Controlling unfolding in type theory

DANIEL GRATZER, Aarhus University, Denmark
JONATHAN STERLING, University of Cambridge, United Kingdom
CARLO ANGIULI, Indiana University, United States
THIERRY COQUAND, University of Gothenburg, Sweden
LARS BIRKEDAL, Aarhus University, Denmark

We present a new way to control the unfolding of definitions in dependent type theory. Traditionally, proof assistants require users to fix whether each definition will or will not be unfolded in the remainder of a development; unfolding definitions is often necessary in order to reason about them, but an excess of unfolding can result in brittle proofs and intractably large proof goals. In our system, definitions are by default not unfolded, but users can selectively unfold them in a local manner. We justify our mechanism by means of elaboration to a core theory with *extension types*—a connective first introduced in the context of homotopy type theory—and by establishing a normalization theorem for our core calculus. We have implemented controlled unfolding in the `cooltt` proof assistant, inspiring an independent implementation in Agda.

1 INTRODUCTION

In dependent type theory, terms are type checked modulo definitional equality, a congruence generated by α-, β-, and η-laws, as well as unfolding of definitions. Unfolding definitions is to some extent a convenience that allows type checkers to silently discharge many proof obligations, *e.g.* a list of length $1 + 1$ is without further annotation also a list of length 2. It is by no means the case, however, that we always want a given definition to unfold:

- *Modularity*: Dependent types are famously sensitive to the smallest changes to definitions, such as whether (+) recurs on its first or its second argument. If we plan to change a definition in the future, it may be desirable to avoid exposing its implementation to the type checker.
- *Usability*: While unfolding may simplify proof states, it also has the potential to complicate them, resulting in unreadable subgoals, error messages, *etc.* A user may find that certain definitions are likely to be problematic in this way, and thus opt not to unfold them.

Many proof assistants accordingly have implementation-level support for marking definitions *opaque* (unable to be unfolded), including Agda's `abstract` [49] and Coq's `Qed` [50]. But unfolding definitions is not merely a matter of convenience: to reason about a function, we must unfold it. For example, if we make the definition of (+) opaque, then (+) is indistinguishable from a variable of type $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and so cannot be shown to be commutative, satisfy $1 + 1 = 2$, *etc.*

In practice, proof assistants resolve this contradiction by adopting an intermediate stance: definitions are *transparent* (unfolded during type checking) by default, but users are given some control over their unfolding. Coq provides conversion tactics (`cbv`, `simpl`, *etc.*) for applying definitional equalities, each of which accepts a list of definitions to unfold; its `Opaque` and `Transparent` commands toggle the default unfolding behavior of a transparent definition; and the SSReflect tactic language natively supports a "`locking`" idiom for controlling when definitions unfold [18]. Agda allows users to group multiple definitions into a single `abstract` block, inside of which those definitions are transparent and outside of which they are opaque; this allows users to define a

Authors' addresses: Daniel Gratzer, gratzer@cs.au.dk, Aarhus University, Denmark; Jonathan Sterling, js2878@cl.cam.ac.uk, University of Cambridge, United Kingdom; Carlo Angiuli, cangiuli@iu.edu, Indiana University, United States; Thierry Coquand, coquand@chalmers.se, University of Gothenburg, Sweden; Lars Birkedal, birkedal@cs.au.dk, Aarhus University, Denmark.

function, prove all lemmas that depend on the function's definition, and then irreversibly make the function and lemmas opaque.

These mechanisms for controlling unfolding pose interesting tradeoffs for users: which definitions should be transparent, and which should be opaque? Transparency is in some cases necessary and in many cases convenient, but is problematic both from an engineering perspective—because any edit to a transparent definition can break the well-typedness of any number of its use sites—and from a performance perspective—because checking definitional equality of type indices often requires unfolding nested definitions into large normal forms.

In addition, the behavior of these mechanisms is more subtle than it may at first appear. In Agda, definitions within `abstract` blocks are transparent to other definitions in the same block, but opaque to the *types* of those definitions; without such a stipulation, those types may cease to be well-formed when the earlier definition is made opaque. Furthermore, `abstract` blocks are anti-modular, requiring users to anticipate all future lemmas about definitions in a block.[1] Coq's conversion tactics are more flexible than Agda's `abstract` blocks, but being tactics, their behavior can be harder to predict. The `lock` idiom in SSReflect is more predictable because it creates opaque definitions, but comes in four different variations to simplify its use in practice.

1.1 Contributions

We propose a mechanism for fine-grained control over the unfolding of definitions in dependent type theory. We introduce language-level primitives for *controlled unfolding* that are elaborated into a core calculus with *extension types*, a connective first introduced by Riehl and Shulman [38]. We justify our elaboration algorithm by establishing a normalization theorem (and hence the decidability of type checking and injectivity of type constructors) for our core calculus, and we have implemented our system for controlled unfolding in the experimental `cooltt` proof assistant [37].

Definitions in our framework are opaque by default, but can be selectively and locally unfolded as if they were transparent. Our system is finer-grained and more modular than Agda's `abstract` blocks: we need not collect all lemmas that unfold a given definition into a single block, making our mechanism better suited to libraries. Our primitives have more predictable meaning and performance than Coq's unfolding tactics[2] because they are implemented by straightforward elaboration into a core Martin-Löf type theory extended with new types and declaration forms.

In particular, we refine earlier approaches to representing definitions within type theory [12, 22, 34, 42] in order to more faithfully represent definitions as they are actually used in practice: as neither fully opaque or transparent but instead a mix of the two. Drawing inspiration from cubical type theory [2, 3, 9], we extend **MLTT** with proof-irrelevant proposition symbols p, dependent products $\{p\}\, A$ over those propositions, and extension types $\{A \mid p \hookrightarrow a\}$, the subtype of A consisting of the elements of A that definitionally equal a under the assumption that p is true. For readers familiar with cubical type theory, extension types are similar to path types ($\mathsf{Path}\ A\ a_0\ a_1$), which classify functions out of an abstract interval $\mathbb{I}$ that are definitionally equal to a_0 and a_1 when evaluated at the interval's endpoints $0, 1 : \mathbb{I}$.

Encoding definitions through particular types confers a number of benefits. For instance, our mechanism for definitions and unfolding are automatically invariant under definitional equivalence: replacing one term by a definitionally-equal alternative cannot change the unfolding behavior of a program. Furthermore, using extension types to encode definitions ensures our elaboration algorithm is extremely modular and predictable: the rules for extension types are simple and,

[1] Indeed, the Agda standard library [47] currently uses `abstract` only once.
[2] https://github.com/coq/coq/blob/V8.16.0/theories/ssr/ssreflect.v/#L388

once grasped, it becomes easy to predict the interactions between unfolding definitions and other features within the language. This elaboration algorithm then serves as a *reference* for the behavior of our mechanism, against which other implementation strategies may be checked.

Like many elaboration algorithms for dependent type theory, executing our elaboration algorithm requires deciding the equality of types in the core language. To show that our elaboration algorithm can be implemented, we prove a *normalization* theorem for our core calculus, characterizing its definitional equivalence classes of types and terms and as a corollary establishing the decidability of type checking. This is more subtle than it may appear: the heart of our normalization proof amounts to correctly tracking when definitions are allowed to unfold as well as when they should remain opaque. In the face of higher-order programs and dependent types, this is quite difficult.

Another benefit to shifting from opaque definitions to extension types is their well-studied metatheory. Specifically, we are able to adapt and extend Sterling's technique of synthetic Tait computability / STC [39, 41, 42] to prove normalization for our core language. Our proof is fully constructive, an improvement on the prior work of Sterling and Angiuli [41]; we have also corrected an error in the handling of universes in an earlier revision of Sterling's doctoral dissertation [39] that was detected while preparing this paper.

1.2 Outline

In Section 2 we introduce our controlled unfolding primitives by way of examples, and in Section 3 we walk through how these examples are elaborated into our core language of type theory with proposition symbols and extension types. In Section 4 we present our elaboration algorithm, and in Section 5 we discuss our implementation of the above in the `cooltt` proof assistant. In Section 6 we establish normalization and its corollaries for our core calculus. We conclude with a discussion of related work in Section 7.

2 A SURFACE LANGUAGE WITH CONTROLLED UNFOLDING

We begin by describing an Agda-like surface language for a dependent type theory with controlled unfolding. In Section 4 we will give precise meaning to this language by explaining how to elaborate it into our core calculus; for now we proceed by example, introducing our new primitives bit by bit. Our examples will concern the inductively defined natural numbers and their addition function:

$$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$
$$\mathsf{ze} + n = n$$
$$\mathsf{su}\ m + n = \mathsf{su}\ (m + n)$$

2.1 A simple dependency: length-indexed vectors

In our language, definitions such as (+) are opaque by default—they are not unfolded automatically. To illustrate the need to *selectively* unfold (+), consider the indexed inductive type of length-indexed vectors with the following constructors:

$$[\,] : \mathsf{vec}\ \mathsf{ze}\ A$$
$$(::) : A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (\mathsf{su}\ n)\ A$$

Suppose we attempt to define the *append* operation on vectors by dependent pattern matching on the first vector. Our goals would be as follows:

$$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$$
$$[\,] \oplus v = \ ?\ : \mathsf{vec}\ (\mathsf{ze} + n)\ A$$
$$(a :: u) \oplus v = \ ?\ : \mathsf{vec}\ (\mathsf{su}\ m + n)\ A$$

As it stands, the goals above are in normal form and cannot be proved; however, we may indicate that the definition of $(+)$ should be unfolded within the definition of $(\oplus)$ by adding the following top-level **unfolds** annotation:

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$

With our new declaration, the goals simplify:

$[\,] \oplus v = \ ?\ : \mathsf{vec}\ n\ A$
$(a :: u) \oplus v = \ ?\ : \mathsf{vec}\ (\mathsf{su}\ (m + n))\ A$

The first goal is solved with v itself; for the second goal, we begin by applying the vcons constructor:

$(a :: u) \oplus v = a :: \ ?\ : \mathsf{vec}\ (m + n)\ A$

The remaining goal is just our induction hypothesis $u \oplus v$. All in all, we have:

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$[\,] \oplus v = v$
$(a :: u) \oplus v = a :: (u \oplus v)$

2.2 Transitive unfolding

Now suppose we want to prove that map distributes over $(\oplus)$. In doing so we will certainly need to unfold map, but it turns out this will not be enough:

$\mathsf{map} : (A \to B) \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ n\ B$
$\mathsf{map}\ f\ [\,] = [\,]$
$\mathsf{map}\ f\ (a :: u) = f\ a :: \mathsf{map}\ f\ u$

map-$\oplus$ **unfolds** map
map-$\oplus$: $(f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{map}\ f\ (u \oplus v) \equiv \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v$
map-$\oplus\ f\ [\,]\ v = \ ?\ : \mathsf{map}\ f\ ([\,] \oplus v) \equiv [\,] \oplus \mathsf{map}\ f\ v$
map-$\oplus\ f\ (a :: u)\ v = \ ?\ : \mathsf{map}\ f\ (a :: u) \oplus v \equiv (f\ a :: \mathsf{map}\ f\ u) \oplus \mathsf{map}\ f\ v$

To make further progress we must also unfold $(\oplus)$:

map-$\oplus$ **unfolds** map; $(\oplus)$
map-$\oplus$: $(f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{map}\ f\ (u \oplus v) \equiv \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v$
map-$\oplus\ f\ [\,]\ v = \ ?\ : \mathsf{map}\ f\ v \equiv \mathsf{map}\ f\ v$
map-$\oplus\ f\ (a :: u)\ v = \ ?\ : f\ a :: \mathsf{map}\ f\ (u \oplus v) \equiv (f\ a :: \mathsf{map}\ f\ u) \oplus \mathsf{map}\ f\ v$

In our language, unfolding $(\oplus)$ has the side effect of *also* unfolding $(+)$: in other words, unfolding is *transitive*. To see why this is the case, observe that the unfolding of $(a :: u) \oplus v : \mathsf{vec}\ (\mathsf{su}\ m + n)\ A$, namely $a :: (u \oplus v) : \mathsf{vec}\ (\mathsf{su}\ (m + n))\ A$, would otherwise not be well-typed. From an implementation perspective, one can think of the transitivity of unfolding as necessary for *subject reduction*. Having unfolded map, $(\oplus)$, and thus $(+)$, we complete our definition:

$\mathsf{cong} : (f : A \to B) \to a \equiv a' \to f\ a \equiv f\ a'$
$\mathsf{cong}\ f\ \mathsf{refl} = \mathsf{refl}$

map-$\oplus$ **unfolds** map; $(\oplus)$
map-$\oplus$: $(f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{map}\ f\ (u \oplus v) \equiv \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v$
map-$\oplus\ f\ [\,]\ v = \mathsf{refl}$

```
map-⊕ f (a :: u) v = cong (f a ::) (map-⊕ f u v)
```

2.3 Recovering unconditionally transparent/opaque definitions

There are also times when we intend a given definition to be a fully transparent *abbreviation*, in the sense of being unfolded automatically whenever possible. We indicate this with an **abbreviation** declaration:

```
abbreviation singleton
singleton : A → vec (su ze) A
singleton a = a :: [ ]
```

Then the following lemma can be defined without any explicit unfolding:

```
abbrv-example : singleton 5 ≡ (5 :: [ ])
abbrv-example = refl
```

The meaning of the **abbreviation** keyword must account for unfolding constraints. For instance, what would it mean to make map-⊕ an abbreviation?

```
abbreviation map-⊕
map-⊕ unfolds map; (⊕)
...
```

We cannot unfold map-⊕ in all contexts, because its definition is only well-typed when map and (⊕) are unfolded. The meaning of this declaration must, therefore, be that map-⊕ shall be unfolded *just as soon as* map and (⊕) are unfolded. In other words, **abbreviation** ϑ followed by ϑ **unfolds** $\kappa_1; \ldots; \kappa_n$ means that unfolding ϑ is synonymous with unfolding all of $\kappa_1; \ldots; \kappa_n$.

Conversely, we may intend a given definition *never* to unfold, which we may indicate by a corresponding **abstract** declaration. Because definitions in our system do not automatically unfold, the force of **abstract** ϑ is simply to prohibit users from including ϑ in any subsequent **unfolds** annotations.

Remark 1. *A slight variation on our system can recover the behavior of Agda's* `abstract` *blocks by* limiting *the scope in which a definition ϑ can be unfolded; the transitivity of unfolding dictates that any definition ϑ' that unfolds ϑ cannot itself be unfolded once we leave that scope. We leave the details to future work.*

2.4 Unfolding within the type

The effect of a ϑ **unfolds** $\kappa_1; \ldots; \kappa_n$ declaration is to make $\kappa_1; \ldots \kappa_n$ unfold within the *definition* of ϑ, but still not within its type; it will happen, however, that a *type* might not be expressible without some unfolding. First we will show how to accommodate this situation using only features we have introduced so far, and then in Section 2.5 we will devise a more general and ergonomic solution.

Consider the left-unit law for (⊕): in order to state that a vector u is equal to the vector $[\,] \oplus u$, we must contend with their differing types vec n A and vec (ze + n) A respectively. One approach is to rewrite along the left-unit law for $\mathbb{N}$; indeed, to state the *right-unit* law for (⊕) one must rewrite along the right-unit law for $\mathbb{N}$. But here, because (+) computes on its first argument, vec n A and vec (ze + n) A would be definitionally equal types if we could unfold (+).

In order to formulate the left-unit law for (⊕), we start by defining its *type* as an abbreviation that unfolds (+):

```
abbreviation ⊕-left-unit-type
⊕-left-unit-type unfolds (+)
⊕-left-unit-type : vec n A → Type
⊕-left-unit-type u = [ ] ⊕ u ≡ u
```

Now we may state the intended lemma using the type defined above:

```
⊕-left-unit : (u : vec n A) → ⊕-left-unit-type u
⊕-left-unit u = ? : ⊕-left-unit-type u
```

Clearly we must unfold (+) and thus ⊕-left-unit-type to simplify our goal:

```
⊕-left-unit unfolds (+)
⊕-left-unit : (u : vec n A) → ⊕-left-unit-type u
⊕-left-unit u = ? : [ ] ⊕ u ≡ u
```

We complete the proof by unfolding (⊕) itself, which transitively unfolds (+):

```
⊕-left-unit unfolds (⊕)
⊕-left-unit : (u : vec n A) → ⊕-left-unit-type u
⊕-left-unit u = refl
```

2.5 Unfolding within subexpressions

We have just demonstrated how to unfold definitions within the *type* of a declaration by defining that type as an additional declaration; using the same technique, we can introduce unfoldings within *any subexpression* by hoisting that subexpression to a top-level definition with its own unfolding constraint.

Unfolding within the type, revisited. Rather than repeating the somewhat verbose pattern of Section 2.4, we abstract it as a new language feature that is easily eliminated by elaboration. In particular, we introduce a new *expression* former **unfold** κ **in** M that can be placed in any expression context. Let us replay the example from Section 2.4, but using **unfold** rather than an auxiliary definition:

```
⊕-left-unit : (u : vec n A) → unfold (+) in [ ] ⊕ u ≡ u
⊕-left-unit u = ? : unfold (+) in [ ] ⊕ u ≡ u
```

The type **unfold** (+) **in** [ ] ⊕ $u \equiv u$ is in normal form; the only way to simplify it is to unfold (+). We could do this with another inline **unfold** expression (see ⊕-left-unit′ below), but here we will use a top-level declaration:

```
⊕-left-unit unfolds (+)
⊕-left-unit : (u : vec n A) → unfold (+) in [ ] ⊕ u ≡ u
⊕-left-unit u = ? : [ ] ⊕ u ≡ u
```

By virtue of the above, the **unfold** expression in our hole has computed away and we are left with `? : [ ] ⊕ u ≡ u` as (⊕) is still abstract in this scope. To make progress, we *strengthen* the declaration to unfold (⊕) in addition to (+):

```
⊕-left-unit unfolds (⊕)
⊕-left-unit : (u : vec n A) → unfold (+) in [ ] ⊕ u ≡ u
⊕-left-unit u = refl
```

The meaning of the code above is exactly as described in Section 2.4: the **unfold** scope is elaborated to a new top-level **abbreviation** that unfolds (+).

Expression-level vs. *top-level unfolding.* We noted in our definition of ⊕-left-unit above that we could have replaced the top-level **unfolds** (⊕) directive of ⊕-left-unit with the new expression-level **unfold** (⊕) **in** as follows:

```
⊕-left-unit′ : (u : vec n A) → unfold (+) in [ ] ⊕ u ≡ u
⊕-left-unit′ u = unfold (⊕) in refl
```

The resulting definition of ⊕-left-unit′ has slightly different behavior than ⊕-left-unit above: whereas unfolding ⊕-left-unit causes (⊕) to unfold transitively, we can unfold ⊕-left-unit′ without unfolding (⊕)—at the cost of **unfold** (⊕) expressions appearing in our goal. This more granular behavior may be desirable in some cases, and it is a strength of our language and its elaborative semantics that the programmer can manipulate unfolding in such a fine-grained manner.

For completeness, we show the elaborated version of ⊕-left-unit′ resulting from eliminating expression-level unfolding from the definition. We defer a systematic discussion of this transformation till Section 4.

abbreviation ⊕-left-unit′-type
⊕-left-unit′-type **unfolds** (+)
⊕-left-unit′-type : vec n A → Type
⊕-left-unit′-type u = [ ] ⊕ u ≡ u

abbreviation ⊕-left-unit′-body
⊕-left-unit′-body **unfolds** (⊕)
⊕-left-unit′-body : (u : vec n A) → ⊕-left-unit′-type u
⊕-left-unit′-body u = refl

⊕-left-unit′ : (u : vec n A) → ⊕-left-unit′-type u
⊕-left-unit′ u = ⊕-left-unit′-body u

In our experience, expression-level unfolding seems more commonly useful for end users than top-level unfolding; on the other hand, the clearest semantics for expression-level unfolding are stated in terms of top-level unfolding! Because one of our goals is to provide an account of unfolding that admits a reliable and precise mental model for programmers, it is desirable to include both top-level and expression-level unfolding in the surface language.

3 CONTROLLING UNFOLDING WITH EXTENSION TYPES

Having introduced our new surface language constructs for controlled unfolding in Section 2, we now describe how to elaborate these constructs into our dependently-typed core calculus. Again we proceed by example, deferring our formal descriptions of the elaboration algorithm to Section 4.

3.1 A core calculus with proposition symbols

Our core calculus parameterizes intensional Martin-Löf type theory (**MLTT**) [33] by a bounded meet semilattice of *proposition symbols* $p \in \mathbb{P}$, and adjoins to the type theory a new form of context extension and two new type formers $\{p\}\,A$ and $\{A \mid p \hookrightarrow M\}$ involving proposition symbols:

$$\begin{array}{llll} \textit{(contexts)} & \Gamma & ::= & \ldots \mid \Gamma, p \\ \textit{(types)} & A & ::= & \ldots \mid \{p\}\,A \mid \{A \mid p \hookrightarrow M\} \end{array}$$

The bounded meet semilattice structure on $\mathbb{P}$ closes proposition symbols under conjunction $\wedge$ and the true proposition $\top$, thereby partially ordering $\mathbb{P}$ by entailment $p \leq q$ ("p entails q") satisfying the usual principles of propositional logic. We say p is *true* if $\top$ entails p; the context extension Γ, p hypothesizes that p is true.

Remark 2. *Our proposition symbols are much more restricted than, and should not be confused with, other notions of proposition in type theory such as h-propositions [53, §3.3] or strict propositions [17]. In particular, unlike types, our proposition symbols have no associated proof terms.*

The type $\{p\}\,A$ is the dependent product "$\{\_ : p\} \to A$", *i.e.*, $\{p\}\,A$ is well-formed when A is a type under the hypothesis that p is true, and $f : \{p\}\,A$ when, given that p is true, we may

conclude $f : A$. The *extension type* $\{A \mid p \hookrightarrow a_p\}$ is well-formed when A is a type and $a_p : \{p\}\, A$; its elements $a : \{A \mid p \hookrightarrow a_p\}$ are terms $a : A$ satisfying the side condition that when p is true, we have $a = a_p : A$. We provide inference rules for the core calculus, including these connectives, in Section 4.1.

3.2 Elaborating controlled unfolding to our core calculus

Our surface language extends a generic surface language for dependent type theory with a new expression former **unfold** and several new declaration forms: ϑ **unfolds** $\kappa_1; \ldots; \kappa_n$ for controlled unfolding, **abbreviation** ϑ for transparent definitions, and **abstract** ϑ for opaque definitions. Elaboration transforms these surface-language declarations into core-language *signatures*, *i.e.* sequences of declarations over our core calculus of **MLTT** with proposition symbols.

Our signatures include the following declaration forms:

- **prop** $p \leq q$ introduces a fresh proposition symbol p such that p entails $q \in \mathbb{P}$;
- **prop** $p = q$ defines the proposition symbol p to be an abbreviation for $q \in \mathbb{P}$;
- **const** $\vartheta : A$ introduces a constant ϑ of type A.

We now revisit our examples from Section 2, illustrating how they are elaborated into our core calculus:

Plain definitions

Recall our unadorned definition of (+) from Section 2:

$(+) : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
$\mathsf{ze} + n = n$
$\mathsf{su}\ m + n = \mathsf{su}\ (m + n)$

We elaborate (+) into a sequence of declarations: first, we introduce a new proposition symbol Υ_+ corresponding to the proposition that "(+) unfolds." Next, we introduce a new definition $\delta_+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ satisfying the defining clauses of (+) above, under the (trivial) assumption of $\top$; finally, we introduce a new constant (+) involving the extension type of δ_+ along Υ_+.

prop $\Upsilon_+ \leq \top$

$\delta_+ : \{\top\}\, (m\, n : \mathbb{N}) \to \mathbb{N}$
$\delta_+\ \mathsf{ze}\ n = n$
$\delta_+\ (\mathsf{su}\ m)\ n = \mathsf{su}\ (\delta_+\ m\ n)$

const $(+) : \{\mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid \Upsilon_+ \hookrightarrow \delta_+\}$

Remark 3. *In a serious implementation, it would be simple to induce δ_+ to be pretty-printed as* (+) *in user-facing displays such as goals and error messages.*

Top-level unfolding

To understand why we have elaborated (+) in this way, let us examine how to elaborate top-level unfolding declarations (Section 2.1):

$(\oplus)$ **unfolds** $(+)$
$(\oplus) : \mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m + n)\ A$
$[\,] \oplus v = v$
$(a :: u) \oplus v = a :: (u \oplus v)$

To elaborate $(\oplus)$ **unfolds** $(+)$, we define the proposition symbol $\Upsilon_\oplus$ to entail Υ_+, capturing the idea that unfolding $(\oplus)$ always causes $(+)$ to unfold; in order to cause $(+)$ to unfold in the body of $(\oplus)$, we assume Υ_+ in the definition of $\delta_\oplus$. In full, we elaborate the definition of $(\oplus)$ as follows:

$$\textbf{prop}\ \Upsilon_\oplus \leq \Upsilon_+$$

$$\begin{aligned}
&\delta_\oplus : \{\Upsilon_+\}\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{vec}\ (m+n)\ A\\
&\delta_\oplus\ [\,]\ v = v\\
&\delta_\oplus\ (a :: u)\ v = a :: (\delta_\oplus\ u\ v)
\end{aligned}$$

$$\textbf{const}\ (\oplus) : \{\mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\ (m+n)\ A \mid \Upsilon_\oplus \hookrightarrow \delta_\oplus\}$$

Observe that the definition of $\delta_\oplus$ is well-typed because Υ_+ is true in its scope: thus the extension type of $(+)$ causes $\mathsf{ze} + n$ to be definitionally equal to $\delta_+\ \mathsf{ze}\ n$, which in turn is defined to be n. The constraint $\Upsilon_\oplus \hookrightarrow \delta_\oplus$ is well-typed because $\Upsilon_\oplus$ entails Υ_+.

If a definition ϑ unfolds multiple definitions $\kappa_1; \ldots; \kappa_n$, we define Υ_ϑ to entail (and define δ_ϑ to assume) the conjunction $\Upsilon_{\kappa_1} \wedge \cdots \wedge \Upsilon_{\kappa_n}$; if a definition ϑ unfolds no definitions, then Υ_ϑ entails (and δ_ϑ assumes) $\top$, as in our $(+)$ example.

Abbreviations

To elaborate the combination of the declarations **abbreviation** ϑ and ϑ **unfolds** $\kappa_1; \ldots; \kappa_n$ we define Υ_ϑ to *equal* the conjunction $\Upsilon_{\kappa_1} \wedge \cdots \wedge \Upsilon_{\kappa_n}$. For example, consider the following code from Section 2.3:

$$\begin{aligned}
&\textbf{abbreviation}\ \mathsf{map}\text{-}\oplus\\
&\mathsf{map}\text{-}\oplus\ \textbf{unfolds}\ \mathsf{map}; (\oplus)\\
&\mathsf{map}\text{-}\oplus : (f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{map}\ f\ (u \oplus v) \equiv \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v\\
&\mathsf{map}\text{-}\oplus\ f\ [\,]\ v = \mathsf{refl}\\
&\mathsf{map}\text{-}\oplus\ f\ (a :: u)\ v = \mathsf{cong}\ ((f\ a) :: -)\ (\mathsf{map}\text{-}\oplus\ f\ u\ v)
\end{aligned}$$

Let us write $\mathfrak{C}$ for the following type:

$$(f : A \to B)\ (u : \mathsf{vec}\ m\ A)\ (v : \mathsf{vec}\ n\ A) \to \mathsf{map}\ f\ (u \oplus v) \equiv \mathsf{map}\ f\ u \oplus \mathsf{map}\ f\ v$$

The above example is then elaborated as follows:

$$\textbf{prop}\ \Upsilon_{\mathsf{map}\text{-}\oplus} = \Upsilon_{\mathsf{map}} \wedge \Upsilon_\oplus$$

$$\begin{aligned}
&\delta_{\mathsf{map}\text{-}\oplus} : \{\Upsilon_{\mathsf{map}} \wedge \Upsilon_\oplus\}\ \mathfrak{C}\\
&\delta_{\mathsf{map}\text{-}\oplus}\ f\ [\,]\ v = \mathsf{refl}\\
&\delta_{\mathsf{map}\text{-}\oplus}\ f\ (a :: u)\ v = \mathsf{cong}\ ((f\ a) :: -)\ (\delta_{\mathsf{map}\text{-}\oplus}\ f\ u\ v)
\end{aligned}$$

$$\textbf{const}\ \mathsf{map}\text{-}\oplus : \{\mathfrak{C} \mid \Upsilon_{\mathsf{map}\text{-}\oplus} \hookrightarrow \delta_{\mathsf{map}\text{-}\oplus}\}$$

Expression-level unfolding

The elaboration of the expression-level unfolding construct **unfold** κ **in** M to our core calculus factors through the elaboration of expression-level unfolding to top-level unfolding as described in Section 2.5; we return to this in Section 4.3.

4 THE ELABORATION ALGORITHM

We now formally specify our mechanism for controlled unfolding by more precisely defining the elaboration algorithm sketched in the previous section, starting with a precise definition of the target of elaboration, our core calculus $\mathbf{TT}_{\mathbb{P}}$.

4.1 The core calculus $\mathbf{TT}_{\mathbb{P}}$

Our core calculus $\mathbf{TT}_{\mathbb{P}}$ is intensional Martin-Löf type theory (**MLTT**) [33] with dependent sums and products, a Tarski universe, *etc.*, extended with (1) a collection of proof-irrelevant proposition symbols, (2) dependent products over propositions, and (3) extension types for those propositions [38].

Remark 4. *We treat the features of* **MLTT** *and of our surface language somewhat generically; our elaboration algorithm can be applied on top of an existing bidirectional elaboration algorithm for type theory,* e.g. *those described by Dagand [11], Gratzer et al. [21], which may separately account for features such as implicit arguments or dependent pattern matching.*

In fact, $\mathbf{TT}_{\mathbb{P}}$ is actually a family of type theories parameterized by a bounded meet semilattice $(\mathbb{P}, \top, \wedge)$ whose underlying set $\mathbb{P}$ is the set of proposition symbols of $\mathbf{TT}_{\mathbb{P}}$; the semilattice structure on $\mathbb{P}$ axiomatizes the conjunctive fragment of propositional logic with $\wedge$ as conjunction, $\top$ as the true proposition, and $\leq$ as entailment (where $p \leq q$ is defined as $p \wedge q = p$), subject to the usual logical principles such as $p \wedge q \leq p$ and $p \wedge q \leq q$ and $p \leq \top$.

Remark 5. *The judgments of* $\mathbf{TT}_{\mathbb{P}}$ *are functorial in the choice of* $\mathbb{P}$*, in the sense that given any homomorphism* $f : \mathbb{P} \longrightarrow \mathbb{P}'$ *of bounded meet semilattices and any type or term in* $\mathbf{TT}_{\mathbb{P}}$ *over* $\mathbb{P}$*, we have an induced type/term in* $\mathbf{TT}_{\mathbb{P}'}$ *over* $\mathbb{P}'$*. In particular, we will use the fact that judgments of* $\mathbf{TT}_{\mathbb{P}}$ *are stable under adjoining new proposition symbols to* $\mathbb{P}$*.*

The language $\mathbf{TT}_{\mathbb{P}}$ augments ordinary **MLTT** with a new judgment $\Gamma \vdash p\ \mathit{true}$ (for $p \in \mathbb{P}$) and the corresponding context extension Γ, p (for $p \in \mathbb{P}$). The judgment $\Gamma \vdash p\ \mathit{true}$ states that the proposition p is true in context Γ, *i.e.*, the conjunction of the propositional hypotheses in Γ entails p while Γ, p extends Γ with the hypothesis that p is true.

$$\frac{\Gamma\ \mathit{ctx} \qquad p \in \mathbb{P}}{\Gamma, p\ \mathit{ctx}} \qquad \frac{p \in \mathbb{P}}{\Gamma, p \vdash p\ \mathit{true}} \qquad \frac{\Gamma, p \vdash \mathcal{J} \qquad \Gamma \vdash p\ \mathit{true}}{\Gamma \vdash \mathcal{J}}$$

$$\frac{}{\Gamma \vdash \top\ \mathit{true}} \qquad \frac{\Gamma \vdash p\ \mathit{true} \qquad \Gamma \vdash q\ \mathit{true}}{\Gamma \vdash p \wedge q\ \mathit{true}} \qquad \frac{\Gamma \vdash p\ \mathit{true} \qquad p \leq q}{\Gamma \vdash q\ \mathit{true}}$$

The dependent product $\{p\}\, A$ is defined as an ordinary dependent product:

$$\frac{\Gamma, p \vdash A\ \mathit{type}}{\Gamma \vdash \{p\}\, A\ \mathit{type}} \qquad \frac{\Gamma, p \vdash M : A}{\Gamma \vdash \langle p \rangle\, M : \{p\}\, A} \qquad \frac{\Gamma \vdash M : \{p\}\, A \qquad \Gamma \vdash p\ \mathit{true}}{\Gamma \vdash M \mathbin{@} p : A}$$

$$\frac{\Gamma, p \vdash M : A \qquad \Gamma \vdash p\ \mathit{true}}{\Gamma \vdash (\langle p \rangle\, M) \mathbin{@} p = M : A} \qquad \frac{\Gamma \vdash M : \{p\}\, A}{\Gamma \vdash M = \langle p \rangle\, (M \mathbin{@} p) : \{p\}\, A}$$

The remaining feature of $\mathbf{TT}_{\mathbb{P}}$ is the extension type $\{A \mid p \hookrightarrow a_p\}$. Given a proposition $p \in \mathbb{P}$ and an element a_p of A under the hypothesis p, the elements of $\{A \mid p \hookrightarrow a_p\}$ correspond to elements of A that equal a_p when p holds.

$$\frac{\Gamma \vdash A\ \mathit{type} \qquad \Gamma, p \vdash a_p : A}{\Gamma \vdash \{A \mid p \hookrightarrow a_p\}\ \mathit{type}} \qquad \frac{\begin{array}{c}\Gamma \vdash a : A \\ \Gamma, p \vdash a_p : A \\ \Gamma, p \vdash a = a_p : A\end{array}}{\Gamma \vdash \mathsf{in}_p\, a : \{A \mid p \hookrightarrow a_p\}} \qquad \frac{\Gamma \vdash a : \{A \mid p \hookrightarrow a_p\}}{\Gamma \vdash \mathsf{out}_p\, a : A}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{out}_p(\mathsf{in}_p\, a) = a : A} \qquad \frac{\Gamma \vdash a : \{A \mid p \hookrightarrow a_p\}}{\Gamma \vdash \mathsf{in}_p(\mathsf{out}_p\, a) = a : \{A \mid p \hookrightarrow a_p\}}$$

$$\frac{\Gamma \vdash p\ \mathit{true} \qquad \Gamma \vdash a : \{A \mid p \hookrightarrow a_p\}}{\Gamma \vdash \mathsf{out}_p\, a = a_p : A}$$

4.2 Signatures over $\mathbf{TT}_{\mathbb{P}}$

Our elaboration procedure takes as input a sequence of surface language definitions, and outputs a well-formed *signature*, a list of declarations over $\mathbf{TT}_{\mathbb{P}}$.

$$\begin{array}{llll} \textit{(sigs)} & \Sigma & ::= & \epsilon \mid \Sigma, D \\ \textit{(decls)} & D & ::= & \mathbf{const}\, x : A \mid \mathbf{prop}\, p \leq q \mid \mathbf{prop}\, p = q \end{array}$$

A signature is well-formed precisely when each declaration in Σ is well-formed relative to the earlier declarations in Σ. Our well-formedness judgment $\vdash \Sigma\ \mathit{sig} \longrightarrow \mathbb{P}, \Gamma$ computes from Σ the $\mathbf{TT}_{\mathbb{P}}$ context Γ and proposition semilattice $\mathbb{P}$ specified by Σ's **const** and **prop** declarations, respectively.

The rules for signature well-formedness are standard except for the $\mathbf{prop}\, p \leq q$ and $\mathbf{prop}\, p = q$ declarations, which extend $\mathbb{P}$ with a new element p satisfying $p \leq q$ or $p = q$ respectively. Recalling that our core calculus $\mathbf{TT}_{\mathbb{P}}$ is really a family of type theories parameterized by a semilattice $\mathbb{P}$, these declarations shift us between type theories, *e.g.*, from $\mathbf{TT}_{\mathbb{P}}$ to $\mathbf{TT}_{\mathbb{Q}}$, where $\mathbb{Q} = \mathbb{P}[p \leq q]$ is the minimal semilattice containing $\mathbb{P}$ and an element p satisfying $p \leq q$. This shifting between theories is justified by Remark 5.

$$\frac{}{\vdash \epsilon\ \mathit{sig} \longrightarrow \{\top\}, \cdot} \qquad \frac{\vdash \Sigma\ \mathit{sig} \longrightarrow \mathbb{P}, \Gamma \qquad \Gamma \vdash_{\mathbf{TT}_{\mathbb{P}}} A\ \mathit{type}}{\vdash (\Sigma,\ \mathbf{const}\, x : A)\ \mathit{sig} \longrightarrow \mathbb{P}, (\Gamma, x : A)}$$

$$\frac{\vdash \Sigma\ \mathit{sig} \longrightarrow \mathbb{P}, \Gamma \qquad q \in \mathbb{P}}{\vdash (\Sigma,\ \mathbf{prop}\, p \leq q)\ \mathit{sig} \longrightarrow \mathbb{P}[p \leq q], \Gamma \qquad \vdash (\Sigma,\ \mathbf{prop}\, p = q)\ \mathit{sig} \longrightarrow \mathbb{P}[p = q], \Gamma}$$

4.3 Bidirectional elaboration

We adopt a bidirectional elaboration algorithm which mirrors bidirectional type-checking algorithms [10, 36]. The top-level elaboration judgment $\Sigma \vdash \vec{S} \rightsquigarrow \Sigma'$ takes as input the current well-formed signature Σ and a list of surface-level definitions $\vec{S}$ and outputs a new well-formed signature Σ'.

We define $\Sigma \vdash \vec{S} \rightsquigarrow \Sigma'$ in terms of three auxiliary judgments for elaborating surface-language types and terms; in the bidirectional style, we divide term elaboration into a checking judgment $\Sigma; \Gamma \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma', M$ taking a core type as input, and a synthesis judgment $\Sigma; \Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma', M$ producing a core type as output. All three judgments take as input a signature Σ and a context (telescope) over Σ, and output a new signature along with a core type or term.

We represent a surface-level definition S as a tuple:

$$(\mathbf{def}\ \vartheta : \mathsf{A}, \mathit{abbrv?}, \mathit{abstr?}, [\kappa_1, \ldots \kappa_n], \mathsf{e})$$

In this expression, ϑ is the name of the definiendum, $\mathtt{A}$ is the surface-level type of the definition, *abbrv?* and *abstr?* are flags governing whether ϑ is an **abbreviation** (resp., is **abstract**), $[\kappa_1, \ldots, \kappa_n]$ are the names of the definitions that ϑ unfolds, and $\mathtt{e}$ is the surface-level definiens.

The elaboration judgment elaborates each surface definition in sequence:

$$\frac{\begin{array}{l}\Sigma \vdash \vec{S} \rightsquigarrow \Sigma_1 \\ \Sigma_1; \cdot \vdash \mathtt{A} \Leftarrow \mathit{type} \rightsquigarrow \Sigma_2, A \\ \Sigma_2; \bigwedge_{i \le n} \Upsilon_{\kappa_i} \vdash \mathtt{e} \Leftarrow A \rightsquigarrow \Sigma_3, M \\ \textbf{let}\ p \coloneqq \textbf{if}\ \mathit{abstr?}\ \textbf{then}\ \mathit{gensym}\,() \ \textbf{else}\ \Upsilon_\vartheta \\ \textbf{let}\ \boxslash \coloneqq \textbf{if}\ \mathit{abbrv?}\ \textbf{then}\ (=)\ \textbf{else}\ (\le) \\ \textbf{let}\ \Sigma_4 \coloneqq \Sigma_3, \textbf{prop}\, p \boxslash \bigwedge_{i \le n} \Upsilon_{\kappa_i}, \textbf{const}\, \vartheta : \{A \mid p \hookrightarrow M\}\end{array}}{\Sigma \vdash \vec{S}, (\textbf{def}\ \vartheta : \mathtt{A}, \mathit{abbrv?}, \mathit{abstr?}, [\kappa_1, \ldots, \kappa_n], \mathtt{e}) \rightsquigarrow \Sigma_4}$$

Remark 6. *When a definition is marked* **abstract***, the name of the unfolding proposition is generated fresh so that it cannot be accessed by any future* **unfold** *declaration. Conversely, when a definition is marked as an* **abbreviation***, its unfolding proposition is defined to be equivalent to the conjunction of its dependencies rather than merely entailing its dependencies.*

The rules for term and type elaboration are largely standard: for instance, we elaborate a surface dependent product to a core dependent product by recursively elaborating the first and second components. We single out two cases below: the boundary between checking and synthesis, and the expression-level **unfold**.

$$\frac{\Sigma; \Gamma \vdash \mathtt{e} \Rightarrow A \rightsquigarrow \Sigma_1; M \qquad \Sigma_1; \Gamma \vdash \mathsf{conv}\, A\, B}{\Sigma; \Gamma \vdash \mathtt{e} \Leftarrow B \rightsquigarrow \Sigma_1; M} \qquad \frac{\begin{array}{l}\Sigma; \Gamma, \Upsilon_\vartheta \vdash \mathtt{e} \Leftarrow A \rightsquigarrow \Sigma_1; M \qquad \textbf{let}\ \chi \coloneqq \mathit{gensym}\,() \\ \textbf{let}\ \Sigma_2 \coloneqq \Sigma_1, \textbf{const}\, \chi : \textstyle\prod_\Gamma \{A \mid \Upsilon_\vartheta \hookrightarrow M\}\end{array}}{\Sigma; \Gamma \vdash \textbf{unfold}\ \vartheta\ \textbf{in}\ \mathtt{e} \Leftarrow A \rightsquigarrow \Sigma_2; \mathsf{out}_{\Upsilon_\vartheta}\, \chi[\Gamma]}$$

The first rule states that a term synthesizing a type A can be checked against a type B provided that A and B are definitionally equal; in order to implement this rule algorithmically, we need definitional equality to be decidable. Additionally, our (omitted) type-directed elaboration rules are only well-defined if type constructors are injective up to definitional equality, *e.g.*, $A \to B = C \to D$ if and only if $A = C$ and $B = D$.

Elaborating expression-level unfolding requires the ability to hoist a type to the top level by iterating dependent products over its context, an operation notated $\prod_\Gamma$ above. Because Γ can hypothesize (the truth of) propositions, this operation relies crucially on the presence of dependent products $\{p\}\, A$.

5 CASE STUDY: AN IMPLEMENTATION IN `cooltt`

We have implemented our approach to controlled unfolding in the experimental **`cooltt`** proof assistant [37]; **`cooltt`** is an implementation of *cartesian cubical type theory* [2], a computational version of homotopy type theory whose syntactic metatheory is particularly well understood [24, 39, 41]. The existing support for partial elements and extension types made **`cooltt`** particularly hospitable for experimentation with elaborating controlled unfolding to extension types. The following example illustrates the use of controlled unfolding in **`cooltt`**, where path $A\ x\ y$ is the cubical notion of propositional equality ($x \equiv y$):

```
def + : ℕ → ℕ → ℕ :=
  elim
  | zero ⇒ n ⇒ n
  | suc {_ ⇒ ih} ⇒ n ⇒ suc {ih n}
```

```
unfold +
def +0L (x : ℕ) : path ℕ {+ 0 x} x ≔
  i ⇒ x

def +0R : (x : ℕ) → path ℕ {+ x 0} x ≔
  elim
  | zero ⇒ +0L 0
  | suc {x ⇒ ih} ⇒
    equation ℕ
      | + 0 {suc y}   =[+0L {suc y}]
      | suc {+ x 0}   =[i ⇒ suc {ih i}]
      | suc x
```

This example follows a common pattern: we prove basic computational laws (+0L) by unfolding a definition, and then in subsequent results (+0R) use these lemmas abstractly rather than unfolding. Doing so controls the size and readability of proof goals, and explicitly demarcates which parts of the library depend on the definitional behavior of a given function.

We have also implemented the derived forms for expression-level unfolding:

```
def two : ℕ ≔ + 1 1
def thm : path ℕ two 2 ≔ unfold two + in i ⇒ 2
def thm-is-refl : path-p {i ⇒ path ℕ two {thm i}} {i ⇒ two} thm ≔
  i j ⇒ unfold two + in 2
def thm-is-refl′ : path {path ℕ two 2} {i ⇒ unfold two + in two} thm ≔
  i j ⇒ unfold two + in 2
```

The third and fourth declarations above illustrate two strategies in `cooltt` for dealing with a dependent type whose well-formedness depends on an unfolding; in thm-is-refl we use a dependent path type but only unfold in the definiens, whereas in thm-is-refl′ we use a non-dependent path type but must unfold in both the definiens and in its type.

Our `cooltt` implementation deviates in a few respects from the presentation in this paper: in particular, the propositions Υ_κ are represented by abstract elements $i_\kappa : \mathbb{I}$ of the interval via the embedding $\mathbb{I} \hookrightarrow \mathbb{F}$ sending i to $(i =_{\mathbb{I}} 1)$.

`cooltt` utilizes a standalone library to compute entailment of cofibrations called Kado [23], created by Kuen-Bang Hou (Favonia). To support our experiment, Favonia modified Kado to support inequalities of dimension variables $i \leq_{\mathbb{I}} j$ in addition to the cofibrations needed for `cooltt`'s core theory. As a result, the modifications to `cooltt` were quite modest. After the changes to Kado—which could in principle be reused in any proof assistant for the same purpose—the entire change required only a net increase of 996 lines of OCaml code.

6 THE METATHEORY OF $\mathbf{TT}_{\mathbb{P}}$

In Section 4 we described an algorithm elaborating a surface language with controlled unfolding to $\mathbf{TT}_{\mathbb{P}}$. In order to actually execute our algorithm, it is necessary to decide the definitional equality of types in $\mathbf{TT}_{\mathbb{P}}$; as is often the case in type theory, type dependency ensures that deciding equality for types also requires us to decide the equality of terms. In order to implement our elaboration algorithm, we therefore prove a *normalization* result for $\mathbf{TT}_{\mathbb{P}}$.

At its heart, a normalization algorithm is a computable bijection between equivalence classes of terms up to definitional equality and a collection of normal forms. By ensuring that the equality of normal forms is evidently decidable, this yields an effective decision procedure for definitional

equality. In our case, we attack normalization through a *synthetic* and *semantic* approach to normalization by evaluation called synthetic Tait computability [39–42] or STC.

Neutral forms for $\mathbf{TT}_{\mathbb{P}}$. The semantic analysis of normalization by evaluation rests on the observation of Fiore [14] that normal forms, though not stable under arbitrary substitutions, are nonetheless stable under *renamings* — substitutions that replace variables with variables (not necessarily injective). Therefore, decisive aspects of the normalization algorithm can be expressed *internally* to a topos of variable sets (presheaves) over the category of contexts and renamings; in order to instrument the semantic normalization algorithm with its proof of correctness, one passes to a larger topos obtained from the former by gluing. Synthetic Tait computability then instantiates the standard topos model of Martin-Löf type theory to substantially simplify various details that would otherwise be exceedingly tedious by means of a form of higher-order abstract syntax.

This appealingly simple story for normalization is substantially complicated by the boundary law for extension types:

$$\frac{\Gamma \vdash p\ \mathit{true} \qquad \Gamma, p \vdash a_p : A \qquad \Gamma \vdash a : \{A \mid p \hookrightarrow a_p\}}{\Gamma \vdash \mathsf{out}_p\, a = a_p : A}$$

When defining normal forms for $\mathbf{TT}_{\mathbb{P}}$, we might naively add a neutral form $\mathbf{out}_p$ to represent out_p. In order to ensure that normal and neutral forms correspond bijectively with equivalence classes of terms, however, we should only allow $\mathbf{out}_p$ to be applied in a context where p is not true; if p were true, $\mathsf{out}_p\, a$ is already represented by the normal form for a_p.

A similar problem arises in the context of cubical type theory [2, 9] where some equalities apply precisely when two *dimensions* coincide. The same problem arises: either renamings must exclude substitutions that identify two dimension terms, or neutral forms will not be stable under renamings. In their proof of normalization for cubical type theory, Sterling and Angiuli [41] refined neutral forms to account for this tension by introducing *stabilized neutrals*. Rather than cutting down on renamings, they expand the class of neutrals by allowing "bad" neutrals akin to $\mathbf{out}_p\, e$ in a context where p is true. They then associate each neutral form with a *frontier of instability*: a proposition that becomes true when the neutral is no longer "stuck". Crucially, although well-behaved neutrals may not be stable under renamings, the frontier of instability *is* stable, and can therefore be incorporated into the internal language.

We adapt Sterling and Angiuli's stabilized neutrals to the simplified setting of $\mathbf{TT}_{\mathbb{P}}$ and establish its normalization theorem. In so doing, we refine the approach of *op. cit.* to obtain a fully constructive[3] normalization proof. We also carefully spell out the details of the universe in the normalization model, correcting an oversight in an earlier revision of Sterling's dissertation [39].

6.1 Type theories as categories with representable maps

While any number of logical frameworks are available (generalized algebraic theories [7], essentially algebraic theories [16], locally cartesian closed categories [20], *etc.*), Uemura's *categories with representable maps* [51, 52] are particularly attractive because they express exactly the binding and dependency structure needed for type theory: a second-order version of generalized algebraic theories.

Definition 7. *A* ***category with representable maps (CwR)*** $\mathcal{C}$ *is a finitely complete category equipped with a pullback-stable class of* ***representable maps*** $\mathcal{R} \subseteq \mathsf{Arr}(\mathcal{C})$ *such that pullback along* $f \in \mathcal{R}$ *has a right adjoint (dependent product along* f*).*

[3] By *constructive*, we mean something that can be carried out in an elementary topos with a natural numbers object.

Definition 8. *A **morphism of CwRs** is a functor between the underlying categories that preserves finite limits, representability of maps, and dependent products along representable maps.*

Definition 9. *CwRs, morphisms between them, and natural isomorphisms assemble into a* $(2,1)$*-category* $\mathbf{CwR}$.

Uemura's logical framework axiomatizes the category of judgments of $\mathbf{TT}_{\mathbb{P}}$ as a particular category with representable maps $\mathbb{T}$. The finite limit structure of $\mathbb{T}$ encodes substitution as well as equality judgments, while the class of representable maps carves out those judgments that may be hypothesized. Uemura [52] develops a syntactic method for presenting a CwR as a signature within a variant of extensional type theory, which he has rephrased in terms of *second order generalized algebraic theories* in his doctoral dissertation [51]. Although we will use the type-theoretic presentation for convenience, the difference between these two accounts is only superficial.

Each judgment of $\mathbf{TT}_{\mathbb{P}}$ is rendered as a (dependent) sort while operators are modeled by elements of the given sorts. In order to record whether a given judgment may be hypothesized, the sorts of the type theory are stratified by *meta*-sorts $\star \subseteq \square$ where $A : \star$ signifies that A is a representable sort (*i.e.* a context-former) and can be hypothesized, whereas $B : \square$ cannot parameterize a framework-level dependent product.

Proposition 10. *Let* $\mathbb{T}$ *be the free category with representable maps generated by a given logical framework signature; then the groupoid of CwR functors* $\hom_{\mathbf{CwR}}(\mathbb{T}, \mathcal{E})$ *is equivalent to the groupoid of interpretations of the signature within* $\mathcal{E}$.

We will often refer to a category with representable maps $\mathbb{T}$ as a type theory; indeed, as the category of judgments of a given type theory, $\mathbb{T}$ is a suitable invariant replacement for it.

Proposition 10 describes the universal property of a type theory generated by a given signature in a logical framework. Type theories *qua* CwRs thus give rise to a form of functorial semantics in which algebras (interpretations) arrange into a *groupoid* of CwR functors $\hom_{\mathbf{CwR}}(\mathbb{T}, \mathcal{E})$.

This is an appropriate setting for studying the syntax of type theory, but it is somewhat inappropriate for studying the *semantics* of type theory—in which one expects models to correspond to structured CwFs [13] or natural models [5], which themselves arrange into a (2,1)-category. The second notion of functorial semantics, developed by Uemura in his doctoral dissertation [51], is a generalization of the theory of CwFs and pseudo-morphisms between them [8, 35].

Note that we may always regard a presheaf category $\mathbf{Pr}\, C$ as a CwR with the representable maps being representable natural transformations, *i.e.* families of presheaves whose fibers at representables are representable [5].

Definition 11. *A **model** of a type theory* $\mathbb{T}$ *is a category* $\mathbf{M}_\diamond$ *together with a CwR functor* $\mathbf{M} : \mathbb{T} \longrightarrow \mathbf{Pr}\, \mathbf{M}_\diamond$.

Models are arranged into a (2,1)-category $\mathbf{Mod}\, \mathbb{T}$ (see Appendix A). Essentially, a morphism of models $\mathbf{M} \longrightarrow \mathbf{N}$ is given by a functor $\alpha_\diamond : \mathbf{M}_\diamond \longrightarrow \mathbf{N}_\diamond$ together with a natural transformation $\mathbf{M} \longrightarrow \alpha^* \mathbf{N} \in \hom_{\mathbf{CwR}}(\mathbb{T}, \mathbf{Pr}\, \mathbf{M}_\diamond)$ that preserves context extensions up to isomorphism; an isomorphism between morphisms of models is a natural isomorphism between the underlying functors satisfying an additional property.

For each CwR $\mathbb{T}$, Uemura has shown the following theorem:

Proposition 12. *The (2,1)-category of models* $\mathbf{Mod}\, \mathbb{T}$ *has a bi-initial object* $\mathbf{I}$*, whose category of contexts* $\mathbf{I}_\diamond$ *is the smallest full subcategory of* $\mathbb{T}$ *closed under the terminal object and pullbacks along representable maps.*

Remark 13. *If one takes* $\mathbb{T}$ *to be* e.g.*, Martin-Löf type theory, the bi-initial model* $\mathbf{I}$ *can be realized by the familiar initial CwF built from the category of contexts.*

6.2 Encoding $\mathbf{TT}_{\mathbb{P}}$ in the logical framework

We begin by defining the signature for a category with representable maps $\mathbb{T}_0$ containing exactly the bare judgmental structure of $\mathbf{TT}_{\mathbb{P}}$, namely the propositions and the judgments for types and terms. In our signature, we make liberal use of the Agda-style notation for implicit arguments. As always, p ranges over $\mathbb{P}$.

$\langle p \rangle : \star$
$\mathsf{tp} : \square$
$\mathsf{tm} : \mathsf{tp} \Longrightarrow \star$
$\_ : \{u, v : \langle p \rangle\} \Longrightarrow u = v$
$\_ : \left\{\_ : \left\langle \bigwedge_{i<n} p_i \right\rangle\right\} \Longrightarrow \langle p_k \rangle$
$\_ : \{\_ : \langle p_i \rangle, \dots\} \Longrightarrow \left\langle \bigwedge_{i<n} p_i \right\rangle$

Note that already this signature encodes the necessary theory of propositions for $\mathbf{TT}_{\mathbb{P}}$. For instance, if $p \leq q$ in $\mathbb{P}$ then a combination of the final two implications in the signature implies $\langle p \rangle \to \langle q \rangle$. We next extend the above to include the type formers of $\mathbf{TT}_{\mathbb{P}}$, writing $\mathbb{T}$ for the CwR generated by the full signature.

Notation 14. *Given $X : \{\_ : \langle p \rangle\} \Longrightarrow \square$, we will write $\{p\}\, X$ to further abbreviate the Agda-style implicit function space $\{\_ : \langle p \rangle\} \to X$. Note that $\{p\}\, X$ still associates to the right and so $\{p\} A \to B$ signifies $\{p\}(A \to B)$.*

For instance, the following constants specify the rules of extension types given in Section 4.1:

$\mathsf{ext}_p : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A) \Longrightarrow \mathsf{tp}$
$\mathsf{in}_p : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A)\ (u : \mathsf{tm}\, A)\ \{\_ : \{p\}\, u = a\} \Longrightarrow \mathsf{tm}\, (\mathsf{ext}_p\, A\, a)$
$\mathsf{out}_p : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A)\ (u : \mathsf{tm}\, (\mathsf{ext}_p\, A\, a)) \Longrightarrow \mathsf{tm}\, A$
$\_ : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A)\ (u : \mathsf{tm}\, (\mathsf{ext}_p\, A\, a))\ \{\_ : \langle p \rangle\} \Longrightarrow \mathsf{out}_p\, A\, a\, u = a$
$\_ : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A)\ (u : \mathsf{tm}\, A)\ \{\_ : \{p\}\, u = a\} \Longrightarrow \mathsf{out}_p\, A\, a\, (\mathsf{in}_p\, A\, a\, u) = u$
$\_ : (A : \mathsf{tp})\ (a : \{p\}\, \mathsf{tm}\, A)\ (u : \mathsf{tm}\, (\mathsf{ext}_p\, A\, a)) \Longrightarrow \mathsf{in}_p\, A\, a\, (\mathsf{out}_p\, A\, a\, u) = u$

The full list of non-standard constants is specified in Figure 1. Once the signature is complete, we obtain from Uemura's framework a category with representable maps $\mathbb{T}$ together with a bi-initial model $\mathbf{I}$.

6.3 The atomic figure shape and its universal property

For each context Γ and type $\Gamma \vdash A\ \mathit{type}$, it is possible to axiomatize the normal forms of type A; unfortunately, this assignment of sets of normal forms does not immediately extend to a presheaf on the category of contexts $\mathbf{I}_\diamond$, precisely because normal forms are not *a priori* closed under substitution! In fact, closing normal forms under substitution is the purpose of normalization, so we are not able to assume it beforehand.

Normal forms *are*, however, closed under substitutions of variables for variables (often called *structural renamings*), and in our case we shall be able to close them additionally under the "phase transitions" $\Gamma, \langle p \rangle \longrightarrow \Gamma, \langle q \rangle$ when $\Gamma, p \vdash q\ \mathit{true}$ is derivable. We shall refer to these substitutions as ***atomic substitutions***, and we wish to organize them into a category.

It is possible to inductively define a category of "atomic contexts" whose objects are those of $\mathbf{I}_\diamond$ and whose morphisms are atomic substitutions, but this construction obscures a beautiful and simple (2,1)-categorical universal property first exposed by Bocquet et al. [6] that leads to a more modular proof. To explicate this universal property, first note that the theory $\mathbb{T}_0$ axiomatizes exactly the structure of variables and phase transitions, and that the initial model $\mathbf{I}$ of $\mathbb{T}$ is, by restriction along $\mathbb{T}_0 \longrightarrow \mathbb{T}$, also a model of $\mathbb{T}_0$.

$$\mathsf{tp} : \square$$
$$\mathsf{tm} : \mathsf{tp} \Longrightarrow \star$$

$$\langle p \rangle : \star$$
$$\_ : \{u, v : \langle p \rangle\} \Longrightarrow u = v$$
$$\_ : \left\{\_ : \left\langle \bigwedge_{i<n} p_i \right\rangle\right\} \Longrightarrow \langle p_k \rangle$$
$$\_ : \{\_ : \langle p_i \rangle, \dots\} \Longrightarrow \left\langle \bigwedge_{i<n} p_i \right\rangle$$

$$\mathsf{ext}_p : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A) \Longrightarrow \mathsf{tp}$$
$$\mathsf{in}_p : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A)\,(u : \mathsf{tm}\,A)\,\{\_ : \{p\}\,u = a\} \Longrightarrow \mathsf{tm}\,(\mathsf{ext}_p\,A\,a)$$
$$\mathsf{out}_p : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A)\,(u : \mathsf{tm}\,(\mathsf{ext}_p\,A\,a)) \Longrightarrow \mathsf{tm}\,A$$
$$\_ : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A)\,(u : \mathsf{tm}\,(\mathsf{ext}_p\,A\,a))\,\{\_ : \langle p \rangle\} \Longrightarrow \mathsf{out}_p\,A\,a\,u = a$$
$$\_ : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A)\,(u : \mathsf{tm}\,A)\,\{\_ : \{p\}\,u = a\} \Longrightarrow \mathsf{out}_p\,A\,a\,(\mathsf{in}_p\,A\,a\,u) = u$$
$$\_ : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A)\,(u : \mathsf{tm}\,(\mathsf{ext}_p\,A\,a)) \Longrightarrow \mathsf{in}_p\,A\,a\,(\mathsf{out}_p\,A\,a\,u) = u$$

$$\mathsf{part}_p : (A : \{p\}\,\mathsf{tp}) \Longrightarrow \mathsf{tp}$$
$$\mathsf{lam}_p : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A) \Longrightarrow \mathsf{tm}\,(\mathsf{part}_p\,A)$$
$$\mathsf{app}_p : (A : \mathsf{tp})\,(u : \mathsf{tm}\,(\mathsf{part}_p\,A)) \Longrightarrow \{p\}\,\mathsf{tm}\,A$$
$$\_ : (A : \mathsf{tp})\,(a : \{p\}\,\mathsf{tm}\,A) \Longrightarrow \mathsf{app}_p\,A\,(\mathsf{lam}_p\,A\,a) = a$$
$$\_ : (A : \mathsf{tp})\,(a : \mathsf{tm}\,(\mathsf{part}_p\,A)) \Longrightarrow \mathsf{lam}_p\,A\,(\mathsf{app}_p\,A\,a) = a$$

Fig. 1. The non-standard aspects of the LF signature for $\mathbf{TT}_{\mathbb{P}}$.

Definition 15. *An **atomic substitution model** over a fixed $\mathbb{T}$-model $\mathbf{M}$ is given by a model $\mathbf{A}$ of the bare judgmental theory $\mathbb{T}_0$, together with a morphism of models $\alpha\colon \mathbf{A} \longrightarrow \mathbf{M}$ in $\mathbf{Mod}\,\mathbb{T}_0$ such that $\alpha_{\mathsf{tp}}\colon \mathbf{A}(\mathsf{tp}) \longrightarrow \alpha^*(\mathbf{M}(\mathsf{tp})) \in \mathbf{Pr}\,\mathbf{A}_\diamond$ is an isomorphism.*

Atomic substitution model over $\mathbf{M}$ arrange themselves into a (2,1)-category, a full subcategory of $\mathbf{Mod}\,\mathbb{T}_0 \downarrow \mathbf{M}$. The following result is due to Bocquet et al. [6].

Proposition 16. *The bi-initial atomic substitution model $\left(\mathbf{A}, \alpha\colon \mathbf{A} \longrightarrow \mathbf{I}\right)$ over $\mathbf{I}$ exists.*

When $\left(\mathbf{A}, \alpha\colon \mathbf{A} \longrightarrow \mathbf{I}\right)$ is the bi-initial atomic substitution model over $\mathbf{I}$ as in Proposition 16, we shall refer to an object $\Gamma \in \mathbf{A}$ as an *atomic context* and a morphism $\gamma\colon \Delta \longrightarrow \Gamma$ in $\mathbf{A}$ as an *atomic substitution*. We shall assume without loss of generality that $\mathbf{A}(\mathsf{tp}) = \alpha^*\mathbf{I}(\mathsf{tp})$ so that the component α_{tp} is the identity map.

6.4 Computability spaces by gluing along the atomic figure shape

We shall use the bi-initial atomic substitution model over $\mathbf{I}$ as a ***figure shape*** in the sense of Sterling [39, §4.3] to instantiate synthetic Tait computability. Here we transition into the 2-category of Grothendieck topoi, geometric morphisms, and geometric transformations, guided by a ***phase distinction*** between "object-space" and "meta-space" [39];[4] object-space refers to the object language embodied in the model $\mathbf{I}$, whereas meta-space refers to the metalanguage embodied in the model $\mathbf{A}$. Later on, we will construct a *glued* topos in which we may speak of constructs that have extent in

[4]In his doctoral dissertation, Sterling referred to "object-space" and "meta-space" as *syntactic* and *semantic* respectively [39]. However, there are compelling reasons to consider object-space more semantic than meta-space (in which various admissibilities hold that will not be preserved by homomorphisms of models), so we have changed terminology to avoid confusion.

both object-space and meta-space. We follow Anel and Joyal [1] and Vickers [54] in emphasizing the distinction between a topos $\mathbf{X}$ and the category of sheaves $\mathbf{Sh}\,\mathbf{X}$ presenting it:

Definition 17. *We denote by* $\mathbf{I}$ *and* $\mathbf{A}$ *the* ***object-space*** *and* ***meta-space topoi*** *respectively, with underlying categories of sheaves* $\mathbf{Sh}\,\mathbf{I} = \mathbf{Pr}\,\mathbf{I}_\diamond$ *and* $\mathbf{Sh}\,\mathbf{A} = \mathbf{Pr}\,\mathbf{A}_\diamond$.

Definition 18. *The functor* $\alpha_\diamond : \mathbf{A}_\diamond \longrightarrow \mathbf{I}_\diamond$ *gives rise under precomposition to a continuous and cocontinuous functor* $\mathbf{Pr}\,\mathbf{I}_\diamond \longrightarrow \mathbf{Pr}\,\mathbf{A}_\diamond$*, that shall serve as the inverse image part of an (essential) geometric morphism* $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$ *named the* ***atomic figure shape****.*

That $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$ is *essential* means that its inverse image $\alpha^* : \mathbf{Sh}\,\mathbf{I} \longrightarrow \mathbf{Sh}\,\mathbf{A}$ has a left adjoint $\alpha_! : \mathbf{Sh}\,\mathbf{A} \longrightarrow \mathbf{Sh}\,\mathbf{I}$; from the point of view of presheaves, this is precisely the Yoneda extension of $\alpha_\diamond : \mathbf{A}_\diamond \longrightarrow \mathbf{I}_\diamond$ as depicted below:

$$\begin{array}{ccc} \mathbf{A}_\diamond & \xrightarrow{\alpha_\diamond} & \mathbf{I}_\diamond \\ {\scriptstyle y_{\mathbf{A}_\diamond}}\downarrow & & \downarrow{\scriptstyle y_{\mathbf{I}_\diamond}} \\ \mathbf{Pr}\,\mathbf{A}_\diamond & \xrightarrow[\alpha_!]{} & \mathbf{Pr}\,\mathbf{I}_\diamond \end{array}$$

Definition 19. *We denote by* $\mathbf{G}$ *the* closed mapping cylinder *[26] of the geometric morphism* $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$*; in other words,* $\mathbf{Sh}\,\mathbf{G}$ *is the comma category* $\mathbf{Sh}\,\mathbf{A} \downarrow \alpha^*$*. We will write* $\boldsymbol{j} : \mathbf{I} \hookrightarrow \mathbf{G}$ *and* $\boldsymbol{i} : \mathbf{A} \hookrightarrow \mathbf{G}$ *for the open and closed subtopos immersions.*

Following Sterling [40], we shall refer to a sheaf on $\mathbf{G}$ as a ***computability space***. A computability space $X \in \mathbf{Sh}\,\mathbf{G}$ is then identified with a family $\pi_X : \boldsymbol{i}^* X \longrightarrow \alpha^* \boldsymbol{j}^* X$ in $\mathbf{Sh}\,\mathbf{A}$. Because the assignment π_X is natural in computability spaces X, it corresponds to a 2-cell $\pi : \boldsymbol{j} \circ \alpha \longrightarrow \boldsymbol{i}$ in the 2-category of Grothendieck topoi. The universal property of $\mathbf{G}$ is then expressed by the fact that $\pi : \boldsymbol{j} \circ \alpha \longrightarrow \boldsymbol{i}$ is a *co-comma* cell in the 2-category of Grothendieck topoi:

$$\begin{array}{ccc} \mathbf{A} & \xrightarrow{\alpha} & \mathbf{I} \\ \| & \Uparrow \pi & \downarrow{\scriptstyle \boldsymbol{j}} \\ \mathbf{A} & \xrightarrow[\boldsymbol{i}]{} & \mathbf{G} \end{array}$$

Remark 20 (Relation to Kripke computability predicates)**.** *Unraveling Definition 19, a computability space is precisely a* family *X' of presheaves on $\mathbf{A}_\diamond$ indexed in the restriction of a given presheaf X on $\mathbf{I}_\diamond$ along $\alpha_\diamond : \mathbf{A}_\diamond \longrightarrow \mathbf{I}_\diamond$. When the family X' is valued in* subterminal *presheaves and the base X is representable, we have precisely the classical notion of a Kripke computability predicate [27]; a* computability space *in our sense is then a generalised, proof-relevant version of a Kripke computability predicate.*

6.4.1 Reflection of object and meta-space. By definition, the inverse image functors $\boldsymbol{j}^*, \boldsymbol{i}^*$ have fully faithful right adjoints $\boldsymbol{j}_* : \mathbf{Sh}\,\mathbf{I} \hookrightarrow \mathbf{Sh}\,\mathbf{G}$ and $\boldsymbol{i}_* : \mathbf{Sh}\,\mathbf{A} \hookrightarrow \mathbf{Sh}\,\mathbf{G}$ respectively. These are computed as follows:

$$\boldsymbol{j}_* E = \left(E, 1_{\alpha^* E} : \alpha^* E \longrightarrow \alpha^* E\right)$$
$$\boldsymbol{i}_* A = \left(\mathbf{1}_{\mathbf{Sh}\,\mathbf{I}}, !_A : A \longrightarrow \mathbf{1}_{\mathbf{Sh}\,\mathbf{A}} \cong \alpha^* \mathbf{1}_{\mathbf{Sh}\,\mathbf{I}}\right)$$

Thus the adjunctions $\boldsymbol{j}^* \dashv \boldsymbol{j}_*$ and $\boldsymbol{i}^* \dashv \boldsymbol{i}_*$ exhibit $\mathbf{Sh}\,\mathbf{I}$ and $\mathbf{Sh}\,\mathbf{A}$ as *reflective* subcategories of $\mathbf{Sh}\,\mathbf{G}$.

(1) The essential image of the reflective embedding $\boldsymbol{j}_* \colon \mathbf{Sh\,I} \hookrightarrow \mathbf{Sh\,G}$ is spanned by computability spaces X for which $\pi_X \colon \boldsymbol{i}^*X \longrightarrow \alpha^*\boldsymbol{j}^*X$ is an isomorphism, *i.e.* so that π_X is terminal in the slice $\mathbf{Sh\,A} \downarrow \alpha^*\boldsymbol{j}^*X$.
(2) The essential of image of the reflective embedding $\boldsymbol{i}_* \colon \mathbf{Sh\,A} \hookrightarrow \mathbf{Sh\,G}$ is spanned by computability spaces X such that $\boldsymbol{j}^*X$ is terminal in $\mathbf{Sh\,I}$.

Definition 21 (Vocabulary for reflective subcategories)**.** *When a computability space lies in the essential image of $\boldsymbol{j}_* \colon \mathbf{Sh\,I} \hookrightarrow \mathbf{Sh\,G}$, we shall refer to it as **lying in object-space**. Likewise, when a computabiltiy space lies in the essential image of $\boldsymbol{i}_* \colon \mathbf{Sh\,A} \hookrightarrow \mathbf{Sh\,G}$, we shall say that it **lies in meta-space**.*

6.4.2 Coreflection of object and meta-space. Both the open and closed immersions are *essential* morphisms of topoi, in the sense that we have additional (necessarily fully faithful) left adjoints $\boldsymbol{j}_! \dashv \boldsymbol{j}^* \colon \mathbf{Sh\,G} \to \mathbf{Sh\,I}$ and $\boldsymbol{i}_! \dashv \boldsymbol{i}^* \colon \mathbf{Sh\,G} \to \mathbf{Sh\,A}$ that are computed as follows:

$$\boldsymbol{j}_!E = \left(E, !_{\alpha^*E} \colon \mathbf{0}_{\mathbf{Sh\,A}} \longrightarrow \alpha^*E\right)$$
$$\boldsymbol{i}_!A = \left(\alpha_!A, \eta_A \colon A \longrightarrow \alpha^*\alpha_!A\right)$$

Thus $\mathbf{Sh\,I}$ and $\mathbf{Sh\,A}$ are not only reflective in $\mathbf{Sh\,G}$ — they are also *coreflective*.

6.5 The language of synthetic Tait computability

As $\mathbf{I}$ and $\mathbf{A}$ are both subtopoi of $\mathbf{G}$, their reflections (Section 6.4.1) can be expressed in the internal language of $\mathbf{Sh\,G}$ by means of a pair of complementary lex idempotent monads ($\bigcirc$,$\bullet$). The internal language of $\mathbf{Sh\,I}$ is presented by the $\bigcirc$-modal or *object-space* types and $\mathbf{Sh\,A}$ is presented by $\bullet$-modal or *meta-space* types. Because they form an open/closed partition, these modal subuniverses admit a particularly simple formulation:

Theorem 22. *There exists a proposition* $\mathrm{obj} : \Omega$ *such that*

(1) *a type X is $\bigcirc$-modal / object-space iff $X \to (\mathrm{obj} \to X)$ is an isomorphism;*
(2) *a type X is $\bullet$-modal / meta-space iff $\mathrm{obj} \times X \to \mathrm{obj}$ is an isomorphism.*

Remark 23. *In fact, the coreflection of* $\mathbf{Sh\,I}$ *in* $\mathbf{Sh\,G}$ *lifts smoothly into the internal language (though we shall not use this fact) by the idempotent comonadic modality $\Box \dashv \bigcirc$ that sends X to the product $\Box X = \mathrm{obj} \times X$. On the other hand, the coreflection of* $\mathbf{Sh\,A}$ *in* $\mathbf{Sh\,G}$ *cannot be expressed directly in the internal language.*

Notation 24. *We will use extension types $\{A \mid \phi \hookrightarrow a\}$ in the internal language of* $\mathbf{Sh\,G}$ *as realized by the subset comprehension of topos logic, treating their introduction and elimination rules silently. Here ϕ will be an element of the subobject classifier, in contrast to the situation in our object language, where it ranged over fixed proposition symbols.*

Remark 25. *We assume a subuniverse $\Omega_{dec} \subseteq \Omega$ of the subobject classifier that is closed under finite disjunctions and contains* obj*; then Ω_{dec} will ultimately be a subuniverse spanned by pointwise/externally decidable propositions [2], but this fact will not play a role in the synthetic development.*

Notation 26. *We will reuse Notation 14 and write* $\{\mathrm{obj}\}\, A$ *rather than* $\{\_ : \mathrm{obj}\} \to A$ *when* $A : \{\_ : \mathrm{obj}\} \to \mathcal{U}$.

As a presheaf topos, $\mathbf{Sh\,G}$ inherits a hierarchy of cumulative universes $\mathcal{U}_i$, each of which supports the ***strict gluing*** or ***(mixed-phase) refinement*** type [19]: a version of the dependent sum of a family of meta-space types indexed in an object-space type A that *additionally* restricts within

object-space to exactly A:

$$\frac{A : \{\mathsf{obj}\}\,\mathcal{U}_i \qquad B : (\{\mathsf{obj}\}\,A) \to \mathcal{U}_i \qquad \{\mathsf{obj}\}(a : A) \to (B\,a \cong \mathbf{1})}{(x : A) \ltimes B\,x : \{\mathcal{U}_i \mid \mathsf{obj} \hookrightarrow A\} \qquad \mathsf{gl} : \{((x : \{\mathsf{obj}\}\,A) \times B\,x) \cong (x : A) \ltimes B\,x \mid \mathsf{obj} \hookrightarrow \pi_1\}}$$

Remark 27. *In topos logic, it is a* property *for a function to have an inverse; thus we have conveniently packaged the introduction and elimination rules for $(x : A) \ltimes B\,x$ into a single function* gl *that is assumed to be an isomorphism.*

Notation 28. *We write* $[\mathsf{obj} \hookrightarrow a \mid b]$ *for* $\mathsf{gl}\,(a, b)$ *and* $\mathsf{ungl}\,x$ *for* $\pi_2\big(\mathsf{gl}^{-1}x\big)$*. When constructing particularly complex inhabitants of $(x : A) \ltimes B\,x$ we will avail ourselves of copattern matching notation and write the following instead of $c = [\mathsf{obj} \hookrightarrow a \mid b]$:*

$\mathsf{obj} \hookrightarrow c = a$
$\mathsf{ungl}\,c = b$

Both $\bigcirc$ and $\bullet$ induce reflective subuniverses $\mathcal{U}^i_\bigcirc, \mathcal{U}^i_\bullet \hookrightarrow \mathcal{U}_i$ spanned by modal types, and these universes are themselves modal. Following Sterling [39], we use strict gluing to choose these universes with additional strict properties:

$$\mathcal{U}^i_\bigcirc : \{\mathcal{U}_{i+1} \mid \mathsf{obj} \hookrightarrow \mathcal{U}_i\} \qquad \mathcal{U}^i_\bullet : \{\mathcal{U}_{i+1} \mid \mathsf{obj} \hookrightarrow \mathbf{1}\}$$

Furthermore, the inclusion $\mathcal{U}^i_\bigcirc \hookrightarrow \mathcal{U}_i$ restricts to the identity under obj. With the modal universes to hand, we may choose $\bigcirc : \mathcal{U}_i \to \mathcal{U}_i$ and $\bullet : \mathcal{U}_i \to \mathcal{U}_i$ to factor through $\mathcal{U}^i_\bigcirc$ and $\mathcal{U}^i_\bullet$ respectively. Henceforth we will suppress the inclusions $\mathcal{U}^i_\bigcirc, \mathcal{U}^i_\bullet \hookrightarrow \mathcal{U}_i$ and write *e.g.* $\bigcirc : \mathcal{U}_i \to \mathcal{U}^i_\bigcirc$ for the reflections.

Remark 29. *The strict gluing types, modal universes, and their modal reflections can be chosen to commute strictly with the liftings $\mathcal{U}_i \longrightarrow \mathcal{U}_{i+1}$.*

The interpretation of the $\mathbf{TT}_{\mathbb{P}}$ signature within $\mathbf{Sh}\,\mathbf{I}$ internalizes into $\mathbf{Sh}\,\mathbf{G}$ as a sequence of constants valued in the subuniverse $\mathcal{U}^0_\bigcirc$; for instance, we have:

$\mathsf{tp} : \mathcal{U}^0_\bigcirc$
$\mathsf{tm} : \mathsf{tp} \to \mathcal{U}^0_\bigcirc$
$\langle p \rangle : \Omega_{dec}$ (for $p \in \mathbb{P}$)
$\mathsf{ext}_p : (A : \mathsf{tp}) \to (a : \{\langle p \rangle\}\,\mathsf{tm}\,A) \to \mathsf{tp}$
$\mathsf{in}_p : (A : \mathsf{tp})\,(a : \{\langle p \rangle\}\,\mathsf{tm}\,A) \to \{\mathsf{tm}\,A \mid \langle p \rangle \hookrightarrow a\} \cong \mathsf{tm}\,(\mathsf{ext}_p\,A\,a)$

Following Remark 27, we package the pair $(\mathsf{in}_p, \mathsf{out}_p)$ as a single isomorphism in_p.

The presheaf of terms in the model $\mathbf{A}$ internalizes as a meta-space type of *variables* which by virtue of the structure map $\mathbf{A} \longrightarrow \mathbf{I}$ can be indexed over the object-space collection of terms. We realize this synthetically as follows:

$\mathsf{var} : (A : \mathsf{tp}) \to \{\mathcal{U} \mid \mathsf{obj} \hookrightarrow \mathsf{tm}\,A\}$

We refer to extensional type theory extended with these constants and modalities as the language of ***synthetic Tait computability*** (STC).

Remark 30. *To account for strict universes—those for which* el *commutes strictly with chosen codes—some prior STC developments employed strict gluing along the image of* el *[39, 41]. By limiting our usage of strict gluing to* obj*, we are able to execute our constructions in a constructive metatheory. To model strict universes, we instead use the cumulativity of the hierarchy of universes $\mathcal{U}_i$ and the fact that all levels are coherently closed under modalities and strict gluing.*

$$\mathbf{var} : (x : \mathsf{var}\, A) \to \mathsf{ne}_\bullet\, A \perp x$$
$$\mathbf{unst} : \{a : \mathsf{tm}\, A\} \to \mathsf{ne}_\bullet\, A \top a$$
$$\_ : (a : \mathsf{tm}\, A)\, (e : \mathsf{ne}_\bullet\, A \top a) \to e = \mathbf{unst}\, a$$

$$\mathbf{ext}_p : \mathsf{nftp}_\bullet\, A \to (\{p\}\, \mathsf{nf}_\bullet\, A\, a) \to \mathsf{nftp}_\bullet\, (\mathsf{ext}_p\, A\, a)$$
$$\mathbf{in}_p : \mathsf{nftp}_\bullet\, A \to \mathsf{nf}_\bullet\, A\, u \to \mathsf{nf}_\bullet\, (\mathsf{ext}_p\, A\, a)\, u$$
$$\mathbf{out}_p : \mathsf{nftp}_\bullet\, A \to \mathsf{ne}_\bullet\, (\mathsf{ext}_p\, A\, a)\, \phi\, u \to \mathsf{ne}_\bullet\, A\, (\phi \vee \langle p \rangle)\, \left(\mathsf{in}_p^{-1}\, u\right)$$

$$\mathbf{up}_{\mathsf{uni}} : \mathsf{ne}_\bullet\, \mathsf{uni}\, \phi\, A \to (\{\phi\}\, \mathsf{nftp}_\bullet\, (\mathsf{el}\, A)) \to \mathsf{nftp}_\bullet\, (\mathsf{el}\, A)$$
$$\mathbf{up}_{\mathsf{el}} : \mathsf{ne}_\bullet\, \mathsf{uni}\, \phi\, A \to (\{\phi\}\, \mathsf{nftp}_\bullet\, (\mathsf{el}\, A)) \to \mathsf{ne}_\bullet\, (\mathsf{el}\, A)\, \psi\, a \to (\{\phi \vee \psi\}\, \mathsf{nf}_\bullet\, (\mathsf{el}\, A)\, a) \to \mathsf{nf}_\bullet\, (\mathsf{el}\, A)\, a$$
$$\mathbf{el} : \mathsf{ne}_\bullet\, \mathsf{uni}\, \phi\, A \to (\{\phi\}\, \mathsf{nf}_\bullet\, \mathsf{uni}\, A) \to \mathsf{nf}_\bullet\, \mathsf{uni}\, A$$

$$\_ : (e : \mathsf{ne}_\bullet\, \mathsf{uni} \top A)\, (u : \mathsf{nftp}_\bullet\, A) \to \mathbf{up}_{\mathsf{uni}}\, e\, u = u$$
$$\_ : (e : \mathsf{ne}_\bullet\, \mathsf{uni} \top A)\, (u : \mathsf{nf}_\bullet\, \mathsf{uni}\, A) \to \mathbf{el}\, e\, u = u$$
$$\_ : \{\phi \vee \psi\}\, (e_A : \mathsf{ne}_\bullet\, \mathsf{uni}\, \phi\, A)\, (u_A : \{\phi\}\, \mathsf{nftp}_\bullet\, (\mathsf{el}\, A))(e_a : \mathsf{ne}_\bullet\, (\mathsf{el}\, A)\, \psi\, a)\, (u_a : \mathsf{nf}_\bullet\, A\, a)$$
$$\to \mathbf{up}_{\mathsf{el}}\, e_A\, u_A\, e_a\, u_a = u_a$$

Fig. 2. Selected rules from the definition of nf, ne, and nftp.

6.6 Normal and neutral forms

Internally to STC, we now specify the normal and neutral forms of terms, and the normal forms of types. Following Sterling and Angiuli [41] we index the type of neutral forms by a ***frontier of instability***, a proposition at which the neutral form is no longer meaningful. Our construction proceeds in two steps. First, we define a series of indexed quotient-inductive definitions [28] specifying the meta-space components of normal and neutral forms:

$$\mathsf{nf}_\bullet : (A : \mathsf{tp}) \to \mathsf{tm}\, A \to \mathcal{U}^0_\bullet$$
$$\mathsf{ne}_\bullet : (A : \mathsf{tp}) \to \Omega_{dec} \to \mathsf{tm}\, A \to \mathcal{U}^0_\bullet$$
$$\mathsf{nftp}_\bullet : \mathsf{tp} \to \mathcal{U}^0_\bullet$$

Next we use the strict gluing connective to define the types of normals, neutrals, and normal types such that they lie strictly over tm and tp:

$$\mathsf{nf}\, A = (a : \mathsf{tm}\, A) \ltimes \mathsf{nf}_\bullet\, A\, a$$
$$\mathsf{ne}_\phi\, A = (a : \mathsf{tm}\, A) \ltimes \mathsf{ne}_\bullet\, A\, \phi\, a$$
$$\mathsf{nftp} = (A : \mathsf{tp}) \ltimes \mathsf{nftp}_\bullet\, A$$

We illustrate a representative fragment of the inductive definitions in Figure 2.

The induction principles for $\mathsf{nf}_\bullet$, $\mathsf{ne}_\bullet$ and $\mathsf{nftp}_\bullet$ play no role in the main development, which works with *any* algebra for these constants. These induction principles, however, are needed in order to prove Theorem 48 and deduce the decidability of definitional equality and the injectivity of type constructors. These same considerations motivate our choice to index $\mathsf{ne}_\bullet$ over Ω_{dec} rather than Ω.

6.7 A glued normalization algebra

We can now construct a new $\mathbf{TT}_{\mathbb{P}}$-algebra internally to $\mathbf{Sh}\,\mathbf{G}$, satisfying the constraint that each of its constituents restricts under obj to the corresponding constant from the $\mathbf{TT}_{\mathbb{P}}$-algebra inherited from $\mathbf{Sh}\,\mathbf{I}$. We shall refer to this as the ***normalization algebra***. For instance, we must define types representing object types and terms:

$$\mathsf{tp}^* : \{\mathcal{U}_2 \mid \mathsf{obj} \hookrightarrow \mathsf{tp}\} \qquad \mathsf{tm}^* : \{\mathsf{tp}^* \to \mathcal{U}_1 \mid \mathsf{obj} \hookrightarrow \mathsf{tm}\}$$

The meta-space component of the computability structure of types is given as a dependent record below:

$$\begin{aligned}&\textbf{record}\ \mathsf{tp}_\bullet\ (A : \mathsf{tp}) : \mathcal{U}_2\ \textbf{where}\\ &\quad \mathsf{code} : \mathsf{nftp}_\bullet\ A\\ &\quad \mathsf{tm}_\bullet : \mathsf{tm}\ A \to \mathcal{U}^1_\bullet\\ &\quad \mathsf{reflect} : (a : \mathsf{tm}\ A)\ (\phi : \Omega_{dec})\ (e : \mathsf{ne}_\bullet\ A\ \phi\ a) \to (a_\phi : \{\phi\}\ \mathsf{tm}_\bullet\ a) \to \{\mathsf{tm}_\bullet\ a \mid \phi \hookrightarrow a_\phi\}\\ &\quad \mathsf{reify} : (a : \mathsf{tm}\ A) \to \mathsf{tm}_\bullet\ a \to \mathsf{nf}_\bullet\ A\ a\end{aligned}$$

The $\mathsf{tm}_\bullet$ field classifies the meta-space component of a given element; the reflect and reify fields generalize the familiar operations of normalization by evaluation, subject to Sterling and Angiuli's stabilization yoga [41]. We finally define both tp^* and tm^* using strict gluing to achieve the correct boundary:

$$\begin{aligned}&\mathsf{tp}^* = (A : \mathsf{tp}) \ltimes \mathsf{tp}_\bullet\ A\\ &\mathsf{tm}^*\ A = (a : \mathsf{tm}\ A) \ltimes (\mathsf{ungl}\ A).\mathsf{tm}_\bullet\ a\end{aligned}$$

Notation 31. *Henceforth we will write* $A.\mathsf{fld}$ *rather than* $(\mathsf{ungl} A).\mathsf{fld}$ *to access a field of the closed component of* A.

We must also define $\langle p\rangle^* : \Omega_{dec}$ for each $p \in \mathbb{P}$ subject to the condition that obj implies $\langle p\rangle^* = \langle p\rangle$. As there is no normalization data associated with these propositions, we define $\langle p\rangle^* = \langle p\rangle$ which clearly satisfies the boundary condition. It remains to show that $(\mathsf{tp}^*, \mathsf{tm}^*)$ are closed under all the connectives of $\mathbf{TT}_{\mathbb{P}}$. We show two representative cases: extension types and the universe.

6.7.1 ***Extension types.*** Fixing $A : \mathsf{tp}^*$, $p : \mathbb{P}$, $a : \{\langle p\rangle\}\ \mathsf{tm}^*\ A$, we must construct the following pair of constants:

$$\begin{aligned}&\mathsf{ext}^*_p\ A\ a : \{\mathsf{tp}^* \mid \mathsf{obj} \hookrightarrow \mathsf{ext}_p\ A\ a\}\\ &\mathsf{in}^*_p\ A\ a : \{\{\mathsf{tm}^*\ A \mid \langle p\rangle \hookrightarrow a\} \cong \mathsf{tm}^*\ (\mathsf{ext}^*_p\ A\ a) \mid \mathsf{obj} \hookrightarrow \mathsf{in}_p\ A\ a\}\end{aligned}$$

Recalling the definition of tp^* as a strict gluing type, we observe that the boundary condition on ext^*_p already fully constrains the first component:

$$\mathsf{ext}^*_p\ A\ a = [\mathsf{obj} \hookrightarrow \mathsf{ext}_p\ A\ a \mid \ ? : \mathsf{tp}_\bullet\ (\mathsf{ext}_p\ A\ a)\ ]$$

In the above, we have used Notation 28 for constructing elements of a strict gluing type.

We define the second component as follows, using copattern matching notation:

$$\begin{aligned}&(\mathsf{ext}^*_p\ A\ a).\mathsf{code} = \mathbf{ext}_p\ A.\mathsf{code}\ (A.\mathsf{reify}\ a)\\ &(\mathsf{ext}^*_p\ A\ a).\mathsf{tm}_\bullet\ x = \bullet\{A.\mathsf{tm}_\bullet\ (\mathsf{in}^{-1}_p x) \mid \langle p\rangle \hookrightarrow a\}\\ &(\mathsf{ext}^*_p\ A\ a).\mathsf{reify}\ (\eta_\bullet x) = \mathbf{in}_p\ A.\mathsf{code}\ (A.\mathsf{reify}\ x)\\ &(\mathsf{ext}^*_p\ A\ a).\mathsf{reflect}\ x\ \phi\ e\ (\eta_\bullet x_\phi) =\\ &\quad \eta_\bullet \begin{pmatrix} A.\mathsf{reflect} \\ \quad (\mathsf{in}^{-1}_p x)\ (\phi \vee \langle p\rangle) \\ \quad (\mathbf{out}_p\ A.\mathsf{code}\ e)\ [\phi \hookrightarrow x_\phi \mid \langle p\rangle \hookrightarrow a] \end{pmatrix}\end{aligned}$$

In the clauses of reify and reflect, we were allowed to assume that the argument was of the form $\eta_\bullet x$ where $\eta_\bullet$ is the unit of the modality $\bullet$: $\eta_\bullet : A \to \bullet A$. This is because we are mapping into meta-space types and so this "pattern-matching" amounts to the bind operation of the monad $\bullet$.

Remark 32. *Stabilized neutrals are crucial to the definition of* $(\mathsf{ext}^*_p\ A\ a).\mathsf{reflect}$ *above: without them, we could not ensure that reflecting* $\mathbf{out}_p\ A.\mathsf{code}\ e$ *lies within the specified subtype of* $A.\mathsf{tm}_\bullet$.

The definition of in^*_p is now straightforward:

$$\mathsf{in}^*_p\ A\ a\ x = [\mathsf{obj} \hookrightarrow \mathsf{in}_p\ A\ a\ x \mid \mathsf{ungl}\ x]$$

record $\mathsf{uni}_\bullet\, A : \mathcal{U}_1$ **where**
  $\mathsf{code} : \mathsf{nf}_\bullet\, \mathsf{uni}\, A$
  $\mathsf{el\text{-}code} : \mathsf{nftp}_\bullet\, (\mathsf{el}\, A)$
  $\mathsf{tm}_\bullet : \mathsf{tm}\, (\mathsf{el}\, A) \to \mathcal{U}^0_\bullet$
  $\mathsf{reflect} : (a : \mathsf{tm}\, A)\, (\phi : \Omega_{dec}) \to (e : \mathsf{ne}_\bullet\, A\, \phi\, a) \to (a_\phi : \{\phi\}\, \mathsf{tm}_\bullet\, a) \to \{\mathsf{tm}_\bullet\, a \mid \phi \hookrightarrow a_\phi\}$
  $\mathsf{reify} : (a : \mathsf{tm}\, A) \to \mathsf{tm}_\bullet\, a \to \mathsf{nf}_\bullet\, A\, a$
$\mathsf{uni}^* : \{\mathsf{tp}^* \mid \mathsf{obj} \hookrightarrow \mathsf{uni}\}$
$\mathsf{uni}^*.\mathsf{code} = \mathbf{uni}$
$\mathsf{uni}^*.\mathsf{tm}_\bullet\, A = \bullet\, (\mathsf{uni}_\bullet\, A)$
$\mathsf{uni}^*.\mathsf{reify}\, \_\, (\eta_\bullet A) = A.\mathsf{code}$
$\mathsf{obj} \hookrightarrow \mathsf{uni}^*.\mathsf{reflect}\, A\, \phi\, e_A\, A_\phi = A$
$\mathsf{ungl}\, (\mathsf{uni}^*.\mathsf{reflect}\, A\, \phi\, e_A\, A_\phi) =$
  let $A_\phi : \{\phi\}\, \mathsf{uni}_\bullet\, A = X \leftarrow A_\phi; X;$
  $\eta_\bullet$ **record**
    $\mathsf{code} = \mathbf{up}_{\mathsf{uni}}\, e_A\, A_\phi.\mathsf{code}$
    $\mathsf{el\text{-}code} = \mathbf{el}\, e_A\, A_\phi.\mathsf{el\text{-}code}$
    $\mathsf{tm}_\bullet\, a = \bullet((u : \mathsf{nf}_\bullet\, A\, x) \times \{\phi\}\{a : A_\phi.\mathsf{tm}_\bullet\, a \mid A_\phi.\mathsf{reify}\, a = u\})$
    $\mathsf{reflect}\, a\, \psi\, e_a\, a_\psi =$
      let $a_\psi : \{\psi\}\, (u : \mathsf{nf}_\bullet\, A\, x) \times \ldots = x \leftarrow a_\psi; x;$
      let $a_\phi = A_\phi.\mathsf{reflect}\, \psi\, e_a\, a_\psi;$
      $\eta_\bullet(\mathbf{el}\, e_A\, A_\phi.\mathsf{el\text{-}code}\, e_a\, [\phi \hookrightarrow A_\phi.\mathsf{reify}\, a_\phi \mid \psi \hookrightarrow \pi_1\, a_\psi], a_\phi)$
    $\mathsf{reify}\, \_\, (\eta_\bullet\, (u, \_)) = u$

Fig. 3. The normalization structure on the universe.

We leave the routine verification of the various boundary conditions to the reader; nearly all of them follow immediately from the properties of strict gluing.

6.7.2 ***The universe.*** We now turn to the construction of the universe in the normalization algebra; it is here that the complexity of unstable neutrals becomes evident. Once again the boundary conditions on uni^* force part of its definition:

$\mathsf{uni}^* = [\mathsf{obj} \hookrightarrow \mathsf{uni} \mid \ ? : \mathsf{tp}_\bullet\, \mathsf{uni}\ ]$

The second component of uni^* is complex and we present its definition in Figure 3. The inclusion of el-code in $\mathsf{uni}_\bullet$ is necessary in order to define el^*:

$\mathsf{obj} \hookrightarrow \mathsf{el}^*\, A = \mathsf{el}\, A$
$(\mathsf{el}^*\, (\eta_\bullet A)).\mathsf{code} = A.\mathsf{el\text{-}code}$
$(\mathsf{el}^*\, (\eta_\bullet A)).\mathsf{tm}_\bullet = A.\mathsf{tm}_\bullet$
$(\mathsf{el}^*\, (\eta_\bullet A)).\mathsf{reflect} = A.\mathsf{reflect}$
$(\mathsf{el}^*\, (\eta_\bullet A)).\mathsf{reify} = A.\mathsf{reify}$

Finally, we must show that uni^* is closed under all small type formers and that el^* preserves them. This flows from the cumulativity of universes in **Sh G**; to close uni^* under *e.g.* products, we essentially 'redo' the construction of products in tp^* by altering its predicate to be valued in $\mathcal{U}_0$ rather than $\mathcal{U}_1$.

6.7.3 *The evaluation functor.* In this section, we equip **Sh G** with the maximal CwR structure, in which *all* maps are representable.[5] We have just now defined an interpretation of $\mathbf{TT}_{\mathbb{P}}$'s signature

[5]It would be possible to choose a more restrictive class of representable maps for **Sh G**, but there is no reason to do so.

(Section 6.2) in $\mathbf{Sh}\,\mathbf{G}$, and so by the universal property (Proposition 10) of $\mathbb{T}$ as the classifying CwR for this signature, we obtain a unique CwR functor $I_{\mathbf{Sh}\,\mathbf{G}} : \mathbb{T} \longrightarrow \mathbf{Sh}\,\mathbf{G}$ sending every construct of $\mathbf{TT}_{\mathbb{P}}$ to its interpretation.

6.8 The normalization algorithm

Having constructed the normalization algebra in $\mathbf{Sh}\,\mathbf{G}$ (Section 6.7), we can now define the actual normalization function using an argument based on those presented by Fiore [14, 15, §II.2] and Sterling [40, §3.3], making use of the *inserter model* of atomic substitutions introduced by Bocquet et al. [6]. As our results are constructive, our normalization function corresponds to an actual normalization by evaluation algorithm — whose executable computational content Fiore [15] has demonstrated explicitly in the simply typed case.

6.8.1 Stripping of atomic contexts. We first must establish an intermediate result: that the functor $\mathbf{A}\langle - \rangle : \mathbb{P} \longrightarrow \mathbf{A}_\diamond$ sending each $p \in \mathbb{P}$ to the associated unary atomic context is fully faithful and has a left adjoint, *i.e.* that $\mathbb{P}$ is reflective in $\mathbf{A}_\diamond$. The left adjoint allows an atomic context to be "stripped" of anything that induces variables, leaving only propositional assumptions. This result is ultimately used in Lemma 40 to exhibit an isomorphism $\mathbf{A}\langle p \rangle \cong \alpha^* \mathbf{I}\langle p \rangle$.

While it is straightforward to imagine how such a reflection can be defined by "induction on atomic contexts and repeated weakening", we have not given an inductive specification of $\mathbf{A}_\diamond$ and instead opted to specify it through its universal property. Accordingly, we define this stripping map using an model to which we may apply the universal property of $\mathbf{A}$. Fundamentally, however, the resulting constructions are the same, but our insistence on using only these universal properties enables us to avoid fixing a particular and explicit construction of atomic contexts.

Construction 33 (The stripping model)**.** *We consider a model* $\mathbf{P}$ *of* $\mathbb{T}$ *in which we set* $\mathbf{P}_\diamond = \mathbb{P}$, $\mathbf{P}(\mathsf{tp}) = \mathbf{P}(\mathsf{tm}) = \mathbf{1}_{\mathbf{Pr}\,\mathbb{P}}$, *and* $\mathbf{P}\langle p \rangle = \mathsf{y}_{\mathbb{P}} p$. *All the remaining constructs of the model are trivial by virtue of these definitions. From the universal property of* $\mathbf{I}$ *as the bi-initial* $\mathbb{T}$*-model, we obtain a unique homomorphism of models* $I_{\mathbf{P}} : \mathbf{I} \longrightarrow \mathbf{P}$ *whose contextual component is a product preserving functor* $I^\diamond_{\mathbf{P}} : \mathbf{I}_\diamond \longrightarrow \mathbb{P}$.

Lemma 34. *The component* $\mathbf{A}\langle - \rangle : \mathbb{P} \longrightarrow \mathbf{A}_\diamond$ *of the bi-initial atomic substitution model over* $\mathbf{I}$ *is full and faithful.*

Proof. Any functor out of a poset is necessarily faithful. To see that $\mathbf{A}\langle - \rangle : \mathbb{P} \longrightarrow \mathbf{A}_\diamond$ is full, we fix a morphism $\mathbf{A}\langle p \rangle \longrightarrow \mathbf{A}\langle q \rangle$ in $\mathbf{A}_\diamond$; as this morphism is necessarily unique, it suffices to show that $p \leq q$ in $\mathbb{P}$. We consider the image of $\mathbf{A}\langle p \rangle \longrightarrow \mathbf{A}\langle q \rangle$ under the contextual component of the composite homomorphism $I_{\mathbf{P}} \circ \alpha : \mathbf{A} \longrightarrow \mathbf{P}$ of $\mathbb{T}_0$-models, which gives precisely the desired inequality $p \leq q$, recalling that each $\langle r \rangle$ is is representable in $\mathbb{T}_0$ and thus preserved by homomorphisms of models. □

Lemma 35. *The functor between categories of contexts induced by* $I_{\mathbf{P}} \circ \alpha : \mathbf{A} \longrightarrow \mathbf{P}$ *is left adjoint to the embedding* $\mathbf{A}\langle - \rangle : \mathbb{P} \hookrightarrow \mathbf{A}_\diamond$.

Proof. The counit in $\mathbb{P}$ is given by the identity inequality, as each $\langle p \rangle$ is representable in $\mathbb{T}_0$ and thus preserved by homomorphisms. For the unit, we must construct a (necessarily unique) arrow $\Gamma \longrightarrow \mathbf{A}\langle I^\diamond_{\mathbf{P}}(\alpha_\diamond \Gamma) \rangle$ in $\mathbf{A}_\diamond$ for each atomic context Γ.

For this, we consider a new atomic substitution model $\mathbf{E}$ over $\mathbf{A}$ whose category of contexts $\mathbf{E}_\diamond$ is the following inserter object [31, Section 6.5] in $\mathbf{Cat}$:

$$\mathbf{E}_\diamond \xrightarrow{\ \phi_\diamond\ } \mathbf{A}_\diamond \underset{\mathbf{A}\langle - \rangle \circ I^\diamond_{\mathbf{P}} \circ \alpha_\diamond}{\overset{1_{\mathbf{A}_\diamond}}{\rightrightarrows}} \mathbf{A}_\diamond$$

Equivalently, $\mathbf{E}_\diamond$ is the full subcategory of $\mathbf{A}_\diamond$ spanned by atomic contexts Γ for which there exists an arrow $\Gamma \longrightarrow \mathbf{A}\langle I_\mathbf{P}^\diamond(\alpha_\diamond\Gamma)\rangle$; as the codomain is subterminal, such arrows are necessarily unique. We define all the constructs of $\mathbb{T}_0$ in $\mathbf{E}$ as in $\mathbf{A}$, and it remains only to check that $\mathbf{E}$ has a terminal object and is closed under context comprehension and phase comprehension.

(1) For the terminal object, we see that $\mathbf{A}\langle I_\mathbf{P}^\diamond(\alpha_\diamond \mathbf{1})\rangle$ is already terminal.
(2) For the context comprehension, we fix $\Gamma \in \mathbf{E}_\diamond$ and $A \in \mathbf{A}(\mathrm{tp})(\Gamma)$, and we must check that there exists a map $\Gamma.A \longrightarrow \mathbf{A}\langle I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma.A))\rangle$. As $\alpha \circ I_\mathbf{P}$ is a homomorphism of models, it preserves context comprehensions; unraveling definitions, we ultimately have $I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma.A)) = I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma))$ and so we are done.
(3) For phase comprehension, we fix $\Gamma \in \mathbf{E}_\diamond$ and $p \in \mathbb{P}$ to check that there exists an arrow $\Gamma.\mathbf{A}\langle p\rangle \longrightarrow \mathbf{A}\langle I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma.\mathbf{A}(\langle p\rangle)))\rangle$. But we have $I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma.\mathbf{A}(\langle p\rangle))) = I_\mathbf{P}^\diamond(\alpha_\diamond(\Gamma)) \wedge p$, so we may use the projection $\Gamma.\mathbf{A}\langle p\rangle \longrightarrow \mathbf{A}\langle p\rangle$.

We evidently have a homomorphism of $\mathbb{T}_0$-models $\eta\colon \mathbf{E} \longrightarrow \mathbf{A}$ that exhibits $\mathbf{E}$ as a atomic substitution model over $\mathbf{A}$. Postcomposing with the structure map $\alpha\colon \mathbf{A} \longrightarrow \mathbf{I}$, we can view $\mathbf{E}$ as a atomic substitution model over $\mathbf{I}$. Thus, by the universal property of $\mathbf{A}$ we have a universal *section* $J_\mathbf{E}\colon \mathbf{A} \longrightarrow \mathbf{E}$ to $\eta\colon \mathbf{E} \longrightarrow \mathbf{A}$. This shows that every atomic context $\Gamma \in \mathbf{A}_\diamond$ can be equipped with an arrow $\Gamma \longrightarrow \mathbf{A}\langle I_\mathbf{P}^\diamond(\alpha_\diamond\Gamma)\rangle$. Assembling all these arrows together, we have the unit of the adjunction $I_\mathbf{P}^\diamond \circ \alpha_\diamond \dashv \mathbf{A}\langle -\rangle$. □

The force of Lemma 35 is to show that $\mathbb{P}$ is a reflective subcategory of $\mathbf{A}_\diamond$.

6.8.2 Computability spaces of atomic and computable substitutions. We will consider two computability spaces induced by an atomic context Γ: the computability space $[\![\Gamma]\!]$ of "computable substitutions into Γ" and the computability space $(\!|\Gamma|\!)$ of "atomic substitutions into Γ".

Construction 36 (The computability space of computable substitutions)**.** *The computability space $[\![\Gamma]\!]$ of computable substitutions into an atomic context Γ is defined in terms of the* interpretation *of $\mathbb{T}$ into $\mathbf{Sh}\,\mathbf{G}$ as follows, sending each atomic context to the computability space determined by the algebra structure:*

$$\begin{array}{ccccccc} \mathbf{A}_\diamond & \xrightarrow{\alpha_\diamond} & \mathbf{I}_\diamond & \xhookrightarrow{\subseteq} & \mathbb{T} & \xrightarrow{I_{\mathbf{Sh}\,\mathbf{G}}} & \mathbf{Sh}\,\mathbf{G} \\ & \searrow & & & & \nearrow & \\ & & & [\![-]\!] & & & \end{array}$$

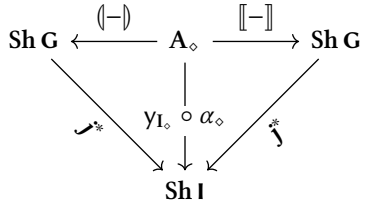
Construction 37 (The computability space of atomic substitutions)**.** *We define an embedding $(\!|-|\!)\colon \mathbf{A}_\diamond \hookrightarrow \mathbf{Sh}\,\mathbf{G}$ sending $\Gamma \in \mathbf{A}_\diamond$ to the computability space $(\!|\Gamma|\!)$ with $\boldsymbol{j}^*(\!|\Gamma|\!) = \mathrm{y}_{\mathbf{I}_\diamond}\alpha_\diamond\Gamma$ and $\boldsymbol{i}^*(\!|\Gamma|\!) = \mathrm{y}_{\mathbf{A}_\diamond}\Gamma$, such that $\pi_{(\!|\Gamma|\!)}\colon \mathrm{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \alpha^*\mathrm{y}_{\mathbf{I}_\diamond}\alpha_\diamond\Gamma$ is defined on generalised elements by the functorial action of $\alpha_\diamond\colon \mathbf{A}_\diamond \longrightarrow \mathbf{I}_\diamond$ as follows:*

$$\pi^\Delta_{(\!|\Gamma|\!)}\big(\gamma\colon \Delta \longrightarrow \Gamma\big) = \alpha_\diamond\gamma\colon \alpha_\diamond\Delta \longrightarrow \alpha_\diamond\Gamma$$

Miraculously, the coreflective *embedding $\boldsymbol{i}_!\colon \mathbf{Sh}\,\mathbf{A} \hookrightarrow \mathbf{Sh}\,\mathbf{G}$ sends $\mathrm{y}_{\mathbf{A}_\diamond}\Gamma$ to **precisely** the computability space $(\!|\Gamma|\!)$, up to isomorphism:*

$$\begin{aligned} \boldsymbol{i}_!\mathrm{y}_{\mathbf{A}_\diamond}\Gamma &= \big(\alpha_!\mathrm{y}_{\mathbf{A}_\diamond}\Gamma, \mathrm{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \alpha^*\alpha_!\mathrm{y}_{\mathbf{A}_\diamond}\Gamma\big) \\ &\cong \big(\mathrm{y}_{\mathbf{I}_\diamond}\alpha_\diamond\Gamma, \mathrm{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \alpha^*\mathrm{y}_{\mathbf{I}_\diamond}\alpha_\diamond\Gamma\big) \\ &= (\!|\Gamma|\!) \end{aligned}$$

Note that the functors $[\![-]\!], (\!|-|\!)$ lift into the slice $\mathbf{Cat} \downarrow \mathbf{Sh}\,\mathbf{I}$ in the following sense:



Notation 38. *Let Γ be an atomic context, and let $A : (\!|\Gamma|\!) \longrightarrow \mathsf{tp}$ be an object-space type, which we may regard as a morphism $\alpha\Gamma \longrightarrow \mathsf{tp}$ in $\mathbb{T}$. We shall write $[\![A]\!] : [\![\Gamma]\!] \to \mathsf{tp}^*$ for the image of A under the interpretation functor $I_{\mathbf{Sh}\,\mathbf{G}}$.*

Lemma 39. *Let Γ be an atomic context, and let $A : (\!|\Gamma|\!) \longrightarrow \mathsf{tp}$ be an object-space type (which we may regard as a morphism $\alpha_\diamond\Gamma \longrightarrow \mathsf{tp}$ in $\mathbb{T}$). Then we have the following cartesian squares:*

$$\begin{array}{ccc} (\!|\Gamma.A|\!) & \longrightarrow & \sum_{A:\mathsf{tp}} \mathsf{var}\,A \\ {\scriptstyle (\!|p_A|\!)}\downarrow & & \downarrow{\scriptstyle \pi_1} \\ (\!|\Gamma|\!) & \xrightarrow[A]{} & \mathsf{tp} \end{array} \qquad \begin{array}{ccc} [\![\Gamma.A]\!] & \longrightarrow & \sum_{A:\mathsf{tp}^*} \mathsf{tm}^* A \\ \downarrow & & \downarrow{\scriptstyle \pi_1} \\ [\![\Gamma]\!] & \xrightarrow[[\![A]\!]]{} & \mathsf{tp}^* \end{array}$$

Stated in the internal language, we have canonical isomorphisms $(\!|\Gamma.A|\!) \cong \sum_{\gamma:(\!|\Gamma|\!)} \mathsf{var}\,(A\gamma)$ and $[\![\Gamma.A]\!] \cong \sum_{\gamma:[\![\Gamma]\!]} \mathsf{tm}^*\,([\![A]\!]\gamma)$.

Proof. The latter is the image of a pullback square in $\mathbb{T}$ under $I_{\mathbf{Sh}\,\mathbf{G}}$, which is finitely continuous. The former can be seen by means of an explicit computation. □

We now come to an important result relating the interpretation of $\langle p \rangle$ in $\mathbf{I}$ to the interpretation of the same in $\mathbf{A}$. Lemma 40 below is the *raison d'être* for the stripping model $\mathbf{P}$ in Section 6.8.1.

Lemma 40. *We have a (necessarily unique) isomorphism $\mathbf{A}\langle p \rangle \cong \alpha^*\mathbf{I}\langle p \rangle$.*

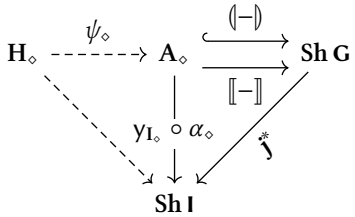
Proof. As both presheaves are subterminal, it is enough to see that one is inhabited if and only if the other is.

(1) We may transpose a map $\mathsf{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \alpha^*\mathbf{I}\langle p \rangle$ to get $\alpha_\diamond\Gamma \longrightarrow \mathbf{I}\langle p \rangle$; applying the functorial action of $I_{\mathbf{P}}^\diamond : \mathbf{I}_\diamond \longrightarrow \mathbb{P}$, we have $I_{\mathbf{P}}^\diamond\alpha_\diamond\Gamma \leq I_{\mathbf{P}}^\diamond\mathbf{I}\langle p \rangle = p$ and by adjoint transpose with Lemma 35, we have $\Gamma \longrightarrow \mathbf{A}\langle p \rangle$.
(2) Conversely, given an arrow $\Gamma \longrightarrow \mathbf{A}\langle p \rangle$ we may apply the functorial action of $\alpha_\diamond : \mathbf{A}_\diamond \longrightarrow \mathbf{I}_\diamond$ to obtain an arrow $\alpha_\diamond\Gamma \longrightarrow \alpha_\diamond\mathbf{A}\langle p \rangle$. As $\langle p \rangle$ is representable in $\mathbb{T}_0$, it is preserved by morphisms of models like $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$; thus $\alpha_\diamond\mathbf{A}\langle p \rangle \cong \mathbf{I}\langle p \rangle$ and so we have $\alpha_\diamond\Gamma \longrightarrow \mathbf{I}\langle p \rangle$, which we may transpose to obtain $\mathsf{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \alpha^*\mathbf{I}\langle p \rangle$. □

6.8.3 Hydration of atomic substitutions. A critical point in concrete normalization by evaluation algorithms is to "reflect" a vector of variables as an environment of (computable) values against which the computability interpretation of an open term can be executed; in a concrete setting, this operation is defined by recursion on the atomic contexts. The same process, which we shall refer to here as the ***hydration of atomic substitutions***, plays an equally important role in semantic proofs of normalization in the guise of a certain ***hydration map*** $\nearrow : (\!|-|\!) \longrightarrow [\![-]\!]$ in $\mathbf{Cat} \downarrow \mathbf{Sh}\,\mathbf{I}$ that we shall need to construct.

Just as in the definition of the stripping map, we are confronted by the fact that we have defined the atomic contexts only by means of a universal property (the bi-initial atomic substitution model over **I**), so we do not immediately have anything concrete to do recursion on. The innovation of Bocquet et al. [6] was to find the correct *categorical induction motive* that explains the usual recursive argument purely in terms of the universal property of the bi-initial atomic substitution model (likewise due to *op. cit.*). In what follows, we adapt their ideas to our setting and show how to construct the desired hydration map.

We begin by defining a new model $\mathbf{H}$ of $\mathbb{T}_0$ that we shall refer to as the ***hydration model***. The category of contexts $\mathbf{H}_\diamond$ is defined to be the underlying category of the *inserter object* for $(\!|-|\!), [\![-]\!] : \mathbf{A}_\diamond \longrightarrow \mathbf{Sh}\,\mathbf{G}$ in $\mathbf{Cat} \downarrow \mathbf{Sh}\,\mathbf{I}$ as depicted below:



Explicitly, an object of $\mathbf{H}_\diamond$ is a pair (Γ, h_Γ) of an object $\Gamma \in \mathbf{A}_\diamond$ together with an arrow $h_\Gamma : (\!|\Gamma|\!) \longrightarrow [\![\Gamma]\!]$ whose image under $j^* : \mathbf{Sh}\,\mathbf{G} \longrightarrow \mathbf{Sh}\,\mathbf{I}$ is the identity map on $\mathsf{y}_{\mathbf{I}_\diamond}\alpha_\diamond\Gamma$. An arrow from (Δ, h_Δ) to (Γ, h_Γ) is given by an arrow $\gamma : \Delta \longrightarrow \Gamma$ in $\mathbf{A}_\diamond$ making the following square commute:

$$\begin{array}{ccc} (\!|\Delta|\!) & \xrightarrow{(\!|\gamma|\!)} & (\!|\Gamma|\!) \\ {\scriptstyle h_\Delta}\downarrow & & \downarrow{\scriptstyle h_\Gamma} \\ [\![\Delta]\!] & \xrightarrow[[\![\gamma]\!]]{} & [\![\Gamma]\!] \end{array}$$

Clearly, $\mathbf{H}_\diamond$ has a terminal object because $[\![\mathbf{1}_{\mathbf{A}_\diamond}]\!]$ is terminal. Anticipating that $\psi_\diamond$ should lift to a morphism of models exhibiting $\mathbf{H}$ as a atomic substitution model over $\mathbf{A}$, we define $\mathbf{H}\langle p\rangle = \psi_\diamond^*\mathbf{A}\langle p\rangle$ and $\mathbf{H}(\mathsf{tp}) = \psi_\diamond^*\mathbf{A}(\mathsf{tp})$. In order to define $\mathbf{H}(\mathsf{tm})$, it will be easiest to first define context comprehensions.

Construction 41 (Context comprehensions in $\mathbf{H}_\diamond$)**.** *Given a context (Γ, h_Γ) in $\mathbf{H}_\diamond$ and a type $A : \mathsf{y}_{\mathbf{A}_\diamond}\Gamma \longrightarrow \mathbf{A}(\mathsf{tp})$, we can lift the context comprehension $\Gamma.A \in \mathbf{A}_\diamond$ into $\mathbf{H}_\diamond$ by finding a suitable map $h_{\Gamma.A} : (\!|\Gamma.A|\!) \longrightarrow [\![\Gamma.A]\!]$ whose image in $\mathbf{Sh}\,\mathbf{I}$ is the identity. Recalling Lemma 39, we may equivalently construct (from the internal point of view) the following map:*

$$h_{\Gamma.A} : \left\{ \left(\textstyle\sum_{\gamma:(\!|\Gamma|\!)} \mathsf{var}\,(A\gamma)\right) \longrightarrow \textstyle\sum_{\gamma:[\![\Gamma]\!]} \mathsf{tm}^*([\![A]\!]\gamma) \;\middle|\; \mathsf{obj} \hookrightarrow \lambda u.\, u \right\}$$
$$h_{\Gamma.A}(\gamma, x) = (h_\Gamma\gamma, [\![A]\!](h_\Gamma\gamma).\mathsf{reflect}\, x\, \bot\, (\mathbf{var}\, x)\, (\lambda\_ : \bot.\, \star))$$

The projection $p_A : \Gamma.A \longrightarrow \Gamma$ tracks a morphism $(\Gamma.A, h_{\Gamma.A}) \longrightarrow (\Gamma, h_\Gamma)$, by definition of $h_{\Gamma.A}$.

Construction 42 (Phase comprehensions in $\mathbf{H}_\diamond$)**.** *Given $\Gamma \in \mathbf{H}_\diamond$ and $p \in \mathbb{P}$, we must exhibit a map $h_{\Gamma.\mathbf{A}\langle p\rangle} : (\!|\Gamma.\mathbf{A}\langle p\rangle|\!) \longrightarrow [\![\Gamma.\mathbf{A}\langle p\rangle]\!]$. Using Lemma 39, we construct this by combining the assumed map $h_\Gamma : (\!|\Gamma|\!) \longrightarrow [\![\Gamma]\!]$ with the (necessarily unique) map $(\!|\mathbf{A}\langle p\rangle|\!) \longrightarrow [\![\mathbf{A}\langle p\rangle]\!]$ obtained from the identity map on $\mathbf{A}\langle p\rangle$ in the following way:*

(1) *First we observe that* $[\![\mathbf{A}\langle p\rangle]\!] \cong \boldsymbol{j}_*\mathbf{I}\langle p\rangle$ *as follows:*

$$\begin{aligned}
[\![\mathbf{A}\langle p\rangle]\!] &= I_{\mathbf{Sh}\,\mathbf{G}}\alpha_\diamond\mathbf{A}\langle p\rangle && \textit{by definition}\\
&\cong I_{\mathbf{Sh}\,\mathbf{G}}\mathbf{I}\langle p\rangle && \textit{α is a homomorphism}\\
&\cong \boldsymbol{j}_*\mathbf{I}\langle p\rangle && \textit{by definition}
\end{aligned}$$

(2) *Then we proceed by adjoint calisthenics:*

$$\begin{aligned}
\hom_{\mathbf{Sh}\,\mathbf{G}}(\langle\!|\mathbf{A}\langle p\rangle|\!\rangle, [\![\mathbf{A}\langle p\rangle]\!]) &\cong \hom_{\mathbf{Sh}\,\mathbf{G}}(\langle\!|\mathbf{A}\langle p\rangle|\!\rangle, \boldsymbol{j}_*\mathbf{I}\langle p\rangle) && \textit{by the above}\\
&\cong \hom_{\mathbf{Sh}\,\mathbf{I}}(\boldsymbol{j}^*\langle\!|\mathbf{A}\langle p\rangle|\!\rangle, \mathbf{I}\langle p\rangle) && \textit{by adjoint transpose}\\
&\cong \hom_{\mathbf{Sh}\,\mathbf{I}}(\alpha_!\mathbf{A}\langle p\rangle, \mathbf{I}\langle p\rangle) && \textit{by definition of } \langle\!|-|\!\rangle\\
&\cong \hom_{\mathbf{Sh}\,\mathbf{A}}(\mathbf{A}\langle p\rangle, \alpha^*\mathbf{I}\langle p\rangle) && \textit{by adjoint transpose}\\
&\cong \hom_{\mathbf{Sh}\,\mathbf{A}}(\mathbf{A}\langle p\rangle, \mathbf{A}\langle p\rangle) && \textit{by Lemma 40}
\end{aligned}$$

Construction 43 (The presheaf of terms)**.** *We define* $\mathbf{H}(\mathsf{tm}) \in \mathbf{Pr}\,\mathbf{H}_\diamond \downarrow \mathbf{H}(\mathsf{tp})$ *to send* $A \in \mathbf{H}(\mathsf{tp})(\Gamma, h_\Gamma)$ *to the set of sections of the projection* $p_A : (\Gamma.A, h_{\Gamma.A}) \longrightarrow (\Gamma, h_\Gamma)$ *in* $\mathbf{H}_\diamond$*.*

Construction 44 (The hydration model)**.** *With the definitions that we have given, the projection functor* $\psi_\diamond : \mathbf{H}_\diamond \longrightarrow \mathbf{A}_\diamond$ *easily extends to a morphism of models, exhibiting* $\mathbf{H}$ *as a atomic substitution model over* $\mathbf{A}$*.*

Construction 45 (The hydration map)**.** *As we may compose* $\psi : \mathbf{H} \longrightarrow \mathbf{A}$ *with the structure map* $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$*, we can view* $\mathbf{H}$ *as a atomic substitution model over* $\mathbf{I}$ *as well, and thus by the universal property of* $\mathbf{A}$ *we have a universal section* $\mathbf{A} \longrightarrow \mathbf{H}$ *to* $\psi : \mathbf{H} \longrightarrow \mathbf{A}$*. Unraveling the section* $\mathbf{A} \longrightarrow \mathbf{H}$ *we obtain precisely a natural assignment of hydration map component* $h_\Gamma : \langle\!|\Gamma|\!\rangle \longrightarrow [\![\Gamma]\!]$ *from which we may assemble a single hydration map* $\nearrow : \langle\!|-|\!\rangle \longrightarrow [\![-]\!]$ *in* $\mathbf{Cat} \downarrow \mathbf{Sh}\,\mathbf{I}$*.*

6.8.4 Normalization and decidability. We can now show how to compute the normal form of a type $\Gamma \vdash A\ \mathit{type}$, which we regard as an arrow $A : \alpha_\diamond\Gamma \longrightarrow \mathsf{tp}$ in $\mathbb{T}$. Then we may apply the functorial action of the evaluation functor $I_{\mathbf{Sh}\,\mathbf{G}} : \mathbb{T} \longrightarrow \mathbf{Sh}\,\mathbf{G}$ to obtain $[\![A]\!] : [\![\Gamma]\!] \longrightarrow \mathsf{tp}^*$ and then postcompose with the projection of normal forms to obtain $[\![A]\!].\mathsf{code} : [\![\Gamma]\!] \longrightarrow \mathsf{nftp}$. Unraveling the meaning of such a map in the gluing category $\mathbf{Sh}\,\mathbf{G}$, we see that this amounts *not* to a normal form for A but instead to an assignment of normal forms of A to computability witnesses for all the variables in the context Γ. It is precisely this gap that hydration fills:

$$\mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash A\ \mathit{type}) = \langle\!|\Gamma|\!\rangle \xrightarrow{\nearrow_\Gamma} [\![\Gamma]\!] \xrightarrow{[\![A]\!]} \mathsf{tp}^* \xrightarrow{-.\mathsf{code}} \mathsf{nftp}$$

The entire map above restricts within object-space, by construction, to the original type A, or (to be more precise) its image in $\mathbf{Sh}\,\mathbf{G}$ under $\boldsymbol{j}_* : \mathbf{Sh}\,\mathbf{I} \hookrightarrow \mathbf{Sh}\,\mathbf{G}$. The meta-space component of such a morphism $\langle\!|\Gamma|\!\rangle \longrightarrow \mathsf{nftp}$ is precisely a normal form for A.

Theorem 46. *Normalization is sound and complete:*

(1) ***Soundness.*** *If* $\mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash A\ \mathit{type}) = \mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash B\ \mathit{type})$*, then* $\Gamma \vdash A = B$*.*
(2) ***Completeness.*** *If* $\Gamma \vdash A = B$*, then* $\mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash A\ \mathit{type}) = \mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash B\ \mathit{type})$*.*

Proof. Completeness holds *by definition*, as the normalization function is defined on the denizens of the syntactic CwR rather than on raw terms. Soundness follows from the fact that $\mathsf{norm}_{\mathsf{tp}}(\Gamma \vdash A\ \mathit{type})$ restricts in object-space to A itself. □

In the same way, we can construct a normalization function for terms and prove that it is sound and complete (though we do not do so here). Of course, deciding equality for terms is practically important only insofar as it arises in the context of deciding equality for the types that mention them.

Definition 47. *An object $X \in \mathbf{Sh}\,\mathbf{G}$ has **levelwise decidable equality** when for each $\Gamma \in \mathbf{A}_\diamond$, the set $(\boldsymbol{i}^* X)\Gamma$ has decidable equality where $\boldsymbol{i}\colon \mathbf{A} \hookrightarrow \mathbf{G}$ is as in Definition 19.*

Theorem 48. *Viewed as objects of $\mathbf{Sh}\,\mathbf{G}$, the following have levelwise decidable equality:*

$$\mathsf{nftp} \qquad (A : \mathsf{tp}) \times \mathsf{nf}\, A \qquad (A : \mathsf{tp}) \times (\phi : \Omega_{dec}) \times \mathsf{ne}_\phi A$$

From this, we obtain our main results concerning $\mathbf{TT}_{\mathbb{P}}$:

Corollary 49. *Definitional equality in $\mathbf{TT}_{\mathbb{P}}$ is decidable.*

6.8.5 Stronger normalization results. A few stronger results can be proved using routine extensions of the methods on display here.

(1) *The external normalization function is surjective, which implies that normalization is idempotent.* The main practical impact of normalization being surjective is to prove that the normalization function is effectively computable, as in Sterling [39], Sterling and Angiuli [41]; this step is redundant in the setting of the present paper, which has been carried out constructively in order to ensure an implicit form of effective computability.
(2) *Type constructors are injective in the sense that $\Gamma \vdash A \to B = A' \to B'$ implies $\Gamma \vdash A = A'$ and $\Gamma \vdash B = B'$*, etc. Injectivity of type constructors is the main ingredient to establishing determinacy of the standard *bidirectional elaboration* algorithm. Injectivity is not strictly needed for a type checker written on fully annotated terms, but practical systems involve the elaboration of less-annotated terms to fully annotated terms; this process relies on injectivity.
(3) *Normalization can be internalised into $\mathbf{Sh}\,\mathbf{G}$ as an inverse $\mathsf{tp} \longrightarrow \mathsf{nftp}$ to the projection $\mathsf{nftp} \to \mathsf{tp}$, as in Sterling [39, 40], Sterling and Angiuli [41].* This implies, for example, that the normalization function is invariant under variable renaming (or, more generally, atomic substitutions).

We do not detail these results here, but instead remark that detailed proofs for similar theories can be found in the cited literature.

7 RELATED WORK

Proof assistants already have support for various means of controlling the unfolding of definitions; we classify these as either *library-* or *language-level*.

7.1 Library-level features

Various library-level idioms for abstract definitions are used in practice such as SSReflect's `lock` idiom. While such approaches are flexible and compatible with existing proof assistants, they are often cumbersome in practice. For instance, `lock` relies on various tactics with subtle behavior, which makes it difficult to use `locking` idioms in pure Gallina code.

7.2 Language-level features

Many proof assistants include a feature like Agda's `abstract` blocks which marks a definition as completely opaque to the remainder of the development. In Remark 1, we explained how to recover Agda's `abstract` definitions using controlled unfolding. Moreover, as controlled unfolding does not require a user to decide up front whether a definition can be unfolded, it gives a more realistic and flexible discipline for abstraction in a proof assistant. In practice, however, `abstract` is often used for performance reasons instead of merely for controlling abstraction; unfolding large or complex definitions can significantly slow down type checking and unification. While we have not discussed performance considerations for controlled unfolding, the same optimizations apply

to our mechanism for definitions that are never unfolded. In total, controlled unfolding strictly generalizes `abstract` blocks.

Recently, Kovács [29, 30] has proposed a *glued evaluation* technique to both improve the pretty-printing of goals and more efficiently handle unfolding during conversion testing. Roughly, the proof assistant's kernel may choose to unfold any definition, but avoids doing so whenever possible for efficiency and strives to never show unfolded goals to the user. Both glued evaluation and controlled unfolding relate to the unfolding of definitions, they are largely orthogonal and complementary. In particular, glued evaluation does not require user intervention, unlike controlled unfolding, but it does not actually preclude any unfolding from taking place. Thus, glued evaluation does not impact the well-formedness of a program and can be used as a "drop-in" technique for improving performance and usability. However, for the same reasons, glued evaluation cannot be used to enforce modularity and independence in the same way as controlled unfolding. Ideally, a proof assistant would support both glued evaluation and controlled unfolding: the more advanced evaluation algorithm would improve baseline performance and controlled unfolding would facilitate users enforcing stronger abstraction boundaries within their programs and assisting the kernel by manually designating certain definitions as opaque.

Program verifiers such as VeriFast and Chalice include similar unfolding mechanisms to cope specifically with recursive definitions [25, 44]. Like our mechanism, these features allow users fine-grained control over how definitions are unfolded. However, these verifiers work only within simply-typed theories and thus avoid the substantial complexity of dependency. Moreover, these mechanisms manage a different problem than controlled unfolding; they allow a user to unfold recursive definitions step-by-step while controlled unfolding is used to control when each definition can be fully inlined.

7.3 Translucent ascription in module systems

Thus far we have focused on proof assistants, but similar considerations arise for ML-style module systems [12, 22, 34, 42]. The default opacity for definitions in module systems is the same as in controlled unfolding and opposite to proof assistants: types are abstract unless marked otherwise. The treatment of translucent type declarations in module systems [22] relies on *singleton kinds* [4, 43], which are the special case of extension types whose boundary proposition is $\top$. Generalizing from compiletime kinds to mixed compiletime–runtime *module signatures*, Sterling and Harper have pointed out that transparent ascriptions are best handled by an extension type whose boundary proposition represents the compiletime phase itself [42]. Thus the translucency of compiletime module components can be seen as a *particular* controlled unfolding policy in the sense of this paper.

7.4 Controlled unfolding in Agda

Inspired by our implementation of controlled unfolding in `cooltt`, Amélia Liao and Jesper Cockx have implemented a version of this mechanism called `opaque` within Agda 2.6.4 [32]. However, rather than using extension types, their Agda implementation simulates the necessary behaviors by instrumenting conversion checking—a workaround made possible by the very restricted ways in which our elaboration procedure relies on extension types. This demonstrates that controlled unfolding can be adapted to proof assistants like Coq whose core calculi do not presently support extension types.

At the time of writing, Agda's `opaque` declarations are new enough that only two major Agda libraries, the 1Lab [45] and the Cubical Agda library [46], use them extensively; the Agda standard library may adopt `opaque` declarations in a future major revision [48]. As of publication, 65 modules

in the 1Lab use `opaque` and 19 use `unfolding`, in addition to over 100 using `abstract` blocks; in the Cubical Agda library, 27 modules use `opaque`, 6 use `unfolding`, and 35 use `abstract`.

8 CONCLUSIONS AND FUTURE WORK

We have proposed *controlled unfolding*, a new mechanism for interpolating between transparent and opaque definitions in proof assistants. We have demonstrated its practical applicability by extending `cooltt` with controlled unfolding; we have also proved its soundness through an elaboration algorithm to a core calculus whose normalization we establish using a constructive synthetic Tait computability argument.

In the future, we hope to see controlled unfolding integrated into more proof assistants and to further explore its applications for large-scale organization of mechanized mathematics. As mentioned above, some our mechanism has implemented in Agda, but features such as local unfolds are still absent. Furthermore, in the context of our `cooltt` implementation, we have also already begun to experiment with potential extensions, including one that allows a *subterm* to be declared locally abstract and then unfolded later on as-needed — a more flexible alternative to Coq's `abstract` *t* tactical. As we mentioned in Remark 1, we also are interested in facilities to limit the scope in which it is possible to unfold a definition.

ACKNOWLEDGMENTS

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation. Jonathan Sterling is funded by the European Union under the Marie Skłodowska-Curie Actions Postdoctoral Fellowship project *TypeSynth: synthetic methods in program verification*, and by the United States Air Force Office of Scientific Research under grant number FA9550-23-1-0728 (*New Spaces for Denotational Semantics*; Dr. Tristan Nguyen, Program Manager). Carlo Angiuli is supported by the U.S. Air Force Office of Scientific Research under grant numbers FA9550-21-1-0009 and FA9550-24-1-0350. 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 European Union, the European Commission, or the AFOSR. Neither the European Union nor the granting authority can be held responsible for them.

COMPETING INTERESTS

The authors declare no competing interests.

REFERENCES

[1] **Anel, M. and Joyal, A.** 2021. Topo-logie. In **Anel, M. and Catren, G.**, editors, *New Spaces in Mathematics: Formal and Conceptual Reflections*, volume 1, chapter 4, pp. 155–257. Cambridge University Press.

[2] **Angiuli, C.**, **Brunerie, G.**, **Coquand, T.**, **Hou (Favonia), K.-B.**, **Harper, R.**, **and Licata, D. R.** 2021. Syntax and models of Cartesian cubical type theory. *Mathematical Structures in Computer Science*, 31(4):424–468.

[3] **Angiuli, C.**, **Hou (Favonia), K.-B.**, **and Harper, R.** 2018. Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. In **Ghica, D. and Jung, A.**, editors, *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, volume 119 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 6:1–6:17, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] **Aspinall, D.** 1995. Subtyping with singleton types. In **Pacholski, L. and Tiuryn, J.**, editors, *Computer Science Logic*, pp. 1–15. Springer Berlin Heidelberg.

[5] **Awodey, S.** 2018. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286.

[6] **Bocquet, R.**, **Kaposi, A.**, **and Sattler, C.** 2021. Relative induction principles for type theories. Unpublished manuscript. https://arxiv.org/abs/2102.11649.

[7] **Cartmell, J.** 1978. *Generalised Algebraic Theories and Contextual Categories.* PhD thesis, University of Oxford.

[8] **Clairambault, P. and Dybjer, P.** 2014. The biequivalence of locally cartesian closed categories and Martin-Löf type theories. *Mathematical Structures in Computer Science*, 24(6).

[9] **Cohen, C., Coquand, T., Huber, S., and Mörtberg, A.** 2017. Cubical Type Theory: a constructive interpretation of the univalence axiom. *IfCoLog Journal of Logics and their Applications*, 4(10):3127–3169.
[10] **Coquand, T.** 1996. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177.
[11] **Dagand, P.-E.** 2013. *A Cosmology of Datatypes: Reusability and Dependent Types*. PhD thesis, University of Strathclyde, Glasgow, Scotland.
[12] **Dreyer, D., Crary, K., and Harper, R.** 2003. A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pp. 236–249, New Orleans, Louisiana, USA. Association for Computing Machinery.
[13] **Dybjer, P.** 1996. Internal type theory. In **Berardi, S. and Coppo, M.**, editors, *Types for Proofs and Programs*, pp. 120–134, Berlin, Heidelberg. Springer Berlin Heidelberg.
[14] **Fiore, M.** 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, pp. 26–37. ACM.
[15] **Fiore, M. P.** 2022. Semantic analysis of normalisation by evaluation for typed lambda calculus. Unpublished extended version of PPDP '02 paper with same title, available at https://arxiv.org/abs/2207.08777.
[16] **Freyd, P.** 1972. Aspects of topoi. *Bulletin of the Australian Mathematical Society*, 7(1):1–76.
[17] **Gilbert, G., Cockx, J., Sozeau, M., and Tabareau, N.** 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.*, 3(POPL).
[18] **Gonthier, G., Mahboubi, A., and Tassi, E.** 2016. A small scale reflection extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France.
[19] **Gratzer, D., Shulman, M., and Sterling, J.** 2022. Strict universes for Grothendieck topoi. Unpublished manuscript. https://arxiv.org/abs/2202.12012.
[20] **Gratzer, D. and Sterling, J.** 2020. Syntactic categories for dependent type theory: sketching and adequacy. Unpublished manuscript. https://arxiv.org/abs/2012.10783.
[21] **Gratzer, D., Sterling, J., and Birkedal, L.** 2019. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.*, 3.
[22] **Harper, R. and Stone, C.** 2000. A type-theoretic interpretation of Standard ML. In **Plotkin, G., Stirling, C., and Tofte, M.**, editors, *Proof, Language, and Interaction*, pp. 341–387. MIT Press, Cambridge, MA, USA.
[23] **Hou (Favonia), K.-B.** 2022. `kado`. http://www.github.com/RedPRL/kado.
[24] **Huber, S.** 2019. Canonicity for cubical type theory. *Journal of Automated Reasoning*, 63(2):173–210.
[25] **Jacobs, B., Vogels, F., and Piessens, F.** 2015. Featherweight VeriFast. *Logical Methods in Computer Science*, Volume 11, Issue 3.
[26] **Johnstone, P. T.** 1977. *Topos Theory*. Academic Press.
[27] **Jung, A. and Tiuryn, J.** 1993. A new characterization of lambda definability. In **Bezem, M. and Groote, J. F.**, editors, *Typed Lambda Calculi and Applications*, pp. 245–257, Berlin, Heidelberg. Springer Berlin Heidelberg.
[28] **Kaposi, A., Kovács, A., and Altenkirch, T.** 2019. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24.
[29] **Kovács, A.** 2023. `smalltt`.
[30] **Kovács, A.** 2024. Efficient elaboration with controlled definition unfolding. In *Third Workshop on the Implementation of Type Systems*.
[31] **Lack, S.** 2009. *A 2-Categories Companion*, 105–191. Springer New York.
[32] **Liao, A. and Cockx, J.** 2022. Unfolding control for abstract blocks. https://github.com/agda/agda/pull/6354.
[33] **Martin-Löf, P.** 1975. An intuitionistic theory of types: predicative part. In **Rose, H. and Shepherdson, J.**, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pp. 73–118. North-Holland.
[34] **Milner, R., Tofte, M., Harper, R., and MacQueen, D.** 1997. *The Definition of Standard ML (Revised)*. MIT Press.
[35] **Newstead, C.** 2018. *Algebraic models of dependent type theory*. Phd thesis, Carnegie Mellon University.
[36] **Pierce, B. C. and Turner, D. N.** 2000. Local type inference. *ACM Transactions Programming Language and Systems*, 22(1):1–44.
[37] **RedPRL Development Team, T.** 2020. `cooltt`. http://www.github.com/RedPRL/cooltt.
[38] **Riehl, E. and Shulman, M.** 2017. A type theory for synthetic ∞-categories. *Higher Structures*, 1(1):147–224.
[39] **Sterling, J.** 2021. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University. Version 1.1, revised May 2022.
[40] **Sterling, J.** 2025. *Toward a Geometry for Syntax*, pp. 391–432. Springer Nature Switzerland.
[41] **Sterling, J. and Angiuli, C.** 2021. Normalization for cubical type theory. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '21, New York, NY, USA. ACM.
[42] **Sterling, J. and Harper, R.** 2021. Logical relations as types: Proof-relevant parametricity for program modules. *Journal of the ACM*, 68(6).

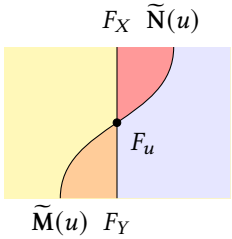
[43] **Stone, C. A. and Harper, R.** 2006. Extensional equivalence and singleton types. *Transactions on Computational Logic*, 7(4):676–722.
[44] **Summers, A. J. and Drossopoulou, S.** 2013. A formal semantics for isorecursive and equirecursive state abstractions. In **Castagna, G.**, editor, *ECOOP 2013 – Object-Oriented Programming*, pp. 129–153, Berlin, Heidelberg. Springer Berlin Heidelberg.
[45] **The 1Lab Development Team** 2022. The 1Lab. https://1lab.dev.
[46] **The Agda Community** 2023. The Cubical Agda library. https://github.com/agda/cubical/.
[47] **The Agda Development Team** 2022. The Agda standard library. https://github.com/agda/agda-stdlib.
[48] **The Agda Development Team** 2023. Consider where we can use opaque mechanism to provide abstraction. https://github.com/agda/agda-stdlib/issues/2136.
[49] **The Agda Team** 2021. *Agda User Manual, Release 2.6.2*.
[50] **The Coq Development Team** 2022. The Coq proof assistant.
[51] **Uemura, T.** 2021. *Abstract and Concrete Type Theories*. PhD thesis, Institute for Logic, Language and Computation, University of Amsterdam.
[52] **Uemura, T.** 2023. A general framework for the semantics of type theory. *Mathematical Structures in Computer Science*, 33(3):134–179.
[53] **Univalent Foundations Program, T.** 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.
[54] **Vickers, S.** 2007. Locales and toposes as spaces. In **Aiello, M.**, **Pratt-Hartmann, I.**, **and Van Benthem, J.**, editors, *Handbook of Spatial Logics*, pp. 429–496. Springer Netherlands, Dordrecht.

A THE (2,1)-CATEGORY OF MODELS

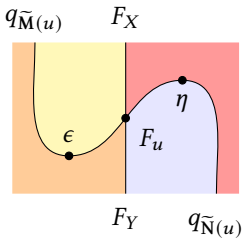
Uemura [51] has observed that a model $(\mathbf{M}_\diamond, \mathbf{M})$ in the sense of Definition 11 can be packaged into a single functor $\widetilde{\mathbf{M}} : \mathbb{T}^\triangleright \longrightarrow \mathbf{Cat}$, in which $\mathbb{T}^\triangleright$ freely extends $\mathbb{T}$ by a new terminal object $\diamond$ and $\mathbf{Cat}$ is the 2-category of categories. From this perspective, a sort $X \in \mathbb{T}$ is taken to the total category $\widetilde{\mathbf{M}}(X) = \int_{\mathbf{M}_\diamond} \mathbf{M}(X)$ of a discrete fibration over $\widetilde{\mathbf{M}}(\diamond) = \mathbf{M}_\diamond$. Here we are using the equivalence between $\mathbf{DFib}_C \simeq \mathbf{Pr}\, C$. The preservation of representable maps is then rendered here as the requirement that for representable $u : X \longrightarrow Y$, each functor $\widetilde{\mathbf{M}}(u) : \widetilde{\mathbf{M}}(X) \longrightarrow \widetilde{\mathbf{M}}(Y)$ shall have a right adjoint $\widetilde{\mathbf{M}}(u) \dashv q_{\widetilde{\mathbf{M}}(u)}$ taking an element of $\widetilde{\mathbf{M}}(Y)$ to the *generic* element of $\widetilde{\mathbf{M}}(X)$ in the extended context.

Example 50. *For the representable map $\pi : \mathsf{tm} \longrightarrow \mathsf{tp}$, the functorial action $\widetilde{\mathbf{M}}(\pi) : \widetilde{\mathbf{M}}(\mathsf{tm}) \longrightarrow \widetilde{\mathbf{M}}(\mathsf{tp})$ takes a term $\Gamma \vdash a : A$ to the type $\Gamma \vdash A$; the right adjoint $q_{\widetilde{\mathbf{M}}(\pi)} : \widetilde{\mathbf{M}}(\mathsf{tp}) \longrightarrow \widetilde{\mathbf{M}}(\mathsf{tm})$ sends a type $\Gamma \vdash A$ to the variable $\Gamma, a : A \vdash a : A$.*

Definition 51. *Given two models $\mathbf{M}, \mathbf{N}$ of $\mathbb{T}$, a **morphism of models** from $\mathbf{M}$ to $\mathbf{N}$ is given by a natural transformation $F : \widetilde{\mathbf{M}} \longrightarrow \widetilde{\mathbf{N}} \in [\mathbb{T}^\triangleright, \mathbf{Cat}]$ such that for each representable map $u : X \longrightarrow Y$ in $\mathbb{T}$ the corresponding naturality datum $F_u : \widetilde{\mathbf{N}}(u) \circ F_X = F_Y \circ \widetilde{\mathbf{M}}(u)$ depicted below*

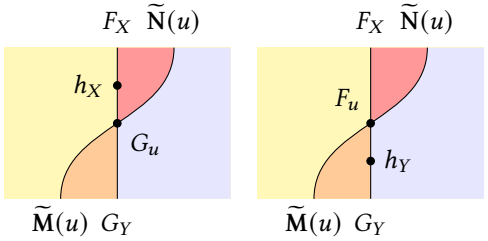


satisfies the Beck–Chevalley condition in the sense the following 2-cell, obtained by conjugating with units and counits, denotes an invertible natural transformation $F_X \circ q_{\widetilde{\mathbf{M}}(u)} \longrightarrow q_{\widetilde{\mathbf{N}}(u)} \circ F_Y$:

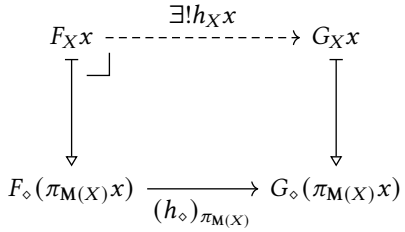


Definition 52. *Let $\mathbf{M}, \mathbf{N}$ be two models of $\mathbb{T}$, and let $F, G : \mathbf{M} \longrightarrow \mathbf{N}$ be two morphisms of models. An **isomorphism** h from F to G is defined to be an* invertible modification *between the underlying natural transformations F, G. This amounts to choosing for each $X \in \mathbb{T}^\triangleright$ a natural isomorphism $h_X : F_X \longrightarrow G_X$ in $[\widetilde{\mathbf{M}}(X), \widetilde{\mathbf{N}}(X)]$, subject to the coherence condition that for each $u : X \longrightarrow Y$ in $\mathbb{T}^\triangleright$*

the following two wiring diagrams are equal:



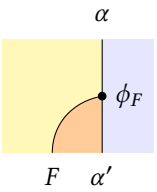
Remark 53. *Because each of the induced maps $\pi_{\mathbf{M}(X)} : \widetilde{\mathbf{M}}(X) \longrightarrow \mathbf{M}_\diamond$ and $\pi_{\mathbf{N}(X)} : \widetilde{\mathbf{N}}(X) \longrightarrow \mathbf{N}_\diamond$ into the cone point are discrete fibrations, it suffices to check the modification condition of h on only the cone maps $X \longrightarrow \diamond$: as any discrete fibration is a faithful functor, it moreover follows that $h_\diamond : F_\diamond \longrightarrow G_\diamond$ uniquely determines all the other h_X if they exist. Unfolding further, given $x \in \widetilde{\mathbf{M}}(X)$ we are only requiring that $(h_\diamond)^*_{\pi_{\mathbf{M}(X)}}(G_X x) = F_X x$ in the sense depicted below in the discrete fibration $\mathbf{N}(X)$ over $\mathbf{N}_\diamond$:*



Thus we have a (2,1)-category of models $\mathbf{Mod}\,\mathbb{T}$ for any category with representable maps $\mathbb{T}$.

B THE (2,1)-CATEGORY OF ATOMIC SUBSTITUTION MODELS

Definition 54. *Given atomic substitution models $\alpha : \mathbf{A} \longrightarrow \mathbf{I}$ and $\alpha' : \mathbf{A}' \longrightarrow \mathbf{I}$, a morphism from $(\mathbf{A}, \alpha)$ to $(\mathbf{A}', \alpha')$ is given by a morphism $F : \mathbf{A} \longrightarrow \mathbf{A}' \in \mathbf{Mod}\,\mathbb{T}_0$ together with an isomorphism $\phi_F : \alpha \longrightarrow \alpha' \circ F$ in $[\mathbf{A}, \mathbf{I}]$ as depicted below:*



Definition 55. *Given two morphisms $F, G : (\mathbf{A}, \alpha) \longrightarrow (\mathbf{A}', \alpha')$, an isomorphism from F to G is given by an isomorphism $h : F \longrightarrow G \in [\mathbf{A}, \mathbf{A}']$ such that the following wiring diagrams denote equal isomorphisms $\alpha \longrightarrow \alpha' \circ G$:*