Controlling unfolding in type theory using extension types

Anonymous Authors

***Abstract*—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.**

I. Introduction

In dependent type theory, terms are type checked modulo definitional equality, a congruence generated by α-, β-, and η-laws, as well 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` [1] and Coq's `Qed` [2].

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 "`lock`ing" idiom for controlling when definitions unfold [3]. 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 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 are more subtle than they 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.

Contributions

We propose a novel 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* [5], a connective first introduced in the context of homotopy type theory. 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 [6].

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[2] than Coq's unfolding tactics because they are implemented by straightforward elaboration into the core calculus (via new types and declaration forms); we anticipate that this *type-theoretic* account of controlled unfolding will also provide a clear path toward integrating our ideas with future language features.

[1] Indeed, the Agda standard library [4] currently uses `abstract` only once.
[2] https://github.com/coq/coq/blob/V8.16.0/theories/ssr/ssreflect.v/#L388

Our core calculus is intensional Martin-Löf type theory extended with proof-irrelevant proposition symbols p, dependent products $\{p\}\,A$ over those propositions, and extension types $\{A \mid p \hookrightarrow a\}$ whose elements are the elements of A that are definitionally equal to a under the assumption that p is true. Extension types are similar to the path types ($\mathsf{Path}\ A\ a_0\ a_1$) of cubical type theory [7]–[9], 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 $0, 1 : \mathbb{I}$.

To justify our elaboration algorithm, 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. Our proof adapts and extends Sterling's technique of synthetic Tait computability (STC) [10], [11], which has previously been used to establish parametricity for an ML-style module calculus [10] and normalization for cubical type theory [12] and multimodal type theory [13]. Our proof is fully constructive, an improvement on the prior work of Sterling and Angiuli [12]; we have also corrected an error in the handling of universes in an earlier revision of Sterling's doctoral dissertation [11].

Outline

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

II. 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 IV 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:

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

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

B. 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:

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

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

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

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

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:

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

cong f refl = refl

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

C. 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 : $A \to$ vec (su ze) A
singleton $a = a :: []$

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

abbrv-example : singleton $5 \equiv (5 :: [])$
abbrv-example = refl

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

abbreviation map-$\oplus$
map-$\oplus$ **unfolds** map; $(\oplus)$
...

We cannot unfold map-$\oplus$ in all contexts, because its definition is only well-typed when map and $(\oplus)$ are unfolded. The meaning of this declaration must, therefore, be that map-$\oplus$ shall be unfolded *just as soon as* map and $(\oplus)$ 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.

D. 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 II-E we will devise a more general and ergonomic solution.

Consider the left-unit law for $(\oplus)$: 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 $(\oplus)$ 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 $(\oplus)$, we start by defining its *type* as an abbreviation that unfolds $(+)$:

abbreviation $\oplus$-left-unit-type
$\oplus$-left-unit-type **unfolds** $(+)$
$\oplus$-left-unit-type : vec n $A \to$ Type
$\oplus$-left-unit-type $u = [] \oplus u \equiv u$

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

$\oplus$-left-unit : $(u$: vec n $A) \to \oplus$-left-unit-type u
$\oplus$-left-unit u = ? : $\oplus$-left-unit-type u

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

$\oplus$-left-unit **unfolds** $(+)$
$\oplus$-left-unit : $(u$: vec n $A) \to \oplus$-left-unit-type u
$\oplus$-left-unit u = ? : $[] \oplus u \equiv u$

We complete the proof by unfolding $(\oplus)$ itself, which transitively unfolds $(+)$:

$\oplus$-left-unit **unfolds** $(\oplus)$
$\oplus$-left-unit : $(u$: vec n $A) \to \oplus$-left-unit-type u
$\oplus$-left-unit u = refl

E. 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 II-D, 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 II-D, but using **unfold** rather than an auxiliary definition:

$\oplus$-left-unit : $(u$: vec n $A) \to$ **unfold** $(+)$ **in** $[] \oplus u \equiv u$
$\oplus$-left-unit u = ? : **unfold** $(+)$ **in** $[] \oplus u \equiv u$

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

$\oplus$-left-unit **unfolds** $(+)$
$\oplus$-left-unit : $(u$: vec n $A) \to$ **unfold** $(+)$ **in** $[] \oplus u \equiv u$
$\oplus$-left-unit u = ? : $[] \oplus u \equiv u$

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

$\oplus$-left-unit **unfolds** $(\oplus)$
$\oplus$-left-unit : $(u : \mathsf{vec}\ n\ A) \to$ **unfold** $(+)$ **in** $[] \oplus u \equiv u$
$\oplus$-left-unit $u =$ refl

The meaning of the code above is exactly as described in Section II-D: 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 $\oplus$-left-unit above that we could have replaced the top-level **unfolds** $(\oplus)$ directive of $\oplus$-left-unit with the new expression-level **unfold** $(\oplus)$ **in** as follows:

$\oplus$-left-unit$'$: $(u : \mathsf{vec}\ n\ A) \to$ **unfold** $(+)$ **in** $[] \oplus u \equiv u$
$\oplus$-left-unit$'$ $u =$ **unfold** $(\oplus)$ **in** refl

The resulting definition of $\oplus$-left-unit$'$ has slightly different behavior than $\oplus$-left-unit above: whereas unfolding $\oplus$-left-unit causes $(\oplus)$ to unfold transitively, we can unfold $\oplus$-left-unit$'$ without unfolding $(\oplus)$—at the cost of **unfold** $(\oplus)$ 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 illustrate the elimination of expression-level unfolding from the definition of $\oplus$-left-unit$'$:

abbreviation $\oplus$-left-unit$'$-type
$\oplus$-left-unit$'$-type **unfolds** $(+)$
$\oplus$-left-unit$'$-type : vec n $A \to$ Type
$\oplus$-left-unit$'$-type $u = [] \oplus u \equiv u$

abbreviation $\oplus$-left-unit$'$-body
$\oplus$-left-unit$'$-body **unfolds** $(\oplus)$
$\oplus$-left-unit$'$-body : $(u : \mathsf{vec}\ n\ A) \to$ $\oplus$-left-unit$'$-type u
$\oplus$-left-unit$'$-body $u =$ refl

$\oplus$-left-unit$'$: $(u : \mathsf{vec}\ n\ A) \to$ $\oplus$-left-unit$'$-type u
$\oplus$-left-unit$'$ $u =$ $\oplus$-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.

III. Controlling unfolding with extension types

Having introduced our new surface language constructs for controlled unfolding in Section II, 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 IV.

A. A core calculus with proposition symbols

Our core calculus parameterizes intensional Martin-Löf type theory (**MLTT**) [14] 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:

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

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. 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 [15, §3.3] or strict propositions [16]. 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 $a = a_p : A$.

B. 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 II, illustrating how they are elaborated into our core calculus:

Plain definitions

Recall our unadorned definition of $(+)$ from Section II:

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

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

prop $\boldsymbol{\Upsilon}_+ \leq \top$

$\boldsymbol{\delta}_+ : \{\top\}\,(m\,n : \mathbb{N}) \to \mathbb{N}$
$\boldsymbol{\delta}_+$ ze $n = n$
$\boldsymbol{\delta}_+$ (su m) $n =$ su $(\boldsymbol{\delta}_+\ m\ n)$

$\mathbf{const}\ (+) : \{\mathbb{N} \to \mathbb{N} \to \mathbb{N} \mid \boldsymbol{\Upsilon}_+ \hookrightarrow \boldsymbol{\delta}_+\}$

Remark 3: In a serious implementation, it would be simple to induce $\boldsymbol{\delta}_+$ 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 II-A):

$(\oplus)$ **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 $\boldsymbol{\Upsilon}_\oplus$ to entail $\boldsymbol{\Upsilon}_+$, capturing the idea that unfolding $(\oplus)$ always causes $(+)$ to unfold; in order to cause $(+)$ to unfold in the body of $(\oplus)$, we assume $\boldsymbol{\Upsilon}_+$ in the definition of $\boldsymbol{\delta}_\oplus$. In full, we elaborate the definition of $(\oplus)$ as follows:

$\mathbf{prop}\ \boldsymbol{\Upsilon}_\oplus \leq \boldsymbol{\Upsilon}_+$

$\boldsymbol{\delta}_\oplus : \{\boldsymbol{\Upsilon}_+\}\,(u : \mathsf{vec}\ m\ A)\,(v : \mathsf{vec}\ n\ A) \to \mathsf{vec}\,(m+n)\ A$
$\boldsymbol{\delta}_\oplus\ []\ v = v$
$\boldsymbol{\delta}_\oplus\ (a :: u)\ v = a :: (\boldsymbol{\delta}_\oplus\ u\ v)$

$\mathbf{const}\ (\oplus) :$
$\quad \{\mathsf{vec}\ m\ A \to \mathsf{vec}\ n\ A \to \mathsf{vec}\,(m+n)\ A \mid \boldsymbol{\Upsilon}_\oplus \hookrightarrow \boldsymbol{\delta}_\oplus\}$

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

If a definition ϑ unfolds multiple definitions $\kappa_1; \ldots; \kappa_n$, we define $\boldsymbol{\Upsilon}_\vartheta$ to entail (and define $\boldsymbol{\delta}_\vartheta$ to assume) the conjunction $\boldsymbol{\Upsilon}_{\kappa_1} \wedge \cdots \wedge \boldsymbol{\Upsilon}_{\kappa_n}$; if a definition ϑ unfolds no definitions, then $\boldsymbol{\Upsilon}_\vartheta$ entails (and $\boldsymbol{\delta}_\vartheta$ 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 $\boldsymbol{\Upsilon}_\vartheta$ to *equal* the conjunction $\boldsymbol{\Upsilon}_{\kappa_1} \wedge \cdots \wedge \boldsymbol{\Upsilon}_{\kappa_n}$. For example, consider the following code from Section II-C:

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

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

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

The above example is then elaborated as follows:

$\mathbf{prop}\ \boldsymbol{\Upsilon}_{\text{map-}\oplus} = \boldsymbol{\Upsilon}_{\mathsf{map}} \wedge \boldsymbol{\Upsilon}_\oplus$

$\boldsymbol{\delta}_{\text{map-}\oplus} : \{\boldsymbol{\Upsilon}_{\mathsf{map}} \wedge \boldsymbol{\Upsilon}_\oplus\}\ \mathfrak{C}$
$\boldsymbol{\delta}_{\text{map-}\oplus}\ f\ []\ v = \mathsf{refl}$
$\boldsymbol{\delta}_{\text{map-}\oplus}\ f\ (a :: u)\ v = \mathsf{cong}\ ((f\ a) ::\ )\ (\boldsymbol{\delta}_{\text{map-}\oplus}\ f\ u\ v)$

$\mathbf{const}\ \text{map-}\oplus : \{\mathfrak{C} \mid \boldsymbol{\Upsilon}_{\text{map-}\oplus} \hookrightarrow \boldsymbol{\delta}_{\text{map-}\oplus}\}$

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 II-E; we return to this in Section IV-C.

IV. 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 $\mathbf{TT}_{\mathbb{P}}$.

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

Our core calculus $\mathbf{TT}_{\mathbb{P}}$ is intensional Martin-Löf type theory (**MLTT**) [14] 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 [5].

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 in [17], [18].

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.

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

$$\frac{\Gamma, p \vdash \mathcal{J} \qquad \Gamma \vdash p\ \mathit{true}}{\Gamma \vdash \mathcal{J}}$$

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

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

The dependent product $\{p\}\,A$ is defined as an ordinary dependent product, omitting the β/η rules for lack of space:

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

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

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.

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

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

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

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

B. 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 $\mathbf{TT}_{\mathbb{P}}$.

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

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.

$$\frac{}{\vdash \epsilon\ \mathit{sig} \longrightarrow \{\top\}, \cdot}$$

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

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

C. Bidirectional elaboration

We adopt a bidirectional elaboration algorithm which mirrors bidirectional type-checking algorithms [19], [20]. 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 $\Sigma; \Gamma \vdash \mathtt{e} \Leftarrow A \rightsquigarrow \Sigma', M$ taking a core type as input, and a synthesis judgment $\Sigma; \Gamma \vdash \mathtt{e} \Rightarrow A \rightsquigarrow \Sigma', M$ 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:

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

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:

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

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, *e.g.*, 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**.

$$\frac{\Sigma;\Gamma \vdash \mathsf{e} \Rightarrow A \rightsquigarrow \Sigma_1; M \qquad \Sigma_1;\Gamma \vdash \mathsf{conv}\, A\, B}{\Sigma;\Gamma \vdash \mathsf{e} \Leftarrow B \rightsquigarrow \Sigma_1; M}$$

$$\frac{\Sigma;\Gamma,\boldsymbol{\Upsilon}_\vartheta \vdash \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_1; M \qquad \textbf{let}\ \chi \coloneqq \textit{gensym}\,() \qquad \textbf{let}\ \Sigma_2 \coloneqq \Sigma_1, \textbf{const}\,\chi : \prod_\Gamma \{A \mid \boldsymbol{\Upsilon}_\vartheta \hookrightarrow M\}}{\Sigma;\Gamma \vdash \textbf{unfold}\ \vartheta\ \textbf{in}\ \mathsf{e} \Leftarrow A \rightsquigarrow \Sigma_2; \mathsf{out}_{\boldsymbol{\Upsilon}_\vartheta}\, \chi[\Gamma]}$$

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 $B = D$. We establish both of these essential properties of definitional equality in Section VI.

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 $\{p\}\, A$.

V. Case study: an implementation in `cooltt`

We have implemented our approach to controlled unfolding in the experimental `cooltt` proof assistant [6]; `cooltt` is an implementation of *cartesian cubical type theory* [9], a computational version of homotopy type theory whose syntactic metatheory is particularly well understood [11], [12]. 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`:

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

unfold +
def +0L (x : ℕ) :
  path {+ 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 $\boldsymbol{\Upsilon}_\kappa$ 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 [21], 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.

VI. The metatheory of $\mathbf{TT}_\mathbb{P}$

In Section IV 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 [10]–[12], [22] or STC.

a) Neutral forms for $\mathbf{TT}_\mathbb{P}$: 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 [7], [9] where some equalities apply precisely when two *dimensions* coincide. The same problem arises: either renamings must exclude substitutions which identify two dimension terms, or neutral forms will not be stable under renamings.

In their recent proof of normalization for cubical type theory, Sterling and Angiuli [12] 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 which is true if the neutral is no longer meaningful. Crucially, while 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 show that Sterling and Angiuli's stabilized neutrals can be adapted to $\mathbf{TT}_{\mathbb{P}}$ and use their approach to establish normalization. In so doing, we refine Sterling and Angiuli's approach to obtain a constructive 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 [11].

A. Type theories as categories with representable maps

While any number of logical frameworks are available (generalized algebraic theories [23], essentially algebraic theories [24], locally cartesian closed categories [25], *etc.*), Uemura's *categories with representable maps* [26], [27] 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).

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 [26] 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 [27]. 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.

Theorem 10: Let $\mathbb{T}$ be the free category with representable maps generated by a given logical framework signature; then the groupoid of CwR functors $[\mathbb{T}, \mathcal{E}]_{\mathbf{CwR}}$ 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. Theorem 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 $[\mathbb{T}, \mathcal{E}]_{\mathbf{CwR}}$.

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 [28] or natural models [29], which themselves arrange into a (2,1)-category. The second notion of functorial semantics, developed by Uemura in his doctoral dissertation [27], is a generalization of the theory of CwFs and pseudo-morphisms between them [30], [31].

Note that we may always regard a presheaf category $\mathbf{Pr}(\mathcal{C})$ as a CwR with the representable maps being representable natural transformations, *i.e.* families of presheaves whose fibers at representables are representable [29].

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

Models are arranged into a (2,1)-category $\mathbf{Mod}\,\mathbb{T}$ (see Appendix B). Essentially, a morphism of models $\mathcal{M} \longrightarrow \mathcal{N}$ is given by a functor $\alpha_\diamond : \mathcal{M}_\diamond \longrightarrow \mathcal{N}_\diamond$ together with a natural transformation $\mathcal{M} \longrightarrow \alpha_\diamond^* \mathcal{N} \in [\mathbb{T}, \mathbf{Pr}(\mathcal{M}_\diamond)]$ 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:

Theorem 12: The (2,1)-category of models $\mathbf{Mod}\,\mathbb{T}$ has a bi-initial object $\mathcal{I}$.

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

B. 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 $\mathbb{P}$.

$$\langle p \rangle : \star \qquad \mathsf{tp} : \square \qquad \mathsf{tm} : \mathsf{tp} \Longrightarrow \star$$
$$\_ : \{u, v : \langle p \rangle\} \Longrightarrow u = v$$

$\_ : \{\_ : \langle \bigwedge_{i<n} p_i \rangle\} \Longrightarrow \langle p_k \rangle$
$\_ : \{\_ : \langle p_i \rangle, \ldots\} \Longrightarrow \langle \bigwedge_{i<n} p_i \rangle$

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 $\{\_ : \langle p \rangle\} \to X$.

For space reasons, we specify only extension types:

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

These clauses precisely axiomatize the rules of extension types given in Section IV-A. Once the signature is complete, we obtain from Uemura's framework a category with representable maps $\mathbb{T}$ together with a bi-initial model $\mathcal{I}$.

C. The atomic figure shape and its universal property

For each context Γ and type $\Gamma \vdash A$ *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 $\mathcal{I}_\diamond$, 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 $\Gamma, p \vdash q$ *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 $\mathcal{I}_\diamond$ and whose morphisms are atomic substitutions, but this construction obscures a beautiful and simple (2,1)-categorical universal property first exposed by Bocquet, Kaposi, and Sattler [32] 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 $\mathcal{I}$ of $\mathbb{T}$ is, by restriction along $\mathbb{T}_0 \hookrightarrow \mathbb{T}$, also a model of $\mathbb{T}_0$.

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

Atomic substitution algebras arrange themselves into a (2,1)-category, a full subcategory of $\mathbf{Mod}\,\mathbb{T}_0 \downarrow \mathcal{I}$.

Theorem 16 (Bocquet, Kaposi, and Sattler [32]): The bi-initial atomic substitution algebra $(\mathcal{A}, \alpha : \mathcal{A} \longrightarrow \mathcal{I})$ exists.

We use the bi-initial atomic substitution algebra as a *figure shape* in the sense of Sterling [11, §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" [22]; object-space refers to the object language embodied in the model $\mathcal{I}$, whereas meta-space refers to the metalanguage embodied in the model $\mathcal{A}$. Later on, we will construct a *glued* topos in which we may speak of constructs that have extent in both object-space and meta-space. We follow Anel and Joyal [33] and Vickers [34] in emphasizing the distinction between a topos $\mathbf{X}$ and the category of sheaves $\mathcal{S}_{\mathbf{X}}$ 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 $\mathcal{S}_{\mathbf{I}} = \mathbf{Pr}(\mathcal{I}_\diamond)$ and $\mathcal{S}_{\mathbf{A}} = \mathbf{Pr}(\mathcal{A}_\diamond)$.

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

Definition 19: We denote by $\mathbf{G}$ the *closed mapping cylinder* [35] of the geometric morphism $\boldsymbol{\alpha} : \mathbf{A} \longrightarrow \mathbf{I}$, such that $\mathcal{S}_{\mathbf{G}}$ is the comma category $\mathcal{S}_{\mathbf{A}} \downarrow \boldsymbol{\alpha}^*$. We will write $\boldsymbol{j} : \mathbf{I} \hookrightarrow \mathbf{G}$ and $\boldsymbol{i} : \mathbf{A} \hookrightarrow \mathbf{G}$ for the open and closed subtopos inclusions.

D. The language of synthetic Tait computability

As $\mathbf{I}$ and $\mathbf{A}$ are both subtopoi of $\mathbf{G}$, they are reflected in the internal language of $\mathcal{S}_{\mathbf{G}}$ by means of a pair of complementary lex idempotent monads ($\bigcirc$,$\bullet$). The internal language of $\mathcal{S}_{\mathbf{I}}$ is presented by the $\bigcirc$-modal or *object-space* types and $\mathcal{S}_{\mathbf{A}}$ 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 20: There exists a proposition $\mathsf{obj} : \Omega$ such that

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

Remark 21: We will use extension types $\{A \mid \phi \hookrightarrow a\}$ in the internal language of $\mathcal{S}_{\mathbf{G}}$ 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 22: We isolate a subuniverse $\Omega_{dec} \subseteq \Omega$ of the subobject classifier that is closed under finite disjunctions and contains obj; Ω_{dec} will ultimately be a subuniverse spanned by pointwise/externally decidable propositions [9], but this fact will not play a role in the synthetic development.

Notation 23: We will reuse Notation 14 and write $\{\mathsf{obj}\}\,A$ rather than $\{\_ : \mathsf{obj}\} \to A$ when $A : \{\_ : \mathsf{obj}\} \to \mathcal{U}$.

As a presheaf topos, $\mathcal{S}_{\mathbf{G}}$ inherits a hierarchy of cumulative universes $\mathcal{U}_i$, each of which supports the *strict gluing* or *(mixed-phase) refinement* type [22], [36]: 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:

$$\frac{\begin{array}{c} A : \{\mathsf{obj}\}\,\mathcal{U}_i \\ B : (\{\mathsf{obj}\}\,A) \to \mathcal{U}_i \qquad \{\mathsf{obj}\}(a : A) \to (B\,a \cong \mathbf{1}) \end{array}}{(x : A) \ltimes B\,x : \{\mathcal{U}_i \mid \mathsf{obj} \hookrightarrow A\}}$$

$$\mathsf{gl} : \left\{ \left(\textstyle\sum_{x:\{\mathsf{obj}\}\,A} B\,x\right) \cong (x : A) \ltimes B\,x \;\middle|\; \mathsf{obj} \hookrightarrow \pi_1 \right\}$$

Remark 24: 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 25: We write $[\mathsf{obj} \hookrightarrow a \mid b]$ for $\mathsf{gl}\,(a, b)$ and $\mathsf{ungl}\,x$ for $\pi_2\big(\mathsf{gl}^{-1} x\big)$.

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 [11], 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 26: 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 $\boldsymbol{\mathcal{S}}_\mathsf{I}$ internalizes into $\boldsymbol{\mathcal{S}}_\mathsf{G}$ as a sequence of constants valued in the subuniverse $\mathcal{U}^0_\bigcirc$; for instance, we have:

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

Following Remark 24, we package the pair $(\mathsf{in}_p, \mathsf{out}_p)$ as a single isomorphism in_p.

The presheaf of terms in the model $\mathcal{A}$ internalizes as a meta-space type of *variables* which by virtue of the structure map $\mathcal{A} \longrightarrow \mathcal{I}$ 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 27: 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 [11], [12]. 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.

$$\begin{array}{l}
\mathbf{var} : (x : \mathsf{var}\,A) \to \mathsf{ne}_\bullet\,A\,\bot\,x \\
\mathbf{unst} : \{a : \mathsf{tm}\,A\} \to \mathsf{ne}_\bullet\,A\,\top\,a \\
\_ : (a : \mathsf{tm}\,A)\,(e : \mathsf{ne}_\bullet\,A\,\top\,a) \to e = \mathbf{unst}\,a \\[1em]
\mathbf{ext}_p : \mathsf{nftp}_\bullet\,A \to (\{p\}\,\mathsf{nf}_\bullet\,A\,a) \to \mathsf{nftp}_\bullet\,(\mathsf{ext}_p\,A\,a) \\
\mathbf{in}_p : \mathsf{nftp}_\bullet\,A \to \mathsf{nf}_\bullet\,A\,u \to \mathsf{nf}_\bullet\,(\mathsf{ext}_p\,A\,a)\,u \\
\mathbf{out}_p : \mathsf{nftp}_\bullet\,A \to \mathsf{ne}_\bullet\,(\mathsf{ext}_p\,A\,a)\,\phi\,u \\
\quad \to \mathsf{ne}_\bullet\,A\,(\phi \vee \langle p \rangle)\,\big(\mathsf{in}_p^{-1}\,u\big) \\[1em]
\mathbf{up}_{\mathsf{uni}} : \mathsf{ne}_\bullet\,\mathsf{uni}\,\phi\,A \to (\{\phi\}\,\mathsf{nftp}_\bullet\,A) \to \mathsf{nftp}_\bullet\,(\mathsf{el}\,A) \\
\mathbf{up}_{\mathsf{el}} : \mathsf{ne}_\bullet\,\mathsf{uni}\,\phi\,A \to (\{\phi\}\,\mathsf{nftp}\,(\mathsf{el}\,A)) \to \mathsf{ne}_\bullet\,(\mathsf{el}\,A)\,\psi\,a \\
\quad \to (\{\phi \vee \psi\}\,\mathsf{nf}_\bullet\,(\mathsf{el}\,A)\,a) \to \mathsf{nf}_\bullet\,(\mathsf{el}\,A)\,a \\
\mathbf{el} : \mathsf{ne}_\bullet\,\mathsf{uni}\,\phi\,A \to (\{\phi\}\,\mathsf{nf}_\bullet\,\mathsf{uni}\,A) \to \mathsf{nf}_\bullet\,\mathsf{uni}\,A \\[1em]
\_ : (e : \mathsf{ne}_\bullet\,\mathsf{uni}\,\top\,A)\,(u : \mathsf{nftp}_\bullet\,A) \to \mathbf{up}_{\mathsf{uni}}\,e\,u = u \\
\_ : (e : \mathsf{ne}_\bullet\,\mathsf{uni}\,\top\,A)\,(u : \mathsf{nf}_\bullet\,\mathsf{uni}\,A) \to \mathbf{el}\,e\,u = u \\
\_ : \{\phi \vee \psi\}\,(e_A : \mathsf{ne}_\bullet\,\mathsf{uni}\,\phi\,A)\,(u_A : \{\phi\}\,\mathsf{nftp}_\bullet\,(\mathsf{el}\,A)) \\
\quad (e_a : \mathsf{ne}_\bullet\,(\mathsf{el}\,A)\,\psi\,a)\,(u_a : \mathsf{nf}_\bullet\,A\,a) \\
\quad \to \mathbf{up}_{\mathsf{el}}\,e_A\,u_A\,e_a\,u_a = u_a
\end{array}$$

Fig. 1. Selected rules from the definition of nf, ne, and nftp.

E. 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 [12] 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 [37] specifying the meta-space components of normal and neutral forms:

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

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:

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

We illustrate a representative fragment of the inductive definitions in Fig. 1.

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 to prove Theorem 32 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.

F. The normalization model

The construction of the normalization model itself is now reduced to a series of programming exercises. We must give a new $\mathbf{TT}_\mathbb{P}$-algebra internally to $\boldsymbol{\mathcal{S}}_\mathsf{G}$ subject to the constraint that each of its constituents restricts under obj to the corresponding constant from the $\mathbf{TT}_{\mathbb{P}}$-algebra inherited from $\boldsymbol{\mathcal{S}}_{\mathsf{l}}$. 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:

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

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 [12]. We finally define both tp^* and tm^* using strict gluing to achieve the correct boundary:

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

Notation 28: 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.

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

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

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:

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

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

$(\mathsf{ext}^*_p\ A\ a).\mathsf{code} = \mathbf{ext}_p\ A.\mathsf{code}\ (A.\mathsf{reify}\ a)$
$(\mathsf{ext}^*_p\ A\ a).\mathsf{tm}_\bullet\ x = \bullet\{A.\mathsf{tm}_\bullet\ (\mathsf{in}^{-1}_p x) \mid \langle p \rangle \hookrightarrow a\}$
$(\mathsf{ext}^*_p\ A\ a).\mathsf{reify}\ (\eta_\bullet x) = \mathbf{in}_p\ A.\mathsf{code}\ (A.\mathsf{reify}\ x)$
$(\mathsf{ext}^*_p\ A\ a).\mathsf{reflect}\ x\ \phi\ e\ (\eta_\bullet x_\phi) =$

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

In the clauses of reify and reflect, we were allowed to pattern match on $\eta_\bullet x$ because we are mapping into meta-space types.

Remark 29: 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:

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

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.

The universe: We now turn to the construction of the universe in the normalization model; 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 Fig. 2. The inclusion of $\mathsf{el\text{-}code}$ 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 $\boldsymbol{\mathcal{S}}_{\mathsf{G}}$; 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 $\mathcal{U}_1$.

G. The normalization algorithm

Having constructed the normalization model in $\boldsymbol{\mathcal{S}}_{\mathsf{G}}$, we now extract the actual normalization algorithm using an argument based on those presented by Fiore [38] and Sterling [11]. Despite the differences in the normalization models, this portion of the proof closely follows Sterling's argument.

Theorem 30: The map $\eta_\circ : \mathsf{nftp} \longrightarrow \mathsf{tp}$ is an isomorphism.

In fact, as our construction of the normalization model is constructive, its inverse is computable.

Definition 31: An object $X \in \boldsymbol{\mathcal{S}}_{\mathsf{G}}$ is called *levelwise decidable* when for each $\Gamma \in \mathcal{A}_\diamond$, the set $(\boldsymbol{i}^* X)\Gamma$ is decidable where $\boldsymbol{i} : \mathbf{A} \hookrightarrow \mathbf{G}$ is as in Definition 19.

Theorem 32: Viewed as objects of $\boldsymbol{\mathcal{S}}_{\mathsf{G}}$, the following are levelwise decidable:

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

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

Corollary 33: Definitional equality in $\mathbf{TT}_{\mathbb{P}}$ is decidable, and type constructors in $\mathbf{TT}_{\mathbb{P}}$ are definitionally injective.

VII. 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.*

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 `lock`ing idioms in pure Gallina code.

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

Fig. 2. The normalization structure on the universe.

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.

Program verifiers such as VeriFast and Chalice include similar unfolding mechanisms to cope specifically with recursive definitions [39], [40]. 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.

Translucent ascription in module systems

Thus far we have focused on proof assistants, but similar considerations arise for ML-style module systems [10], [41]–[43]. 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 [42] relies on *singleton kinds* [44], [45], 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 [10]. Thus the translucency of compiletime module components can be seen as a *particular* controlled unfolding policy in the sense of this paper.

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 within Agda [46]. 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.

VIII. 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 novel 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.

References

[1] The Agda Development Team, "The Agda programming language," 2022. [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php

[2] The Coq Development Team, "The Coq proof assistant," 2022. [Online]. Available: https://www.coq.inria.fr

[3] G. Gonthier, A. Mahboubi, and E. Tassi, "A small scale reflection extension for the Coq system," Inria Saclay Ile de France, Research Report RR-6455, 2016. [Online]. Available: https://hal.inria.fr/inria-00258384

[4] The Agda Development Team, "The Agda standard library," 2022. [Online]. Available: https://github.com/agda/agda-stdlib

[5] E. Riehl and M. Shulman, "A type theory for synthetic ∞-categories," *Higher Structures*, vol. 1, no. 1, pp. 147–224, 2017. [Online]. Available: https://arxiv.org/abs/1705.07442

[6] T. RedPRL Development Team, "`cooltt`," 2020. [Online]. Available: http://www.github.com/RedPRL/cooltt

[7] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical Type Theory: a constructive interpretation of the univalence axiom," vol. 4, no. 10, pp. 3127–3169, 2017.

[8] C. Angiuli, K.-B. Hou (Favonia), and R. Harper, "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities," in *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), D. Ghica and A. Jung, Eds., vol. 119. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 6:1–6:17. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9673

[9] C. Angiuli, G. Brunerie, T. Coquand, K.-B. Hou (Favonia), R. Harper, and D. R. Licata, "Syntax and models of Cartesian cubical type theory," vol. 31, no. 4, pp. 424–468, 2021.

[10] J. Sterling and R. Harper, "Logical relations as types: Proof-relevant parametricity for program modules," *Journal of the ACM*, vol. 68, no. 6, Oct. 2021.

[11] J. Sterling, "First steps in synthetic Tait computability: The objective metatheory of cubical type theory," Ph.D. dissertation, Carnegie Mellon University, 2021, version 1.1, revised May 2022.

[12] J. Sterling and C. Angiuli, "Normalization for cubical type theory," in *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '21. New York, NY, USA: ACM, 2021.

[13] D. Gratzer, "Normalization for multimodal type theory," 2021.

[14] P. Martin-Löf, "An intuitionistic theory of types: predicative part," in *Logic Colloquium '73, Proceedings of the Logic Colloquium*, ser. Studies in Logic and the Foundations of Mathematics, H. Rose and J. Shepherdson, Eds. North-Holland, 1975, vol. 80, pp. 73–118.

[15] T. Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, Institute for Advanced Study, 2013. [Online]. Available: https://homotopytypetheory.org/book

[16] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau, "Definitional proof-irrelevance without K," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290316

[17] P.-E. Dagand, "A cosmology of datatypes: Reusability and dependent types," Ph.D. dissertation, University of Strathclyde, Glasgow, Scotland, 08 2013.

[18] D. Gratzer, J. Sterling, and L. Birkedal, "Implementing a Modal Dependent Type Theory," *Proc. ACM Program. Lang.*, vol. 3, 2019.

[19] T. Coquand, "An algorithm for type-checking dependent types," *Science of Computer Programming*, vol. 26, no. 1, pp. 167–177, 1996.

[20] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Transactions Programming Language and Systems*, vol. 22, no. 1, pp. 1–44, 2000.

[21] K.-B. Hou (Favonia), "`kado`," 2022, open-source OCaml library for deciding cofibrations in Cartesian cubical type theory. [Online]. Available: http://www.github.com/RedPRL/kado

[22] J. Sterling, "Naïve logical relations in synthetic Tait computability," Jun. 2022, unpublished manuscript.

[23] J. Cartmell, "Generalised algebraic theories and contextual categories," Ph.D. dissertation, University of Oxford, 1978.

[24] P. Freyd, "Aspects of topoi," *Bulletin of the Australian Mathematical Society*, vol. 7, no. 1, p. 1–76, 1972.

[25] D. Gratzer and J. Sterling, "Syntactic categories for dependent type theory: sketching and adequacy," 2020.

[26] T. Uemura, "A general framework for the semantics of type theory," 04 2019. [Online]. Available: https://arxiv.org/abs/1904.04097

[27] ——, "Abstract and concrete type theories," Ph.D. dissertation, Institute for Logic, Language and Computation, University of Amsterdam, 2021.

[28] P. Dybjer, "Internal type theory," in *Types for Proofs and Programs*, S. Berardi and M. Coppo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134.

[29] S. Awodey, "Natural models of homotopy type theory," *Mathematical Structures in Computer Science*, vol. 28, no. 2, pp. 241–286, 2018.

[30] P. Clairambault and P. Dybjer, "The biequivalence of locally cartesian closed categories and martin-löf type theories," *Mathematical Structures in Computer Science*, vol. 24, no. 6, 2014.

[31] C. Newstead, "Algebraic models of dependent type theory," PhD thesis, Carnegie Mellon University, 2018. [Online]. Available: https://sites.math.northwestern.edu/~newstead/thesis-clive-newstead.pdf

[32] R. Bocquet, A. Kaposi, and C. Sattler, "Relative induction principles for type theories," 2021.

[33] M. Anel and A. Joyal, "Topo-logie," in *New Spaces in Mathematics: Formal and Conceptual Reflections*, M. Anel and G. Catren, Eds., vol. 1, ch. 4, pp. 155–257.

[34] S. Vickers, "Locales and toposes as spaces," in *Handbook of Spatial Logics*, M. Aiello, I. Pratt-Hartmann, and J. Van Benthem, Eds. Dordrecht: Springer Netherlands, 2007, pp. 429–496.

[35] P. T. Johnstone, *Topos Theory*. Academic Press, 1977.

[36] D. Gratzer, M. Shulman, and J. Sterling, "Strict universes for Grothendieck topoi," 2022, unpublished manuscript. [Online]. Available: https://arxiv.org/abs/2202.12012

[37] A. Kaposi, A. Kovács, and T. Altenkirch, "Constructing quotient inductive-inductive types," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 2:1–2:24, Jan. 2019.

[38] M. Fiore, "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*, ser. PPDP '02. ACM, 2002, pp. 26–37.

[39] B. Jacobs, F. Vogels, and F. Piessens, "Featherweight VeriFast," *Logical Methods in Computer Science*, vol. Volume 11, Issue 3, Sep. 2015.

[40] A. J. Summers and S. Drossopoulou, "A formal semantics for isorecursive and equirecursive state abstractions," in *ECOOP 2013 – Object-Oriented Programming*, G. Castagna, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 129–153.

[41] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[42] R. Harper and C. Stone, "A type-theoretic interpretation of Standard ML," in *Proof, Language, and Interaction*, G. Plotkin, C. Stirling, and M. Tofte, Eds. Cambridge, MA, USA: MIT Press, 2000, pp. 341–387.

[43] D. Dreyer, K. Crary, and R. Harper, "A type system for higher-order modules," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03, New Orleans, Louisiana, USA, 2003, pp. 236–249.

[44] D. Aspinall, "Subtyping with singleton types," in *Computer Science Logic*, L. Pacholski and J. Tiuryn, Eds. Springer Berlin Heidelberg, 1995, pp. 1–15.

[45] C. A. Stone and R. Harper, "Extensional equivalence and singleton types," vol. 7, no. 4, pp. 676–722, 2006.

[46] A. Liao and J. Cockx, "Unfolding control for abstract blocks," 2022. [Online]. Available: https://github.com/agda/agda/pull/6354

Appendix

A. *The LF signature for* $\mathbf{TT}_{\mathbb{P}}$

We present the nonstandard portion of the signature of $\mathbf{TT}_{\mathbb{P}}$ in Fig. 3.

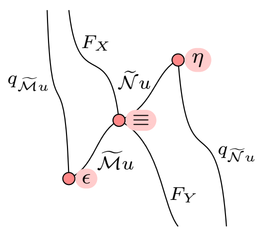
B. *The (2,1)-category of models of a type theory*

Uemura [27] has observed that a model $(\mathcal{M}_\diamond, \mathcal{M})$ in the sense of Definition 11 can be packaged into a single functor $\widetilde{\mathcal{M}} : \mathbb{T}^{\triangleright} \longrightarrow \mathbf{Cat}$, 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 $\widetilde{\mathcal{M}}X = \int_{\mathcal{M}_\diamond} \mathcal{M}X$ of a discrete fibration over $\widetilde{\mathcal{M}}\diamond = \mathcal{M}_\diamond$. Here we are using the equivalence between $\mathbf{DFib}_{\mathcal{C}} \simeq \mathbf{Pr}(\mathcal{C})$. The preservation of representable maps is then rendered here as the requirement that for representable $u : X \longrightarrow Y$, each functor $\widetilde{\mathcal{M}}u : \widetilde{\mathcal{M}}X \longrightarrow \widetilde{\mathcal{M}}Y$ shall have a right adjoint $\widetilde{\mathcal{M}}u \dashv q_{\widetilde{\mathcal{M}}u}$ taking an element of $\widetilde{\mathcal{M}}Y$ to the *generic* element of $\widetilde{\mathcal{M}}X$ in the extended context.

Example 34: For a representable map $\pi : \mathsf{tm} \longrightarrow \mathsf{tp}$, the functorial action $\widetilde{\mathcal{M}}\pi : \widetilde{\mathcal{M}}\,\mathsf{tm} \longrightarrow \widetilde{\mathcal{M}}\,\mathsf{tp}$ takes a term $\Gamma \vdash a : A$ to the type $\Gamma \vdash A$; the right adjoint $q_{\widetilde{\mathcal{M}}\pi} : \widetilde{\mathcal{M}}\,\mathsf{tp} \longrightarrow \widetilde{\mathcal{M}}\,\mathsf{tm}$ sends a type $\Gamma \vdash A$ to the variable $\Gamma, a : A \vdash a : A$.

Definition 35: Given two models $\mathcal{M}, \mathcal{N}$ of $\mathbb{T}$, a *morphism of models* from $\mathcal{M}$ to $\mathcal{N}$ is given by a natural transformation $F : \widetilde{\mathcal{M}} \longrightarrow \widetilde{\mathcal{N}} \in [\mathbb{T}^{\triangleright}, \mathbf{Cat}]$ such that the left-hand square below satisfies the Beck–Chevalley condition in the sense that for each representable $u : X \longrightarrow Y : \mathbb{T}$ the right-hand wiring diagram (oriented from top left to bottom right) denotes an invertible natural transformation $F_X \circ q_{\widetilde{\mathcal{M}}u} \longrightarrow q_{\widetilde{\mathcal{N}}u} \circ F_Y$:

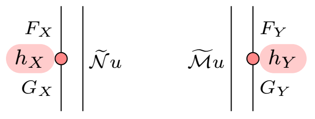
$$\begin{tikzcd}
\widetilde{\mathcal{M}}X \arrow[r, "F_X"] \arrow[d, "\widetilde{\mathcal{M}}u"'] & \widetilde{\mathcal{N}}X \arrow[d, "\widetilde{\mathcal{N}}u"] \\
\widetilde{\mathcal{M}}Y \arrow[r, "F_Y"'] & \widetilde{\mathcal{N}}Y
\end{tikzcd}$$



F_X
η
$q_{\widetilde{\mathcal{M}}u}$
$\widetilde{\mathcal{N}}u$
$\equiv$
$q_{\widetilde{\mathcal{N}}u}$
$\widetilde{\mathcal{M}}u$
ϵ
F_Y

Given a natural transformation $F : \widetilde{\mathcal{M}} \longrightarrow \widetilde{\mathcal{N}} \in [\mathbb{T}^{\triangleright}, \mathbf{Cat}]$, we have a left Kan extension $F_{\diamond !} : \mathbf{DFib}_{\mathcal{M}_\diamond} \longrightarrow \mathbf{DFib}_{\mathcal{N}_\diamond}$. This functor can be used to re-express the Beck–Chevalley condition in a manner that is more amenable to computations.

Definition 36: Let $\mathcal{M}, \mathcal{N}$ be two models of $\mathbb{T}$, and let $F, G : \mathcal{M} \longrightarrow \mathcal{N}$ 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 $[\widetilde{\mathcal{M}}X, \widetilde{\mathcal{N}}X]$, subject to the coherence condition that for each $u : X \longrightarrow Y$ in $\mathbb{T}^{\triangleright}$ the following two wiring diagrams are equal:



Naturality of F, G ensures that the diagrams above have the same boundary.

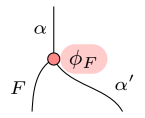
Remark 37: Because each of the induced maps $\pi_{\mathcal{M}X} : \widetilde{\mathcal{M}}X \longrightarrow \mathcal{M}_\diamond$ and $\pi_{\mathcal{N}X} : \widetilde{\mathcal{N}}X \longrightarrow \mathcal{N}_\diamond$ into the cone point are discrete fibrations, it suffices to check the modification 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 $x \in \widetilde{\mathcal{M}}X$ we are only requiring that $(h_\diamond)^*_{\pi_{\mathcal{M}x}}(G_X x) = F_X x$ in the sense depicted below in the discrete fibration $\mathcal{N}X$ over $\mathcal{N}_\diamond$:

$$\begin{tikzcd}
F_X x \arrow[r, dashed, "\exists! h_X x"] \arrow[d, maps to] \arrow[dr, phantom, very near start, "\lrcorner"] & G_X x \arrow[d, maps to] \\
F_\diamond(\pi_{\mathcal{M}X} x) \arrow[r, "(h_\diamond)_{\pi_{\mathcal{M}x}}"'] & G_\diamond(\pi_{\mathcal{M}X} x)
\end{tikzcd}$$

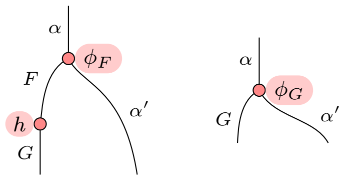
Thus we have a (2,1)-category of models $\mathbf{Mod}\,\mathbb{T}$ for any category with representable maps $\mathbb{T}$.

C. *The (2,1)-category of atomic substitution algebras*

Definition 38: Given atomic substitution algebras $\alpha : \mathcal{A} \longrightarrow \mathcal{I}$ and $\alpha' : \mathcal{A}' \longrightarrow \mathcal{I}$, a morphism from $(\mathcal{A}, \alpha)$ to $(\mathcal{A}', \alpha')$ is given by a morphism $F : \mathcal{A} \longrightarrow \mathcal{A}' \in \mathbf{Mod}\,\mathbb{T}_0$ together with an isomorphism $\phi_F : \alpha \longrightarrow \alpha' \circ F$ in $[\mathcal{A}, \mathcal{I}]$ as depicted below:



Definition 39: Given two morphisms $F, G : (\mathcal{A}, \alpha) \longrightarrow (\mathcal{A}', \alpha')$, an isomorphism from F to G is given by an isomorphism $h : F \longrightarrow G \in [\mathcal{A}, \mathcal{A}']$ such that the following wiring diagrams denote equal isomorphisms $\alpha \longrightarrow \alpha' \circ G$:



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

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

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

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

Fig. 3. The non-standard aspects of the LF signature for $\mathbf{TT}_{\mathbb{P}}$.