlo Angiuli and Daniel Gratzer. 2025. *Principles of Dependent Type Theory.* https://carloangiuli.com/papers/type-theory-book.pdf

[15] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 119)*, Dan R. Ghica and Achim Jung (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:17. doi:10.4230/LIPIcs.CSL.2018.6

[16] Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., USA.

[17] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. 2005. Mechanized metatheory for the masses: the PoplMark challenge. In *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics* (Oxford, UK) *(TPHOLs'05)*. Springer-Verlag, Berlin, Heidelberg, 50–65. doi:10.1007/11541868_4

[18] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. 2018. Design and Implementation of the Andromeda Proof Assistant. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 97)*, Silvia Ghilezan, Herman Geuvers, and Jelena Ivetic (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:31. doi:10.4230/LIPIcs.TYPES.2016.5

[19] David G. Berry and Marcelo P. Fiore. 2025. Formal P-Category Theory and Normalization by Evaluation in Rocq. arXiv:2505.07780 [cs.LO] https://arxiv.org/abs/2505.07780

[20] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2016. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 23:1–23:17. doi:10.4230/LIPIcs.CSL.2016.23

[21] Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. 2023. For the Metatheory of Type Theory, Internal Sconing Is Enough. In *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 260)*, Marco Gaboardi and Femke van Raamsdonk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:23. doi:10.4230/LIPIcs.FSCD.2023.18

[22] N. Bourbaki, M. Artin, A. Grothendieck, P. Deligne, and J.L. Verdier. 1983. *Theorie des Topos et Cohomologie Etale des Schemas. Seminaire de Geometrie Algebrique du Bois-Marie 1963-1964 (SGA 4): Tome 1.* Springer Berlin Heidelberg.

[23] Andrew Cave and Brigitte Pientka. 2018. Mechanizing proofs with logical relations – Kripke-style. *Mathematical Structures in Computer Science* 28, 9 (2018), 1606–1638. doi:10.1017/S0960129518000154

[24] Jesper Cockx. 2020. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In *25th International Conference on Types for Proofs and Programs (TYPES 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 175)*, Marc Bezem and Assia Mahboubi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:27. doi:10.4230/LIPIcs.TYPES.2019.2

[25] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:34. doi:10.4230/LIPIcs.TYPES.2015.5

[26] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. 1986. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ.

[27] Thierry Coquand. 2018. Canonicity and normalisation for Dependent Type Theory. arXiv:1810.09367 [cs.PL] https://arxiv.org/abs/1810.09367

[28] Karl Crary. 2025. Istari. GitHub. https://github.com/kcrary/istari

[29] Karl Crary. 2025. The Istari Proof Assistant. https://istarilogic.org/. Accessed: 2025-09-08.

[30] Roy L. Crole. 1994. *Categories for Types.* Cambridge University Press.

[31] Nils Anders Danielsson, Naïm Favier, and Ondřej Kubánek. 2026. Normalisation for First-Class Universe Levels. *Proc. ACM Program. Lang.* POPL.

[32] Samuel Eilenberg and John C. Moore. 1965. Adjoint functors and triples. *Illinois Journal of Mathematics* 9 (1965), 381–398. https://api.semanticscholar.org/CorpusID:50713743

[33] Marcelo Fiore. 2002. Semantic analysis of normalisation by evaluation for typed lambda calculus. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '02)*. Association for Computing Machinery, 26–37. doi:10.1145/571157.571161

[34] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (Jan. 2019), 28 pages. doi:10.1145/3290316

[35] Tarakaram Gollamudi, Jules Jacobs, Yue Yao, and Stephanie Balzer. 2025. A Semantic Logical Relation for Termination of Intuitionistic Linear Logic Session Types. *11th International Workshop on Coq for Programming Languages (CoqPL)* (2025). https://www.cs.cmu.edu/~balzers/publications/coqpl-2025.pdf

[36] Daniel Gratzer. 2022. Normalization for Multimodal Type Theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Haifa Israel, 1–13. doi:10.1145/3531130.3532398

[37] Daniel Gratzer, Michael Shulman, and Jonathan Sterling. 2024. Strict universes for Grothendieck topoi. https://arxiv.org/abs/2202.12012

[38] Daniel Gratzer and Jonathan Sterling. 2021. Syntactic categories for dependent type theory: sketching and adequacy. arXiv:2012.10783 [cs.LO] https://arxiv.org/abs/2012.10783

[39] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2025. Controlling unfolding in type theory. (2025). doi:10.48550/ARXIV.2210.05420 Mathematical Structures in Computer Science.

[40] Daniel R. Grayson, Benedikt Ahrens, Ralph Matthes, Niels van der Weide, Anders Mörtberg, cathlelay, Langston Barrett, Peter LeFanu Lumsdaine, Marco Maggesi, Vladimir Voevodsky, Tomi Pannila, Gianluca Amato, Michael Lindgren, Mitchell Riley, Skantz, CalosciMatteo, Arnoud van der Leer, Tobias-Schmude, Daniel Frumin, Auke Booij, Hichem Saghrouni, Dennis, varkor, Dimitris Tsementzis, niccoloveltri, Anthony Bordg, Dominik Kirst, Karl Palmskog, Satoshi Kura, and Tamara von Glehn. 2025. UniMath/UniMath: v20250923. https://doi.org/10.5281/zenodo.17186647. doi:10.5281/zenodo.17186647 Version v20250923.

[41] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL, Article 10 (Jan. 2021), 29 pages. doi:10.1145/3434291

[42] Harrison Grodin and Robert Harper. 2024. Amortized Analysis via Coalgebra. *Electronic Notes in Theoretical Informatics and Computer Science* Volume 4 - Proceedings of MFPS XL (Dec. 2024). doi:10.46298/entics.14797

[43] Harrison Grodin, Runming Li, and Robert Harper. 2026. Abstraction Functions as Types. *Proc. ACM Program. Lang.* 10, POPL, Article 31 (Jan. 2026), 28 pages. doi:10.1145/3776673

[44] Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 10:273–10:301. doi:10.1145/3632852

[45] Robert Harper. 2021. An Equational Logical Framework for Type Theories. https://arxiv.org/abs/2106.01484

[46] Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. 40, 1 (1993), 143–184. doi:10.1145/138027.138060

[47] Robert Harper, John C. Mitchell, and Eugenio Moggi. 1989. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 341–354. doi:10.1145/96709.96744

[48] Jason Hickey, Aleksey Nogin, Robert Constable, Brian Aydemir, Yegor Bryukhov, Richard Eaton, Adam Granicz, Christoph Kreitz, Vladimir Krupski, Lori Lorigo, Carl Witty, and Xin Yu. 2003. MetaPRL - A Modular Logical Environment. (10 2003).

[49] D.J. Howe. 1989. Equality in lazy computation systems. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*. 198–203. doi:10.1109/LICS.1989.39174

[50] Jason Z. S. Hu and Jacques Carette. 2021. Formalizing category theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Virtual, Denmark) *(CPP 2021)*. Association for Computing Machinery, New York, NY, USA, 327–342. doi:10.1145/3437992.3439922

[51] Xu Huang. 2023. Synthetic Tait Computability the Hard Way. arXiv:2310.02051 [cs.LO] https://arxiv.org/abs/2310.02051

[52] Peter T. Johnstone. 2002. *Sketches of an Elephant: A Topos Theory Compendium, Volume 1*. Clarendon Press, Oxford.

[53] Ambrus Kaposi, Simon Huber, and Christian Sattler. 2019. Gluing for Type Theory. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 25:1–25:19. doi:10.4230/LIPIcs.FSCD.2019.25

[54] Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.* 3, POPL, Article 2 (Jan. 2019), 24 pages. doi:10.1145/3290315

[55] Ambrus Kaposi and Loïc Pujet. 2025. Type Theory in Type Theory using a Strictified Syntax. *Proc. ACM Program. Lang.* ICFP (Aug. 2025), 31 pages. doi:10.1145/3747535

[56] Ambrus Kaposi and Szumi Xie. 2024. Second-Order Generalised Algebraic Theories: Signatures and First-Order Semantics. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:24. doi:10.4230/LIPIcs.FSCD.2024.10

[57] Lukas Kebuladze. 2025. *Amortized Analysis of Splay Trees via a Lax Homomorphism*. Technical Report. Carnegie Mellon University. https://www.cs.cmu.edu/~runmingl/student/kebuladze-splay.pdf

[58] András Kovács. 2026. Canonicity for Indexed Inductive-Recursive Types. *Proc. ACM Program. Lang.* 10, POPL, Article 43 (Jan. 2026), 29 pages. doi:10.1145/3776685

[59] Saul Kripke. 1963. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica* 16 (1963), 83–94.

[60] F. William Lawvere. 1963. Functorial Semantics of Algebraic Theories. 50, 5 (1963), 869–872. doi:10.1073/pnas.50.5.869

[61] Paul Blain Levy. 2003. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Springer Netherlands, Dordrecht. doi:10.1007/978-94-007-0954-6

[62] Runming Li, Harrison Grodin, and Robert Harper. 2023. A Verified Cost Analysis of Joinable Red-Black Trees. arXiv:2309.11056 [cs.PL] https://arxiv.org/abs/2309.11056

[63] Runming Li and Robert Harper. 2025. Canonicity for Cost-Aware Logical Framework via Synthetic Tait Computability. arXiv:2504.12464 (April 2025). doi:10.48550/arXiv.2504.12464 arXiv:2504.12464 [cs].

[64] Runming Li, Yue Yao, and Robert Harper. 2025. *Mechanizing Synthetic Tait Computability in Istari (Artifact)*. doi:10.5281/zenodo.17808361

[65] Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2025. Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories. (2025). https://electriclam.com/papers/mltt.pdf

[66] Per Martin-Löf. 1984. Intuitionistic Type Theory. (1984). https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-retypeset-1984.pdf Lecture notes Padua 1984, Bibliopolis, Napoli.

[67] Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Logic, Methodology and Philosophy of Science VI*, L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 104. Elsevier, 153–175. doi:10.1016/S0049-237X(09)70189-2

[68] The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New Orleans, LA, USA) *(CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. doi:10.1145/3372885.3373824

[69] Robin Milner. 1972. *Logic for Computable Functions: description of a machine implementation.* Technical Report. Stanford, CA, USA.

[70] John C. Mitchell and Andre Scedrov. 1992. Notes on Sconing and Relators. In *Selected Papers from the Workshop on Computer Science Logic (CSL '92)*. Springer-Verlag, Berlin, Heidelberg, 352–378.

[71] Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL, Article 9 (Jan. 2022), 31 pages. doi:10.1145/3498670

[72] Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's type theory: an introduction.* Clarendon Press, USA.

[73] Ian Orton and Andrew M. Pitts. 2016. Axioms for Modelling Cubical Type Theory in a Topos. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 62)*, Jean-Marc Talbot and Laurent Regnier (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 24:1–24:19. doi:10.4230/LIPIcs.CSL.2016.24

[74] Pierre-Marie Pédrot. 2020. Russian Constructivism in a Prefascist Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 782–794. doi:10.1145/3373718.3394740

[75] Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proc. ACM Program. Lang.* 6, POPL, Article 32 (Jan. 2022), 27 pages. doi:10.1145/3498693

[76] Emily Riehl. 2016. *Category Theory in Context.* Dover Publications.

[77] Emily Riehl and Michael Shulman. 2017. A Type Theory for Synthetic ∞-Categories. *Higher Structures* 1, 1 (Dec. 2017), 147–224. doi:10.21136/HS.2017.06

[78] Egbert Rijke, Michael Shulman, and Bas Spitters. 2020. Modalities in homotopy type theory. *Logical Methods in Computer Science* Volume 16, Issue 1, Article 2 (Jan 2020). doi:10.23638/LMCS-16(1:2)2020

[79] Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015 (LNAI)*, Xingyuan Zhang and Christian Urban (Eds.). Springer-Verlag.

[80] Michael Shulman. 2015. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science* 25, 5 (2015), 1203–1277. doi:10.1017/S0960129514000565

[81] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 166–180. doi:10.1145/3293880.3294101

[82] Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory.* Ph. D. Dissertation. Carnegie Mellon University. doi:10.5281/zenodo.6990769 Version 1.1, revised May 2022.

[83] Jonathan Sterling. 2022. Naïve Logical Relations in Synthetic Tait Computability. https://www.jonmsterling.com/bafkrmialyvkzh6w6snnzr3k4h2b62bztsk4le57idughqik24bltinieki.pdf

[84] Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–15. doi:10.1109/LICS52264.2021.9470719

[85] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2019. Cubical Syntax for Reflection-Free Extensional Equality. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:25. doi:10.4230/LIPIcs.FSCD.2019.31

[86] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. 2022. A Cubical Language for Bishop Sets. *Logical Methods in Computer Science* Volume 18, Issue 1 (March 2022). doi:10.46298/lmcs-18(1:43)2022

[87] Jonathan Sterling, Daniel Gratzer, and Lars Birkedal. 2023. Denotational semantics of general store and polymorphism. arXiv:2210.02169 [cs.PL] https://arxiv.org/abs/2210.02169

[88] Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6 (Dec. 2021), 1–47. doi:10.1145/3474834

[89] Jonathan Sterling and Robert Harper. 2022. Sheaf Semantics of Termination-Insensitive Noninterference. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 5:1–5:19. doi:10.4230/LIPIcs.FSCD.2022.5

[90] Jonathan Sterling and Bas Spitters. 2018. Normalization by gluing for free λ-theories. https://arxiv.org/abs/1809.08646

[91] W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

[92] The RedPRL Development Team. 2018. redtt. https://www.github.com/RedPRL/redtt

[93] Taichi Uemura. 2021. *Abstract and Concrete Type Theories.* Ph. D. Dissertation. Universiteit van Amsterdam. https://www.illc.uva.nl/cms/Research/Publications/Dissertations/DS-2021-09.text.pdf

[94] Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq formalization of normalization by evaluation for Martin-Löf type theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 266–279. doi:10.1145/3167091

[95] Zhixuan Yang. 2024. *Structure and Language of Higher-Order Algebraic Effects.* Ph. D. Dissertation. Imperial College London. https://yangzhixuan.github.io/pdf/yang-thesis.pdf

[96] Zhixuan Yang. 2025. Revisiting the Logical Framework for Locally Cartesian Closed Categories. https://yangzhixuan.github.io/pdf/lcclf.pdf

[97] Zhixuan Yang and Nicolas Wu. 2026. Handling Higher-Order Effectful Operations with Judgemental Monadic Laws. *Proc. ACM Program. Lang.* POPL (Jan. 2026). doi:10.48550/arXiv.2511.05739

[98] Tesla Zhang. 2024. Three non-cubical applications of extension types. arXiv:2311.05658 [cs.PL] https://arxiv.org/abs/2311.05658

[99] Tesla Zhang, Sonya Simkin, Rui Li, Yue Yao, and Stephanie Balzer. 2025. A Language-Agnostic Logical Relation for Message-Passing Protocols. arXiv:2506.10026 [cs.PL] https://arxiv.org/abs/2506.10026

[100] Andrew Zhou. 2025. *Formally Verified Cost of the Parallel Prefix Sum Algorithm.* Technical Report. Carnegie Mellon University. https://www.cs.cmu.edu/~runmingl/student/zhou-scan.pdf

Received 2025-09-12; accepted 2025-11-13