CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Formalization of Opaque Definitions for a Dependent Type Theory

Master's thesis in Computer science and engineering

Eve Geng

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Master's thesis 2025

Formalization of Opaque Definitions for a Dependent Type Theory

Eve Geng

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Formalization of Opaque Definitions for a Dependent Type Theory
Eve Geng

Supervisor: Nils Anders Danielsson, Computer Science and Engineering
Examiner: Patrik Jansson, Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Formalization of Opaque Definitions for a Dependent Type Theory
Eve Geng
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Definitions allow for the efficient reuse of code, but the process of unfolding complex nested definitions may cause usability issues for the programmer. Opaque definitions help mitigate this by restricting the unfolding of definitions at type-checking time, but their implementation demands some subtlety—for instance, subject reduction may be lost if an opaque type definition doesn't unfold in exactly the right way. Despite this, there has been relatively little work on formally characterizing opaque definitions.

We contribute here a formalization of opaque top-level definitions based on their implementation in the Agda programming language. The formalization is fully mechanized in Agda as an extension of the prior graded-type-theory project. We give typing and reduction rules for the definitions, then show, through a Kripke logical relation argument, that they enjoy many of the usual desirable type-theoretic properties: subject reduction, normalization, consistency, decidability of conversion, and so on.

Keywords: dependent types, top-level definitions, opaque definitions, formalization, logical relations, Agda.

Acknowledgements

I extend my thanks to my advisor Nils Anders, whose advice and guidance, both in the realm of theory as well as in the practical work of a master's project, have been indispensable to the completion of this thesis.

Eve Geng, Gothenburg, 2025-06-21

Contents

1 Introduction

Most programming languages in everyday use today offer some facility for *definitions*: a programmer can choose a name α, then declare that α refers to some fixed definiens t, which we might denote by $\alpha \triangleq t$. This makes it very easy to reuse code without having to repeat oneself: one can simply write the name α in place of the code at any desired usage site. For example, the following Agda definitions bind the name `double` to a doubling function, then reuse it to define `quadruple`:

```
double : ℕ → ℕ – Doubles a natural by adding it to itself
double n = n + n

quadruple : ℕ → ℕ – Quadruples a natural by doubling it twice
quadruple n = double (double n)
```

This same sort of construction also works at the type level, giving us *type aliasing*. For example, we can name the type of positive natural numbers `ℕ>0`, allowing us to reuse it as such in a later type signature:

```
ℕ>0 : Set – Positive naturals: a natural paired with a proof that it's >0
ℕ>0 = Σ[ n ∶∶ ℕ ] n > 0

pred : ℕ>0 → ℕ – Predecessor as a total function on the positive naturals
pred (suc n , >zero) = n
```

A key feature of definitions is that by composing them, we can incrementally build up complexity with comparatively little syntactical cost. Consider, for example, the following definitions for a number hierarchy with setoid equality relations[1]:

```
ℤ : Set – Integers: the pair (m , n) represents the integer m - n
ℤ = ℕ × ℕ

_≡ℤ_ : ℤ → ℤ → Set – Integer equality: (a - c) = (b - d) exactly when a + d = b + c
x ≡ℤ y = fst x + snd y ≡ fst y + snd x

_*ℤ_ : ℤ → ℤ → ℤ – Integer product: (a - c) * (b - d) is just (ab + cd) - (ad + bc)
x *ℤ y = fst x * fst y + snd x * snd y , fst x * snd y + fst y * snd x

0ℤ : ℤ – Integer zero: 0 - 0
0ℤ = 0 , 0

ℤ≠0 : Set – Nonzero integers: an integer paired with a proof that it's not zero
ℤ≠0 = Σ[ x ∶∶ ℤ ] ¬ x ≡ℤ 0ℤ

ℚ : Set – Rationals: the pair (m , n) represents the fraction m / n
ℚ = ℤ × ℤ≠0

_≡ℚ_ : ℚ → ℚ → Set – Rational equality: a/c = b/d exactly when ad = bc
x ≡ℚ y = fst x *ℤ fst (snd y) ≡ℤ fst y *ℤ fst (snd x)
```

[1] This particular example might be more clearly expressed with pattern matching, but we avoid it here for illustrative purposes.

Although equality on ℚ is, on its face, a fairly involved notion, our ability to build on earlier definitions allows us to state it here quite concisely. This kind of scalability makes definitions indispensable to any large software project.

The semantics of definitions—at least, informally—are fairly obvious: a definition name is equal to its corresponding definiens. This is straightforward for execution, but it may pose some problems at compile time, especially when definitions become nested in complex ways. For example, suppose that we want to show that our equality relation on the rationals is symmetric:

```
sym-ℚ : ∀ {x y} → x ≡ℚ y → y ≡ℚ x – Symmetry of rational equality
sym-ℚ p = {! insert proof here !}
```

When approaching a proof obligation like this one, a typical step is to ask Agda for the normalized goal type, which reveals exactly what shape of term is needed to resolve it. For this particular goal, we get the following normalized type:

```
fst (fst y) * fst (fst (snd x)) +
snd (fst y) * snd (fst (snd x)) +
(fst (fst x) * snd (fst (snd y)) +
fst (fst (snd y)) * snd (fst x)) ≡
fst (fst x) * fst (fst (snd y)) +
snd (fst x) * snd (fst (snd y)) +
(fst (fst y) * snd (fst (snd x)) +
fst (fst (snd x)) * snd (fst y))
```

What's happening here? When the type checker normalizes a type, it *unfolds* all the definitions it sees, replacing each definition name with its corresponding definiens. In this case, the definition of rationals ℚ is unfolded into pairs of integers ℤ, which is then unfolded into pairs of naturals ℕ, carrying along all the `fst` and `snd` projections on the way down. At the same time, `_≡ℚ_` is unfolded to reveal the underlying `_≡ℤ_`, which is then unfolded into `_≡_` over ℕ. This unfolding process is necessary for terms containing definitions to type-check, but it introduces two pain points, as illustrated by the above:

1. Excessive unfolding of definitions can impair usability by obfuscating types shown to users, such as in the goal type above.
2. Unnecessary unfolding of complex nested definitions can bog down type checking, which usually involves the normalization of types.

What we're missing here is a sort of separation of concerns: when considering the properties of rationals, neither the programmer nor the compiler should be dealing with the "implementation details", so to speak, of integers or naturals under the hood.

Opaque definitions [10] are one way of enforcing this separation. By marking a definition as "opaque", the programmer declares that the type checker should *not* unfold it unless explicitly instructed to. In Agda, this is done using the `opaque` keyword:

```
opaque
  ℤ : Set – Integers: the pair (m , n) represents the integer m - n
  ℤ = ℕ × ℕ
```

With ℤ now opaque, it is no longer definitionally equal to ℕ × ℕ. As a consequence, the definition of _≡ℤ_ no longer type-checks, since the pair projections `fst` and `snd` no longer apply to ℤ. To fix this, we can use the `unfolding` keyword to tell the type checker that the definition _≡ℤ_ should unfold the opaque definition ℤ, thus re-establishing the lost equality:

```
opaque
  unfolding ℤ
  _≡ℤ_ : ℤ → ℤ → Set – (a - c) = (b - d) exactly when a + d = b + c
  x ≡ℤ y = fst x + snd y ≡ fst y + snd x
```

Such definitions make up a sort of "interface" for the opaque ℤ, allowing us to reason about it without having to look into the underlying implementation details. Note that interface definitions must also be opaque in order to avoid leaking details about the definitions they unfold. Later on, when defining the rationals, we can use the interface to avoid unfolding ℤ at all:

```
opaque
  ℤ≠0 : Set – Nonzero integers: an integer paired with a proof that it's not zero
  ℤ≠0 = Σ[ x ∶∶ ℤ ] ¬ x ≡ℤ 0ℤ

opaque
  ℚ : Set – Rationals: the pair (m , n) represents the fraction m / n
  ℚ = ℤ × ℤ≠0

opaque
  unfolding ℚ ℤ≠0
  _≡ℚ_ : ℚ → ℚ → Set – Rational equality: a/c = b/d exactly when ad = bc
  x ≡ℚ y = fst x *ℤ fst (snd y) ≡ℤ fst y *ℤ fst (snd x)
```

The symmetry proof is now part of the interface for ℚ, and so it unfolds _ℚ_:

```
opaque
  unfolding _≡ℚ_
  sym-ℚ : ∀ {x y} → x ≡ℚ y → y ≡ℚ x – Symmetry of rational equality
  sym-ℚ p = {! insert proof here !}
```

As a result of opacity, the normalized goal type is now:

```
fst y *ℤ fst (snd x) ≡ℤ fst x *ℤ fst (snd y)
```

Here, the type checker has unfolded _≡ℚ_ (and, implicitly, ℚ and ℤ≠0) as asked, but has gotten stuck on the opaque _*ℤ_ and _≡ℤ_, thus yielding a rather more readable goal type than before. In fact, this goal reduces to symmetry of _≡ℤ_, which would be part of the interface for ℤ.

The implicit unfolding of transitive dependencies as in the above is used by Agda to ensure subject reduction. Suppose, for example, that we wrote a new definition unfolding 0ℤ without transitively unfolding ℤ. In that definition's body, 0ℤ of type ℤ would reduce to (0 , 0) of type ℕ × ℕ, but we would *not* be able to deduce that ℤ is equal to ℕ × ℕ! This illustrates that the implementation of opaque definitions in a metatheoretically-sound way is not totally trivial, which makes it a good target for formalization.

Our main contribution is a fully-mechanized formalization of a dependent type theory with opaque top-level definitions. We build off of the previous work of the graded-type-theory project [3], an Agda formalization of a (graded modal) dependent type theory which, prior to this work, lacked definitions of any kind. The core calculus is loosely based on the Agda language and comes equipped with various common type formers: dependent functions and pairs, a universe hierarchy, natural numbers, identity types, and so on. A Kripke logical relation [1][2] is then employed to derive a number of desirable metatheoretic properties: subject reduction, normalization, consistency, decidability of conversion, and so on.

In this work, we extend the type theory with opaque definitions—adding top-level definitions in general along the way—then show that the metatheoretic results established by the formalization are preserved by our extension. This is, to the best of our knowledge, the first mechanization of the metatheory of opaque definitions. We do, however, note a few limitations to our work:

- For simplicity, we only allow top-level definitions; more on this in §2.3.
- The graded portion of the graded-type-theory project gives a "resourcing system" for analyzing the usage of terms in a program; we do not address it here.
- The type theory optionally supports *equality reflection*, a feature present in a handful of dependently-typed languages. We support equality reflection in the presence of definitions, but not *opaque* definitions, and so we will largely be ignoring it here; more on this in §4.4.

The remainder of this paper is structured as follows:

- In §2, we recall the methodology of the formalization and go over the general idea of our extensions. We also compare our formalism with other existing ones.
- In §3, we detail our extensions for supporting top-level definitions without opacity.
- In §4, we detail our modifications for supporting *opaque* definitions.
- Finally, in §5, we wrap up the work and discuss possibilities for future work.

Finally, a note on mechanization: the type-checked formalization in Agda should be considered the "authoritative" form of this work, this paper being only a presentation of that work. Because of this, some inaccuracies may appear in this paper that are not representative of the formalization. Interested readers can find the original Agda source code on GitHub[2]. Additionally, in the digital version of this paper, definitions and results will be accompanied by links, displayed in blue, to the relevant parts of the Agda code. Because we build up our extensions incrementally throughout the paper, such links will point to a snapshot of the codebase at the appropriate point in development. Accordingly, we distinguish these references with the following coloured markings:

The original formalization, before any of our extensions.

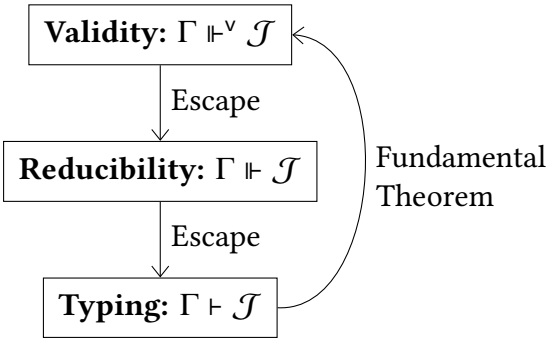
The formalization extended with top-level definitions (without opacity).

The formalization extended with opaque definitions.

[2] https://github.com/phantamanta44/graded-type-theory

2 Background

Here, we give an overview of the structure of the graded-type-theory formalization in order to set the context for our extensions and to fix notation. The idea is broadly to define three different versions of the typing judgements with varying levels of "strength", then to establish a logical equivalence that lets us freely move up and down the ladder:



We also briefly describe the general idea of our extensions and discuss where they lie relative to the state of the art.

2.1 The Type Theory

As previously mentioned, the graded-type-theory formalization supports a variety of type formers; in fact, for generality, it's *configurable* in the sense that it is parametric in a set of flags that control the inclusion or exclusion of certain type formers. However, it turns out that the choice of type formers is largely orthogonal to the problem of opaque definitions, and so we shall not discuss them in too much detail. A small sampling of the term language is shown below:

Definition 2.1. A **term** is either a variable, represented as a de Bruijn index, or a term constructor applied to subterms:

$$\text{Variable } x ::= x_0 \mid x_1 \mid \dots$$

$$\text{Term } A, B, t, u ::= x \mid \Pi\ A\ B \mid t\ u \mid \Sigma\ A\ B \mid (t, u) \mid \mathbb{N} \mid \mathsf{zero} \mid \mathsf{suc}\ t \mid \dots$$

Here, x_i refers to the variable at index i.

Note that terms are well-scoped in the formalization proper, and so the term type is indexed by $\mathbb{N}$[3]. Typing contexts are as usual:

Definition 2.2. A **typing context** is a list of types for variables in the context:

$$\text{Typing Context } \Gamma, \Delta ::= \varepsilon \mid \Gamma \cdot A$$

Typing contexts are dependent in the sense that later entries may refer to variables bound in earlier ones.

[3]The term and context data types are indexed by the number n of variables present, and the variable term constructor is restricted to indices smaller than n. This ensures that only variables in scope are representable as terms and also simplifies the statement of certain theorems about empty contexts.

We also entertain notions of weakenings and substitutions:

Definition 2.3. A **weakening** is an operation ρ that lifts a term t to a term $t[\rho]$ by incrementing all de Bruijn indices that occur in t by an offset, which has the effect of embedding t into a weaker context with more variables.

Definition 2.4. A **substitution** is an operation σ that transforms a term t into a term $t[\sigma]$ by replacing each variable occuring in t with a specified term.

Weakenings and substitutions are also properly well-scoped, but we again elide this. The basic typing judgements are as usual, as are the reduction relations:

Definition 2.5. The **typing judgements** are:

$\vdash \Gamma$	Γ is a well-formed typing context.
$\Gamma \vdash A$	A is a well-formed type in Γ.
$\Gamma \vdash t : A$	t is a well-formed term of type A in Γ.
$\Gamma \vdash A \equiv B$	A equals B as a type in Γ.
$\Gamma \vdash t \equiv u : A$	t equals u as a term of type A in Γ.
$\Gamma \vdash A \Rightarrow B$	Weak head reduction of A to B as a type in Γ.
$\Gamma \vdash t \Rightarrow u : A$	Weak head reduction of t to u as a term of type A in Γ.
$\rho : \Delta \supseteq \Gamma$	ρ is a well-formed weakening from Γ to Δ.
$\Delta \vdash \sigma : \Gamma$	σ is a well-formed substitution from Γ to Δ.

The typing rules are more or less standard, albeit with some configurability for the inclusion and exclusion of rules that may differ between programming languages: equality reflection, axiom K, and so on. With these rules, we can immediately derive a handful of "direct" results:

Lemma 2.6 (Weakening)**.** Typing judgements are preserved under weakening. That is, if $\Gamma \vdash \mathcal{J}$, then for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, we have $\Delta \vdash \mathcal{J}[\rho]$.

Lemma 2.7 (Substitution)**.** Typing judgements are preserved under substitution. That is, if $\Gamma \vdash \mathcal{J}$, then for any well-formed substitution $\Delta \vdash \sigma : \Gamma$, we have $\Delta \vdash \mathcal{J}[\sigma]$.

Lemma 2.8 (Well-formedness)**.** Any typing judgement requires that its constituents are well-formed. That is:

- If $\Gamma \vdash A$, then $\vdash \Gamma$.
- If $\Gamma \vdash t : A$, then $\Gamma \vdash A$.
- If $\Gamma \vdash A \equiv B$, then $\Gamma \vdash A$ and $\Gamma \vdash B$.
- If $\Gamma \vdash t \equiv u : A$, then $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$.
- Similarly for reduction.

Note that because the reduction relation for terms is typed, subject reduction reduces to well-formedness of reduction, and so we immediately get:

Theorem 2.9 (Subject Reduction)**.** If $\Gamma \vdash t \Rightarrow u : A$ and $\Gamma \vdash t : A$, then $\Gamma \vdash u : A$.

2.2 The Logical Relation

For more substantial results about reduction, we resort to a Kripke logical relation argument in the style of Abel *et al.* [2]. We define a series of relations $\Gamma \Vdash \mathcal{J}$, called "reducibility judgements", which mirror the structure of the usual typing judgements $\Gamma \vdash \mathcal{J}$. The idea is that the reducibility judgements capture the behaviour of terms under reduction to weak head normal form:

Definition 2.10. A term is **neutral** if it has a variable in its head position.

Definition 2.11. A term is in **weak head normal form** (WHNF) if it is either neutral or a constructor application.

Definition 2.12. The **logical relation for reducibility** consists of the relations[a]:

$\Gamma \Vdash A$	A is a reducible type in Γ.
$\Gamma \Vdash t : A$	t is a reducible term of type A in Γ.
$\Gamma \Vdash A \equiv B$	A and B are reducibly equal types in Γ.
$\Gamma \Vdash t \equiv u : A$	t and u are reducibly equal terms of type A in Γ.

[a] In the formalization proper, the three latter cases are parameterized by a proof of type reducibility. We elide this here for brevity; in fact, this presentation corresponds roughly to the "hidden" variants of the relations, which "hide" the proofs by existential quantification. A universe level parameter is similarly elided.

Reducibility generally entails three things:

1. Everything in sight reduces to a WHNF.
2. The corresponding typing judgement holds for the WHNFs.
3. The WHNFs behave correctly with respect to reducibility.

What it means for a WHNF to "behave correctly" depends on the type in question; for the full details, refer to the Agda code, or to Abel *et al.* [1] or Abel *et al.* [2]. As an example, consider term reducibility for function types: given a reducible type $\Gamma \Vdash T$ that reduces to $\Pi\ A\ B$, the reducibility judgement $\Gamma \Vdash t : T$ holds roughly when:

- t reduces to some WHNF f (i.e. either a lambda abstraction or a neutral).
- $\Gamma \vdash f : \Pi\ A\ B$.
- Reducible application: for any well-formed weakening $\rho : \Delta \supseteq \Gamma$ and any $\Delta \Vdash a : A[\rho]$, we have $\Delta \Vdash f[\rho]\ a : B[\rho][a/x_0]$.
- Reducible congruence: for any well-formed weakening $\rho : \Delta \supseteq \Gamma$ and any $\Delta \Vdash a \equiv b : A[\rho]$, we have $\Delta \Vdash f[\rho]\ a \equiv f[\rho]\ b : B[\rho][a/x_0]$.

Note the generalization over weakenings in the above. We'll want to use reducibility weakening lemmas in later proofs, but because reducibility occurs in negative positions in its own definition (e.g. as a premise to the two latter conditions above), it's difficult to prove these lemmas directly by induction. Instead, by building weakening directly into the definition of reducibility, we're essentially shifting the proof burden of weakening elsewhere—in particular, to the fundamental theorem (2.17). For now, we have:

Lemma 2.13 (Weakening)**.** Reducibility judgements are preserved under weakening. That is, if $\Gamma \Vdash \mathcal{J}$, then for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, we have $\Delta \Vdash \mathcal{J}[\rho]$.

Keep this trick in mind; we'll be using it again later.

What we want now is a means of moving between the normal typing judgements and the reducibility judgements of the logical relation. The backwards direction, known as "escape", is fairly straightforward:

Lemma 2.14 (Escape)**.** If $\Gamma \Vdash \mathcal{J}$, then $\Gamma \vdash \mathcal{J}$.

Since reducibility implies reduction, we get these escape lemmas by well-formedness plus some minor fiddling with equality rules. The forwards direction, on the other hand, is rather more involved. The general idea will be to proceed by induction on the typing derivations $\Gamma \vdash \mathcal{J}$. However, to make the induction go through more smoothly, we'll want to strengthen the motive from just reducibility to what's known as *validity*:

Definition 2.15. The **validity judgements** are the relations:

$\Vdash^{\mathsf{v}} \Gamma$	Every variable in Γ has a valid type.
$\Gamma \Vdash^{\mathsf{v}} A$	A is a valid type in Γ.
$\Gamma \Vdash^{\mathsf{v}} t : A$	t is a valid term of type A in Γ.
$\Gamma \Vdash^{\mathsf{v}} A \equiv B$	A and B are validly equal as types in Γ.
$\Gamma \Vdash^{\mathsf{v}} t \equiv u : A$	t and u are validly equal as terms of type A in Γ.
$\Delta \Vdash^{\mathsf{s}} \sigma : \Gamma$	σ is a valid substitution from Γ to Δ.
$\Delta \Vdash^{\mathsf{s}} \sigma_1 \equiv \sigma_2 : \Gamma$	σ_1 and σ_2 are validly equal substitutions from Γ to Δ.

Validity boils down to reducibility, but respecting substitution. Just as we baked weakening into reducibility to trivialize weakening lemmas, baking substitutions into validity will trivialize substitution lemmas. As an example, consider validity for terms: $\Gamma \Vdash^{\mathsf{v}} t : A$ holds roughly when:

- $\Gamma \Vdash^{\mathsf{v}} A$.
- For any validly equal substitutions $\Delta \Vdash^{\mathsf{s}} \sigma_1 \equiv \sigma_2 : \Gamma$, we have $\Delta \Vdash t[\sigma_1] \equiv t[\sigma_2] : A[\sigma_1]$.

In particular, plain reducibility falls out of this definition as the special case for the trivial substitution $\sigma_1 = \sigma_2 = \mathsf{id}$. By combining this with the escape lemmas for reducibility, we can derive analogous escape lemmas for validity:

Lemma 2.16 (Escape)**.** If $\Gamma \Vdash^{\mathsf{v}} \mathcal{J}$, then $\Gamma \vdash \mathcal{J}$. Similarly, if $\Vdash^{\mathsf{v}} \Gamma$, then $\vdash \Gamma$.

Now, we finally have what we need to state the *fundamental theorem of logical relations*:

Theorem 2.17 (Fundamental Theorem)**.** $\Gamma \vdash \mathcal{J}$ if and only if $\Gamma \Vdash^{\mathsf{v}} \mathcal{J}$. Similarly, $\vdash \Gamma$ if and only if $\Vdash^{\mathsf{v}} \Gamma$.

The principal challenge of our work will be to repair the proof of the fundamental theorem after making our extensions. As mentioned before, we'll proceed by induction on the typing derivation $\Gamma \vdash \mathcal{J}$, and so it suffices to show that the premises of

each typing rule, given a suitable inductive hypothesis, imply the corresponding validity judgement. As an example, consider the successor typing rule for natural numbers:

$$\frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\ t : \mathbb{N}}\ \mathsf{suc}$$

The corresponding case of the fundamental theorem would have us prove that given $\Gamma \vdash t : \mathbb{N}$ and the inductive hypothesis $\Gamma \Vdash^{\mathsf{v}} t : \mathbb{N}$, we have $\Gamma \Vdash^{\mathsf{v}} \mathsf{suc}\ t : \mathbb{N}$, which is to say that $\mathsf{suc}\ t$ reduces to some WHNF of $\mathbb{N}$ in a way that respects substitution. Later on, when we're throwing opaque definitions into the mix, re-proving the fundamental theorem will largely amount to proving the new cases corresponding to the new typing rules we add.

With the fundamental theorem now in hand, we've obtained a way to reduce generalizations over all terms to generalizations over only normal forms. The simplest example of this is the normalization theorem:

Theorem 2.18 (Normalization)**.** Any well-typed term $\Gamma \vdash t : A$ reduces to some WHNF $\overline{t}$.

By the fundamental theorem, it follows from $\Gamma \vdash t : A$ that $\Gamma \Vdash^{\mathsf{v}} t : A$, from which it follows that $\Gamma \Vdash t : A$. Reducibility tells us that t reduces to a WHNF, and so we're done. Various other useful results also fall out of the fundamental theorem:

Theorem 2.19 (Canonicity)**.** In the empty context ε, if $\varepsilon \vdash t : A$, then t reduces to a canonical form of A.

Theorem 2.20 (Consistency)**.** In the empty context ε, the empty type $\bot$ is uninhabited.

Theorem 2.21 (Decidable Conversion)**.** Judgemental equality is decidable. That is:

- Given well-formed types $\Gamma \vdash A$ and $\Gamma \vdash B$, it is decidable whether or not $\Gamma \vdash A \equiv B$.
- Given well-typed terms $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$, it is decidable whether or not $\Gamma \vdash t \equiv u : A$.

Theorem 2.22 (Decidable Type Checking)**.** Typing is decidable for a certain "checkable" fragment of the language[a]. That is:

- Given a well-formed context $\vdash \Gamma$ and a checkable type A, it is decidable whether or not $\Gamma \vdash A$.
- Given a well-formed type $\Gamma \vdash A$ and a checkable term t, it is decidable whether or not $\Gamma \vdash t : A$.
- Given a well-formed context $\vdash \Gamma$ and an inferable term t, it is decidable whether or not there exists an A for which $\Gamma \vdash t : A$.
- Given a typing context Γ of checkable types, it is decidable whether or not $\vdash \Gamma$.

[a] Corresponding to bidirectional type checking with sparse annotations; more on this in §4.6.

2.3 Towards Opaque Definitions

To model opaque definitions, the idea will be to augment every typing judgement with a *definition context* ∇ analogous to the usual typing contexts Γ, but carrying information about definitions rather than variables. For a judgement $\Gamma \vdash \mathcal{J}$, we denote this by $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$. In this way, the typing rules can depend on the contents of the definition context; for example, the rule for well-typed definitions can look up a definition's type in ∇, much like the rule for well-typed variables can look up a variable's type in Γ. Of course, we'll also augment the logical relation accordingly, so that $\Gamma \Vdash \mathcal{J}$ becomes $\nabla \mathbin{»} \Gamma \Vdash \mathcal{J}$ and $\Gamma \Vdash^{\mathsf{v}} \mathcal{J}$ becomes $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} \mathcal{J}$.

Whereas a typical variable only needs to know its type, a definition also needs to know its definiens as well as its opacity in order to know how it should unfold. Moreover, because opaque definitions can unfold other opaque definitions, it also needs to know which of those other definitions it's unfolding in order to correctly type check. The responsibility of the definition context, then, is to store all of this information for later use by the typing rules. This design is loosely based on the implementation of opaque definitions in the Agda type checker [4], where information about in-scope definitions is held in a table in the type-checking environment[4].

A key limitation of this approach is that only top-level definitions are possible; because definitions in the definition context cannot depend on variables in the typing context, there's no way to model, for example, a definition containing a local variable. In fact, the dependency goes in the opposite direction: since the type of a local variable may contain a definition, the typing context must depend on the definition context!

One way to deal with this interdependency might be to use only a single heterogeneous context such that later entries in the context can depend on earlier entries regardless of whether they're variables or definitions. This is the approach taken by the Rocq proof assistant, which allows it to represent local lemmas in proofs [24]. While this works in principle, it complicates things considerably; for instance, how should de Bruijn indices interact with weakenings and substitutions when they can also refer to definitions rather than just variables? Because Agda itself only supports top-level opaque definitions, as well as for the sake of simplicity, we've opted to stick with a separate definition context for the present work.

2.4 Related Work

Opaque definitions have been studied and implemented in various dependently-typed settings, often without formal backing:

- We briefly discussed the Agda implementation of top-level opaque definitions above. As it stands, Agda does not yet have a full formalization of its metatheory. While there has been some recent work towards rectifying this [6], opaque definitions are a fairly new addition to the language[5] which have avoided formal-

[4] Technically, Agda keeps track of "opaque blocks", each of which may contain multiple opaque definitions, but the distinction is not really that important.

[5] The feature was implemented in May of 2023 during the 36th Agda Implementors' Meeting; see the relevant pull request here: https://github.com/agda/agda/pull/6628

ization thus far.

- In Rocq, opaque definitions are present in the "core language" as defined by the reference manual [24], but not in the calculus of (inductive) constructions on which it is originally based [8][17]—indeed, the δ-reduction rules as listed in the manual do not appear at all in the Co(I)C literature. Above, we discussed the heterogeneous context idea used by Rocq to represent local definitions; in this sense, their system is more general than ours, but will likely also be more difficult to formalize. There has been some effort towards formalizing/mechanizing Rocq's metatheory (including its extensions to the CoC) [19][18], but they generally operate on somewhat simplified versions of the core language that eschew the notion of opacity.
- In Lean, opacity is managed by the attribute system. When the "irreducible" attribute is applied to a definition, the type checker will refuse to unfold it; this can then be locally undone by removing the attribute with the `unseal` command [22]. However, reducibility information is discarded during elaboration to the core (kernel) language, and so opacity does not seem to be addressed in Lean metatheoretic formalization efforts, which generally focus on the kernel.
- Gratzer *et al.* [10] give a formalism for opaque definitions based on an elaboration to a language with extension types, which they have implemented in `cooltt`[6] [23]. Their system is similar to Rocq's in that definitions and variables both live in a single shared context, thus allowing for local definitions and unfoldings, but it differs in that it represents definitions as specially-typed variables[7] rather than as a separate kind of entity. They show normalization for their system, but the proof does not seem to be mechanized.
- Automath, one of the earliest systems for mechanized mathematics, implements definitions (known as "abbreviations") by introducing a "defining axiom" $d(\overline{x}) = t$ for each definition $d \triangleq \lambda\overline{x}.\, t$ and a corresponding δ-reduction rule $d(\overline{v}) \Rightarrow t[\overline{v}/\overline{x}]$ [25]. While the original Automath system did not feature opacity, modern reimplementations have added support for it [27]. Moreover, Automath has had a far-reaching influence on the design of other dependent type systems; Guidi [11], for example, mentions a "conditioned abbreviation" construction similar to an opaque definition.

The information-hiding approach to separation of concerns is by no means limited to the dependently-typed space; indeed, analogous constructions have long been used, even in non-dependent settings, to address the problem of software organization [16]:

- In many languages, the notion of an "abstract data type" or "interface" offers a similar form of separation of concerns: since the abstract type itself is not bound to any specific implementation, consumers must work with only that which is given in the type's interface in much the same way an outside party can only work with the interface to an opaque type definition.
- Harper and Lillibridge [12] gives an account of "translucent sum types" for rep-

[6] See the relevant pull request here: https://github.com/RedPRL/cooltt/pull/373

[7] A definition $\alpha \triangleq t : A$ elaborates to a variable of the extension type $\{A \mid \Upsilon_\alpha \hookrightarrow t\}$, a subtype of A whose terms must be definitionally equal to t in contexts where the "unfolding proposition" Υ_α holds.

resenting abstract data types. A translucent sum type is essentially a dependent record type for which each field's type can optionally be annotated with a definiens, which then forces that field to take on the given value[8]. The idea is that a sum can bundle up the data type's data along with its entire interface; fields can then be selectively made transparent by annotating them as such in the sum type.

- In the realm of algebraic semantics, the relationship between an equational signature Σ and its Σ-algebras is similar to that between an abstract data type and its implementations, in the sense that equational results on Σ (or, equivalently, on an initial algebra of Σ) extend to all Σ-algebras, giving a form of *representation independence* [14].

This is not an exhaustive review of prior literature by any means; the existing body of work on information-hiding is immense, and it would be nigh-impossible to cover it all here. Instead, we hope that the interested reader now knows of a few directions in which they might start exploring.

3 Formalizing Top-level Definitions

We now consider top-level definitions *without* opacity. Our goal in this section will be to formally define the definition contexts ∇, then to use them to define typing rules for definitions. After that, we will discuss how the presence of the definition context interacts with the logical relation and, in particular, the fundamental theorem.

3.1 The Formalism

To represent the definitions, we augment the term language with a new kind of term similar to a variable:

Definition 3.1. A **term** can be a definition name, represented as a de Bruijn *level*:

$$\text{Definition Name } \alpha, \beta ::= \alpha_0 \mid \alpha_1 \mid \dots$$

$$\text{Term } A, B, t, u ::= \alpha \mid \dots$$

Here, α_i refers to the definition at level i.

Why de Bruijn levels over indices? Unlike the usual typing context, the definition context is invariant under binders, and so we avoid an unnecessary dependency on the size of the context by using de Bruijn levels. Note also that terms are *not* well-scoped with respect to definitions; this is merely to avoid having to parameterize everything with a definition context size.

We can now define *definition contexts*, which are similar to typing contexts, but which additionally carry a definiens for each binding:

Definition 3.2. A **definition context** is a list of type annotations and definientia

[8] The fields might be thought of as having extension types $\{A \mid \psi \hookrightarrow t\}$ where ψ is one of the constant propositions $\top$ or $\bot$.

for definitions in the context:

$$\text{Definition Context } \nabla ::= \epsilon \mid \nabla \cdot (t : A)$$

Definition contexts are dependent in the sense that later entries may refer to definitions given in earlier ones.

Note that each entry in a definition context contains precisely the data that is given in an Agda definition; the entry $(t : A)$ can be thought of as the definition:

```
α : A – Type annotation
α = t – Definiens
```

We can also define inductive "maps-to" relations that let us peek into a definition context:

Definition 3.3. The **maps-to relations** are the relations:

$\alpha \mapsto t : A \in \nabla$	The name α refers to a definition with type annotation A and definiens t in ∇.
$\alpha \mapsto\!: A \in \nabla$	The name α refers to a definition with type annotation A in ∇.

They are inductively defined by:

$$\frac{\nabla \text{ has size } n}{\alpha_n \mapsto t : A \in \nabla \cdot (t : A)}\ \textsc{here} \qquad \frac{\alpha \mapsto t : A \in \nabla}{\alpha \mapsto t : A \in \nabla \cdot (u : B)}\ \textsc{there}$$

$$\frac{\nabla \text{ has size } n}{\alpha_n \mapsto\!: A \in \nabla \cdot (t : A)}\ \textsc{here}' \qquad \frac{\alpha \mapsto\!: A \in \nabla}{\alpha \mapsto\!: A \in \nabla \cdot (u : B)}\ \textsc{there}'$$

These relations will give us the language we need to state typing rules for our definitions.

To begin with, we need to define what it means for a definition context ∇ to be well-formed, which we denote by $\gg \nabla$. Such a ∇ is well-formed precisely when all of its definitions are well-typed in the empty context (being top-level definitions):

Definition 3.4. **Well-formedness for definition contexts** is defined inductively by:

$$\frac{}{\gg \epsilon}\ \textsc{empty} \qquad \frac{\gg \nabla \qquad \nabla \gg \varepsilon \vdash t : A}{\gg \nabla \cdot (t : A)}\ \textsc{extend}$$

We can then redefine the well-formedness of typing contexts $\vdash \Gamma$ to require as a premise a well-formed definition context, which we denote by $\nabla \gg\!\vdash \Gamma$:

Definition 3.5. **Well-formedness for typing contexts** is defined inductively by:

$$\frac{\gg \nabla}{\nabla \gg\!\vdash \varepsilon}\ \textsc{empty} \qquad \frac{\nabla \gg\!\vdash \Gamma \qquad \nabla \gg \Gamma \vdash A}{\nabla \gg\!\vdash \Gamma \cdot A}\ \textsc{extend}$$

None of the existing typing rules depend on ∇, and so they remain largely untouched (aside from slapping a ∇ onto the judgements). However, our new rules for definitions

will need to look into ∇, which they can do using the maps-to relations from above:

$$\frac{\nabla \gg\!\vdash \Gamma \qquad \alpha \mapsto\!: A \in \nabla}{\nabla \gg \Gamma \vdash \alpha : A}\ \textsc{defn}$$

$$\frac{\nabla \gg\!\vdash \Gamma \qquad \alpha \mapsto t : A \in \nabla}{\nabla \gg \Gamma \vdash \alpha \equiv t : A}\ \delta\textsc{-red} \qquad \frac{\nabla \gg\!\vdash \Gamma \qquad \alpha \mapsto t : A \in \nabla}{\nabla \gg \Gamma \vdash \alpha \Rightarrow t : A}\ \delta\textsc{-red}$$

Essentially, these rules state that a definition takes its type from its type annotation and is equal to/reduces to its definiens. We refer to the reduction step as "δ-reduction" in the style of Automath [25].

Note that because of the well-scopedness of terms in the formalization proper, it is technically necessary to weaken the definition types and terms to embed them into Γ in the conclusions of the above rules. However, since the definitions are well-scoped in the empty context ε and therefore contain no variables, weakenings are trivial for them, so we elide them here.

We also consider a notion of "definition context extension", analogous to a weakening for a typing context:

Definition 3.6. A **definition context extension** is a sequence ξ of definitions to be appended to a definition context. Such an extension is a well-formed extension $\xi \gg \nabla' \supseteq \nabla$ from ∇ to ∇' when each definition in the sequence is well-typed.

Using these rules, we can derive some "direct" results about definitions:

Lemma 3.7 (Uniqueness)**.** A definition has at most one type annotation and one definiens. That is:

- If $\alpha \mapsto t : A \in \nabla$ and $\alpha \mapsto u : B \in \nabla$, then $A = B$ and $t = u$.
- If $\alpha \mapsto\!: A \in \nabla$ and $\alpha \mapsto\!: B \in \nabla$, then $A = B$.

Lemma 3.8 (Weakening)**.** Typing judgements are preserved under weakening of the definition context. That is, if $\nabla \gg \Gamma \vdash \mathcal{J}$, then for any well-formed extension $\xi \gg \nabla' \supseteq \nabla$, we have $\nabla' \gg \Gamma \vdash \mathcal{J}$.

Lemma 3.9 (Well-formedness)**.** If $\gg \nabla$ and $\alpha \mapsto t : A \in \nabla$, then $\nabla \gg \varepsilon \vdash t : A$.

Note that the proof of the above well-formedness lemma (3.9) uses definitional weakening (3.8): it amounts to extracting the proof from $\gg \nabla$ that $\varepsilon \vdash t : A$ for some sub-context of ∇, then weakening it up to ∇. This is simple enough for now, but we'll soon see that proving an analogous lemma for validity is a bit more complicated.

In addition, the previous "direct" results from §2.1 still hold, albeit with minor changes to their proofs to account for the new typing rules.

3.2 Updating the Logical Relation

To attack the fundamental theorem, we'll start by augmenting the logical relation with definition contexts in the same way we did for the typing judgements:

Definition 3.10. The **logical relation for reducibility** consists of the relations:

$\nabla \mathbin{»} \Gamma \Vdash A$	A is a reducible type in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash t : A$	t is a reducible term of type A in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash A \equiv B$	A and B are reducibly equal types in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash t \equiv u : A$	t and u are reducibly equal terms of type A in ∇ and Γ.

Definition 3.11. The **validity judgements** are the relations:

$\nabla \mathbin{»}\Vdash^{\mathsf{v}} \Gamma$	Every variable in Γ has a valid type in ∇.
$\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} A$	A is a valid type in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} t : A$	t is a valid term of type A in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} A \equiv B$	A and B are validly equal as types in ∇ and Γ.
$\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} t \equiv u : A$	t and u are validly equal as terms of type A in ∇ and Γ.
$\nabla \mathbin{»} \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma$	σ is a valid substitution from Γ to Δ in ∇.
$\nabla \mathbin{»} \Delta \Vdash^{\mathsf{s}} \sigma_1 \equiv \sigma_2 : \Gamma$	σ_1 and σ_2 are validly equal substitutions from Γ to Δ in ∇.

These new relations are defined in mostly the same way as before, just with the definition context ∇ tacked on so that it can be passed into the underlying typing judgements. There's one key difference, and we'll address it when it comes up. For now, note how there is no validity variant of definition context well-formedness given here.

If we were to define validity for definition contexts $»^{\mathsf{v}}\ \nabla$ mutually-recursively with the rest of the validity judgements (since definition validity would be term validity, which implies context validity, which would imply definition context validity), we would run into positivity issues later down the line. However, we've seen that the logical relation's job is to classify WHNFs, which means that in a manner of speaking, any definitions will already have been unfolded away before the logical relation can ever see them. Thus, it turns out that we largely don't need validity for definitions, and so definition context well-formedness $» \nabla$ is sufficient for our purposes.

Nevertheless, validity for definitions will be useful for the cases of the fundamental theorem for definitions. Thus, we define it now, *after* the other validity judgements:

Definition 3.12. **Validity for definition contexts** is defined by the following recursion equations:

$$»^{\mathsf{v}}\ \epsilon = \top$$

$$»^{\mathsf{v}}\ (\nabla \cdot (t : A)) = (»^{\mathsf{v}}\ \nabla) \times (\nabla \mathbin{»} \varepsilon \Vdash^{\mathsf{v}} t : A)$$

Now, we can finally state the updated fundamental theorem:

Theorem 3.13 (Fundamental Theorem)**.** All of the following hold:

- $» \nabla$ if and only if $»^{\mathsf{v}}\ \nabla$.
- $\nabla \mathbin{»}\vdash \Gamma$ if and only if $\nabla \mathbin{»}\Vdash^{\mathsf{v}} \Gamma$.
- $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$ if and only if $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} \mathcal{J}$.

Proceeding by mutual induction, the first two statements follow from the third: we can handle each definition in a definition context by recursively applying the third statement

for term validity, and we can handle each variable in a typing context by recursively applying the third statement for type validity. It remains, then, to re-prove this third statement by addressing the cases for our two new definition typing rules.

We'll start with the case for the δ-reduction equality rule:

$$\frac{\nabla \gg\vdash \Gamma \qquad \alpha \mapsto t : A \in \nabla}{\nabla \gg \Gamma \vdash \alpha \equiv t : A}\ \delta\text{-RED}$$

By well-formedness, $\nabla \gg\vdash \Gamma$ gives us $\gg \nabla$, then the mutual-inductive hypothesis gives us $\gg^{\mathsf{v}} \nabla$ and $\nabla \gg\Vdash^{\mathsf{v}} \Gamma$. Then, it suffices to show that:

> **Lemma 3.14** (Validity of δ-reduction)**.** Given that $\gg^{\mathsf{v}} \nabla$ and $\nabla \gg\Vdash^{\mathsf{v}} \Gamma$, if $\alpha \mapsto t : A \in \nabla$, then $\nabla \gg\vdash \Gamma \Vdash^{\mathsf{v}} \alpha \equiv t : A$.

To make use of the assumption $\alpha \mapsto t : A \in \nabla$, we'll use a well-formedness lemma for valid definition contexts analogous to the one given earlier for well-formed ones (3.9):

> **Lemma 3.15** (Well-formedness)**.** If $\gg^{\mathsf{v}} \nabla$ and $\alpha \mapsto t : A \in \nabla$, then $\nabla \gg \varepsilon \Vdash^{\mathsf{v}} t : A$.

As before, we'd like a definitional weakening lemma for the proof—this time, for validity:

> **Lemma 3.16** (Weakening)**.** Validity judgements are preserved under weakening of the definition context. That is, if $\nabla \gg \Gamma \Vdash^{\mathsf{v}} \mathcal{J}$, then for any well-formed extension $\xi \gg \nabla' \supseteq \nabla$, we have $\nabla' \gg \Gamma \Vdash^{\mathsf{v}} \mathcal{J}$.

In proving this, we run into the same problem we previously encountered with weakening of the typing context (2.13)—namely, that reducibility occurs negatively in its own definition! Fortunately, we can use the same trick again to work around the problem: we'll just bake the definitional weakenings into validity. To illustrate, let's see how validity for terms has changed to accommodate these weakenings: $\nabla \gg \Gamma \Vdash^{\mathsf{v}} t : A$ holds roughly when:

- $\nabla \gg \Gamma \Vdash^{\mathsf{v}} A$.
- For any well-formed extension $\xi \gg \nabla' \supseteq \nabla$ and any validly equal substitutions $\nabla' \gg \Delta \Vdash^{\mathsf{s}} \sigma_1 \equiv \sigma_2$ from Γ to Δ, we have $\nabla' \gg \Delta \Vdash t[\sigma_1] \equiv t[\sigma_2] : A[\sigma_1]$.

The key insight is that because reducibility, and by extension validity, speak principally about a term's WHNF, any term that reduces to a valid term—and therefore has the same WHNF as a valid term—is also valid. Ergo, validity is preserved under weak head expansion:

> **Lemma 3.17** (Expansion)**.** Given that:
> - $\nabla \gg \Gamma \Vdash^{\mathsf{v}} u : A$
> - For any well-formed extension $\xi \gg \nabla' \supseteq \nabla$ and any valid substitution $\nabla' \gg \Delta \Vdash^{\mathsf{s}} \sigma : \Gamma$, we have $\nabla' \gg \Delta \vdash t[\sigma] \Rightarrow u[\sigma] : A[\sigma]$
>
> We can conclude that $\nabla \gg \Gamma \Vdash^{\mathsf{v}} t \equiv u : A$.

This gives us all we need: by well-formedness, we know that $\nabla \gg \Gamma \Vdash^{\mathsf{v}} t : A$, and so we can conclude by weak head expansion of δ-reduction that $\nabla \gg \Gamma \Vdash^{\mathsf{v}} \alpha \equiv t : A$.

Now, we move on to the case for the definition typing rule:

$$\frac{\nabla \gg\vdash \Gamma \qquad \alpha \mapsto\!: A \in \nabla}{\nabla \gg \Gamma \vdash \alpha : A}\ \textsc{defn}$$

As before, we apply the mutual-inductive hypothesis to the premises to get $\gg^{\mathsf{v}} \nabla$ and $\nabla \gg\Vdash^{\mathsf{v}} \Gamma$, and so our goal is to show:

> **Lemma 3.18** (Validity of Definitions)**.** Given that $\gg^{\mathsf{v}} \nabla$ and $\nabla \gg\Vdash^{\mathsf{v}} \Gamma$, if $\alpha \mapsto\!: A \in \nabla$, then $\nabla \gg\vdash \Gamma \Vdash^{\mathsf{v}} \alpha : A$.

As we are not yet considering opacity, a definition will always reduce to its definiens, and so there must be some term t for which $\alpha \mapsto t : A \in \nabla$. Then, we get the desired result by more or less the same argument as for δ-reduction: since α reduces to t by δ-reduction, they have the same WHNF, and so α is valid whenever t is.

4 Formalizing Opacity

Now that definitions are out of the way, we can begin modeling *opacity*. We first describe how definition contexts are modified to carry information about opacity as well as how the typing rules use this information to restrict unfolding. We then discuss the impact of these changes on normalization and the resulting implications for the logical relation.

4.1 The Formalism, Take Two

To track the opacity of definitions, we mark each entry in a definition context as either transparent or opaque. In the latter case, since opaque definitions can unfold other opaque definitions (as part of the "interface"), we also track which of those other definitions are unfolded:

> **Definition 4.1.** In a definition context ∇ with n entries, an **opacity** for the $(n+1)$th entry is either tra, indicating that it is transparent; or $\mathsf{opa}(\varphi)$ for an n-bitvector φ, indicating that it is opaque and that it unfolds the definitions in ∇ indicated by φ:
>
> $$\text{Opacity } \omega ::= \mathsf{tra} \mid \mathsf{opa}(\varphi)$$
>
> $$\text{Unfolding Vector } \varphi ::= \varepsilon \mid \varphi 0 \mid \varphi 1$$

> **Definition 4.2.** A **definition context** is a list of definition bindings, where each entry consists of a type annotation, a definiens, and an opacity:
>
> $$\text{Definition Context } \nabla ::= \epsilon \mid \nabla \cdot_{\omega} (t : A)$$

As before, the data carried by an entry in a definition context reflects the information given in an opaque Agda definition: the opa and tra annotations represent the presence or absence of the `opaque` keyword, and in the $\mathsf{opa}(\varphi)$ case, the unfolding vector φ represents the presence of `unfolding` clauses:

```
opaque – Opacity
  unfolding β – Unfolding vector
  α : A – Type annotation
  α = t – Definiens
```

To distinguish between transparent and opaque definitions in the typing rules, we'll split up the maps-to relations:

Definition 4.3. The **maps-to relations** are the relations:

$\alpha \mapsto t : A \in \nabla$	α refers to a transparent definition with type annotation A and definiens t in ∇.
$\alpha \mapsto \emptyset : A \in \nabla$	α refers to an opaque definition with type annotation A in ∇.
$\alpha \mapsto\!: A \in \nabla$	α refers to a definition with type annotation A in ∇.

They are inductively defined by:

$$\frac{\nabla \text{ has size } n}{\alpha_n \mapsto t : A \in \nabla \cdot_{\mathsf{tra}} (t : A)}\ \textsc{here} \qquad \frac{\alpha \mapsto t : A \in \nabla}{\alpha \mapsto t : A \in \nabla \cdot_{\omega} (u : B)}\ \textsc{there}$$

$$\frac{\nabla \text{ has size } n}{\alpha_n \mapsto \emptyset : A \in \nabla \cdot_{\mathsf{opa}(\varphi)} (t : A)}\ \textsc{here}' \qquad \frac{\alpha \mapsto \emptyset : A \in \nabla}{\alpha \mapsto \emptyset : A \in \nabla \cdot_{\omega} (u : B)}\ \textsc{there}'$$

$$\frac{\nabla \text{ has size } n}{\alpha_n \mapsto\!: A \in \nabla \cdot_{\omega} (t : A)}\ \textsc{here}'' \qquad \frac{\alpha \mapsto\!: A \in \nabla}{\alpha \mapsto\!: A \in \nabla \cdot_{\omega} (u : B)}\ \textsc{there}''$$

With a bit of easy induction, we can move back and forth between these relations:

Lemma 4.4 (Forget Definiens)**.** If $\alpha \mapsto t : A \in \nabla$, then $\alpha \mapsto\!: A \in \nabla$.

Lemma 4.5 (Forget $\emptyset$)**.** If $\alpha \mapsto \emptyset : A \in \nabla$, then $\alpha \mapsto\!: A \in \nabla$.

Lemma 4.6 (Dichotomy)**.** If $\alpha \mapsto\!: A \in \nabla$, then either $\alpha \mapsto \emptyset : A \in \nabla$ or there is some t for which $\alpha \mapsto t : A \in \nabla$.

We also define "glassification", which makes all definitions in a context transparent:

Definition 4.7. The **glassify** operation is given by the following recursion equations:

$$\mathsf{glassify}(\epsilon) = \epsilon$$
$$\mathsf{glassify}(\nabla \cdot_{\omega} (t : A)) = \mathsf{glassify}(\nabla) \cdot_{\mathsf{tra}} (t : A)$$

As we'll see in §4.5, certain results will only hold in glass (i.e. fully transparent) contexts, and so glassification will give us the language to express them. For now, we can prove a simple lemma that encodes the intention of glassification:

Lemma 4.8 (Glassification)**.** If $\alpha \mapsto\!: A \in \nabla$, then there is some t for which $\alpha \mapsto t : A \in \mathsf{glassify}(\nabla)$.

This follows directly from the definition of glassification.

While checking that an opaque definition $\alpha \triangleq_{\mathsf{opa}(\varphi)} t : A$ is well-formed, we'll now need to unfold other definitions in the context as specified by φ. To do this, we define an unfolding relation $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$, which expresses that the unfolding vector φ "unfolds" a definition context ∇ to a new one ∇' by making the referenced definitions transparent:

Definition 4.9. The **unfolding relation** is defined inductively by:

$$\frac{}{\varepsilon » \epsilon \leftsquigarrow \epsilon}\ \textsc{empty} \qquad \frac{\varphi » \nabla' \leftsquigarrow \nabla}{\varphi 0 » \nabla' \cdot_\omega (t : A) \leftsquigarrow \nabla \cdot_\omega (t : A)}\ \textsc{no}$$

$$\frac{\varphi \sqcup \varphi' » \nabla' \leftsquigarrow \nabla}{\varphi 1 » \nabla' \cdot_{\mathsf{tra}} (t : A) \leftsquigarrow \nabla \cdot_{\mathsf{opa}(\varphi')} (t : A)}\ \textsc{yes-opa}$$

$$\frac{\varphi » \nabla' \leftsquigarrow \nabla}{\varphi 1 » \nabla' \cdot_{\mathsf{tra}} (t : A) \leftsquigarrow \nabla \cdot_{\mathsf{tra}} (t : A)}\ \textsc{yes-tra}$$

Here, $\sqcup$ is a binary operator on unfolding vectors that specifies how transitive unfolding is handled. It does this by specifying, at each unfolding step, how the remainder of the unfolding vector (on the left) is modified by the unfolding vector of the definition being unfolded (on the right); more on this in §4.2. We can now update the well-formedness judgement with unfoldings:

Definition 4.10. **Well-formedness for definition contexts** is defined inductively by:

$$\frac{}{» \epsilon}\ \textsc{empty} \qquad \frac{» \nabla \qquad \nabla » \varepsilon \vdash t : A}{» \nabla \cdot_{\mathsf{tra}} (t : A)}\ \textsc{extend-tra}$$

$$\frac{» \nabla \qquad \nabla » \varepsilon \vdash A \qquad \varphi » \nabla' \leftsquigarrow \nabla \qquad \nabla' » \varepsilon \vdash t : A}{» \nabla \cdot_{\mathsf{opa}(\varphi)} (t : A)}\ \textsc{extend-opa}$$

Note that even in the opaque case, the type of the definition must check in the *unfolded* context; this reflects the fact that the type signature of an "interface" definition must not leak the "implementation details" of the other definitions it unfolds. For example, imagine if we had tried to define an interface in §1 of type `(0 , 0) ≡ℤ (1 , 1)`. Since $\mathbb{Z}$ is not definitionally equal to $\mathbb{N} \times \mathbb{N}$ to an outside observer, to them, this type would be ill-formed!

At a glance, the typing rules for definitions look exactly the same as before:

$$\frac{\nabla »\vdash \Gamma \qquad \alpha \mapsto\!: A \in \nabla}{\nabla » \Gamma \vdash \alpha : A}\ \textsc{defn}$$

$$\frac{\nabla »\vdash \Gamma \qquad \alpha \mapsto t : A \in \nabla}{\nabla » \Gamma \vdash \alpha \equiv t : A}\ \delta\textsc{-red} \qquad \frac{\nabla »\vdash \Gamma \qquad \alpha \mapsto t : A \in \nabla}{\nabla » \Gamma \vdash \alpha \Rightarrow t : A}\ \delta\textsc{-red}$$

The difference is in the meaning of the maps-to relations, which we've updated to respect opacity. Since definitions are typed in the same way regardless of opacity, the DEFN rule uses the $\mapsto\!:$ variant, which is opacity-agnostic. In contrast, because opaque definitions shouldn't unfold, the δ-RED rules use the $\mapsto$ variant, which only applies to transparent definitions.

There are two useful classes of context-modification lemmas which we can now prove by induction on typing judgements:

Lemma 4.11 (Weakening)**.** Typing judgements are preserved under weakening of the definition context. That is, if $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, then for any well-formed extension $\xi \mathbin{»} \nabla' \supseteq \nabla$, we have $\nabla' \mathbin{»} \Gamma \vdash \mathcal{J}$.

Lemma 4.12 (Glassification)**.** Typing judgements are preserved by glassification. That is, if $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, then $\mathsf{glassify}(\nabla) \mathbin{»} \Gamma \vdash \mathcal{J}$.

Since extensions of the definition context must now also carry information about the opacity of the new definitions, the weakening lemmas are slightly more complicated than before, but not in any particularly notable way.

4.2 Transitive Unfolding

When unfolding opaque definitions, there is the question of *transitive* unfoldings; consider the following Agda code:

```
opaque
  A : Set
  A = ℕ
opaque
  unfolding A
  a : A
  a = 0
```

```
opaque
  unfolding a
  B : Set
  B = Id A a 0

  b : B
  b = rfl
```

When checking B and b, Agda will unfold a as specified by the `unfolding` clause, then *transitively* unfold A as specified by the `unfolding` clause for a. This transitive unfolding is necessary for the unfolding of a to be well-typed; otherwise, if we unfolded only a and not A, both $a \Rightarrow 0 : A$ and $a \Rightarrow 0 : \mathbb{N}$ would be ill-typed, as neither $0 : A$ nor $a : \mathbb{N}$ would be deducible. Note also that we would not have been able to write $b : \mathsf{Id}\ \mathbb{N}\ a\ 0$ on its own: since we can't deduce that a and 0 have the same type in the un-unfolded context, we aren't able to form the identity type at all.

To handle these transitive unfoldings in our system, we parameterize the theory with a binary operator $\sqcup$ on unfolding vectors. Recall the case of the unfolding relation for unfolding opaque definitions:

$$\frac{\varphi \sqcup \varphi' \mathbin{»} \nabla' \leftsquigarrow \nabla}{\varphi 1 \mathbin{»} \nabla' \cdot_{\mathsf{tra}} (t : A) \leftsquigarrow \nabla \cdot_{\mathsf{opa}(\varphi')} (t : A)}\ \textsc{yes-opa}$$

The idea is that when $\varphi 1$ unfolds the opaque definition $\alpha \triangleq_{\mathsf{opa}(\varphi')} t : A$ in $\nabla \cdot \alpha$, the remainder of the unfolding φ may be modified by the unfolding vector φ' of α via the $\sqcup$ operator, which then changes how the remainder of the context ∇ is unfolded in the premise $\varphi \sqcup \varphi' \mathbin{»} \nabla' \leftsquigarrow \nabla$.

Since our definition of unfolding is relational, we could, in principle, pick just about any binary operator for $\sqcup$; it would merely restrict which unfoldings we can prove valid and

therefore which contexts we can prove well-formed. Even with an ill-conditioned choice of $\sqcup$, any bad unfolding that unfolds a definition without its dependencies will simply create a definition context that cannot be proven well-formed and which is therefore ineligible to participate in typing derivations. However, we expect that most choices of $\sqcup$ yield systems which are either nonsensical or completely unusable.

We concern ourselves with only two choices for $\sqcup$:

$$\varphi \sqcup_{\text{transitive}} \varphi' = \varphi \vee \varphi' \qquad\qquad \varphi \sqcup_{\text{explicit}} \varphi' = \varphi$$

With $\vee$ denoting bitwise disjunction, $\sqcup_{\text{transitive}}$, which merges the definition's unfolding vector into the running one, models a system where unfolding a definition also unfolds the closure of its dependencies, as in Agda. On the other hand, $\sqcup_{\text{explicit}}$, which just discards the definition's unfolding vector, models a system where *no* transitive dependencies are unfolded, and so all other definitions depended on by a definition must be explicitly unfolded.

With transitive unfolding, it is impossible to construct an unfolding vector that unfolds a well-formed definition context to an ill-formed one:

Theorem 4.13 (Unfolding)**.** With $\sqcup_{\text{transitive}}$, typing judgements are preserved under unfolding. That is, if $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, then for any well-formed unfolding $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$, we have $\nabla' \mathbin{»} \Gamma \vdash \mathcal{J}$.

In contrast, the same does not hold for explicit unfolding:

Counterexample 4.14 (Unfolding)**.** With $\sqcup_{\text{explicit}}$, there exist definition contexts ∇ and ∇' and a well-formed unfolding $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$ such that $\mathbin{»} \nabla$, but not $\mathbin{»} \nabla'$.

This behaviour is analogous to the observation made by Gratzer *et al.* [10] that it's easy to lose subject reduction without transitive unfolding. In our system, we do manage to maintain subject reduction, but only on a technicality: because the definition context becomes ill-formed after a bad unfolding, the reduction cannot even occur.

4.3 Updating the Logical Relation, Take Two

"Neutrality", generally speaking, refers to terms that are not in canonical form, but which cannot reduce any further. Traditionally, this has just been identified with terms blocked on variables, but the introduction of opaque definitions, which don't unfold and therefore can't reduce, will necessitate a reformulation:

Definition 4.15. A term is **neutral** if it has either a variable or an opaque definition in its head position[a].

[a] In the formalization proper, it's sometimes useful to distinguish between terms blocked on variables and those blocked on definitions, and so neutrality is parametric over a proposition expressing whether variables are included or not; we elide this here for simplicity.

Naturally, this will also change—semantically, at least—what it means for a term to be in WHNF:

Definition 4.16. A term is in **weak head normal form** (WHNF) if it is either neutral or a constructor application.

The logical relation itself, both in reducibility and validity, does not change all that much with this reformulation outside of some minor shuffling around to accommodate the new definition of WHNFs. Moreover, since neutrals blocked on opaque definitions have more or less the same reduction semantics as neutrals blocked on variables, most of the same arguments for the fundamental theorem go through with minimal changes. The two key exceptions to this are the cases for well-typed definitions, which no longer reduces to δ-reduction; and for equality reflection, which acts directly on neutral terms.

Recall that to discharge the case for well-typed definitions, it suffices to prove the following validity lemma (previously Lemma 3.18):

Lemma 4.17 (Validity of Definitions)**.** Given that $\Vdash^{\mathsf{v}} \Delta$ and $\Delta \Vdash^{\mathsf{v}} \Gamma$, if $\alpha \mapsto A \in \Delta$, then $\Delta \Vdash \Gamma \Vdash^{\mathsf{v}} \alpha : A$.

In §3.2, we noted that α must always unfold to some t, which allowed us to reduce this lemma to validity of δ-reduction (3.14). In light of opacity, however, we can no longer guarantee that α unfolds at all! Fortunately, we can still use this argument when the definition *does* unfold, and so we can proceed like this: by the dichotomy principle for the maps-to relations (4.6), we either have that $\alpha \mapsto \emptyset : A \in \Delta$ or that there is some t for which $\alpha \mapsto t : A \in \Delta$. The latter case reduces to validity of δ-reduction, and so it now suffices to show validity for only *opaque* definitions. Since opaque definitions are neutral, they are already in WHNF, and so reducibility is trivial. Moreover, because definition names contain no variables, they are invariant under substitution, and so validity is trivial.

4.4 Equality Reflection

Equality reflection, on the other hand, poses a bigger problem. Our language is equipped with the usual Martin-Löf identity type $\mathsf{Id}\ A\ t\ u$ expressing the propositional equality of terms t and u in type A [13]. Its only introduction rule is reflexivity:

$$\frac{\Delta \Vdash \Gamma \vdash t : A}{\Delta \Vdash \Gamma \vdash \mathsf{refl} : \mathsf{Id}\ A\ t\ t}\ \textsc{rfl}$$

The equality reflection principle states that internal proofs of identity can be lifted to judgemental equalities [20], and is encoded by an optional equality rule:

$$\frac{\Delta \Vdash \Gamma \vdash v : \mathsf{Id}\ A\ t\ u}{\Delta \Vdash \Gamma \vdash t \equiv u : A}\ \textsc{equality-reflection}$$

Crucially, validity for equality reflection is restricted to closed terms, since equality reflection would otherwise contradict normalization:

Counterexample 4.18 (Normalization with Equality Reflection)**.** With equality reflection, there exists a well-formed type $\Delta \Vdash \Gamma \vdash A$ in a non-empty context Γ for which A does not reduce to any WHNF.

Thus, the desired result, which holds for the formalization *without* opacity, is:

Lemma 4.19 (Validity of Equality Reflection)**.** Given $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} v : \mathsf{Id}\ A\ t\ u$ with v not blocked on a variable, we have $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} t \equiv u : A$.

The argument, prior to opacity, was as follows: because v is reducible, it must normalize to a WHNF, which will either be a canonical form of $\mathsf{Id}\ A\ t\ u$—namely, the reflexivity constructor rfl—or a neutral term. Since v is not blocked on a variable by assumption, it cannot be neutral, and so it can only be the case that it's rfl, in which case t and u must be syntactically equal and therefore judgementally equal. The issue, then, is that by expanding neutrality to include opaque definitions, we can no longer conclude that v is not neutral, since it could also be blocked on an opaque definition. Moreover, it's not clear what to do with such a blocked term: we can't do any syntactic unification as in the rfl case, and it's difficult to reason about the blocking definition within the confines of the logical relation, since it's already in WHNF.

For now, our way around this problem is to simply assert that equality reflection is incompatible with opaque definitions, which trivializes the proof:

Lemma 4.20 (Validity of Equality Reflection)**.** Given $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} v : \mathsf{Id}\ A\ t\ u$ with v not blocked on a variable and opaque definitions disallowed, we have $\nabla \mathbin{»} \Gamma \Vdash^{\mathsf{v}} t \equiv u : A$.

In some sense, this is a reasonable compromise: opaque definitions are generally used to improve usability (e.g. by enforcing separation of concerns and reducing type-checking burden), whereas equality reflection, which breaks normalization and which is known to make conversion checking undecidable [20], tends to destroy usability entirely. This is not a completely satisfactory conclusion, however, and we leave the resolution to possible future work in §5.

4.5 Consequences of the Fundamental Theorem

The same results mentioned in §2.2 still hold, but with the minor caveat that results about canonical forms must be stated in *glass* contexts to avoid getting stuck on opaque definitions. For instance, canonicity glassifies its context:

Theorem 4.21 (Canonicity)**.** If $\nabla \mathbin{»} \varepsilon \vdash t : A$, then t reduces to a canonical form of A in $\mathsf{glassify}(\nabla)$.

A particularly interesting example of this is canonicity for identity types, from which we get the following theorem:

Theorem 4.22 (Pseudo-reflection)**.** If $\nabla \mathbin{»} \varepsilon \vdash v : \mathsf{Id}\ A\ t\ u$, then $\mathsf{glassify}(\nabla) \mathbin{»} \varepsilon \vdash t \equiv u : A$.

This holds because by canonicity, any term of $\mathsf{Id}\ A\ t\ u$ must normalize to rfl, which allows us to syntactically unify t and u. But why does this have to occur in a glass context? Consider the following Agda code, where $\nabla = \epsilon \cdot x \cdot \mathrm{eq}$:

```
opaque
  x : ℕ
  x = 0
```

```
opaque
  unfolding x
  eq : Id ℕ x 0
  eq = rfl
```

This is well-typed because x reduces to 0 after unfolding it, and so rfl is a valid constructor for $\mathsf{Id}\ \mathbb{N}\ x\ 0 \equiv \mathsf{Id}\ \mathbb{N}\ 0\ 0$. However, we *cannot* lift eq to a judgemental equality in a non-glass ∇ and empty ε! Because x is opaque, it is neutral, which precludes it from being equal to a canonical form like 0. Thus:

Counterexample 4.23 (Non-glass Pseudo-reflection)**.** There exists a definition context ∇, type A, and terms t, u, and v such that $\nabla \gg \varepsilon \vdash v : \mathsf{Id}\ A\ t\ u$, but not $\nabla \gg \varepsilon \vdash t \equiv u : A$.

A few other interesting results:

Theorem 4.24 (Consistency)**.** In the empty context ε, the empty type $\bot$ is uninhabited.

Theorem 4.25 (Definition Consistency)**.** For a well-formed definition context $\gg \nabla$, there can be no definitions in ∇ of the empty type $\bot$.

Theorem 4.26 (Type Normalization)**.** Any well-formed type $\nabla \gg \Gamma \vdash A$ reduces to some WHNF.

Theorem 4.27 (Normalization)**.** Any well-typed term $\nabla \gg \Gamma \vdash t : A$ reduces to some WHNF.

Theorem 4.28 (Decidable Conversion)**.** Judgemental equality is decidable. That is:

- Given well-formed types $\nabla \gg \Gamma \vdash A$ and $\nabla \gg \Gamma \vdash B$, it is decidable whether or not $\nabla \gg \Gamma \vdash A \equiv B$.
- Given well-typed terms $\nabla \gg \Gamma \vdash t : A$ and $\nabla \gg \Gamma \vdash u : A$, it is decidable whether or not $\nabla \gg \Gamma \vdash t \equiv u : A$.

4.6 Deciding Type Checking With Unfolding

The proof of decidable type checking in the formalization is given by an implementation of a bidirectional type checker, which serves as a decision procedure. Since we try to avoid explicit type annotations in our term language, we suffer from some of the typical pitfalls of bidirectional type checking, namely that certain terms don't carry enough intrinsic type information to unambiguously decide local well-typedness [9, §4.5.4]. Thus, we restrict our decidability result to the fragment of the language where we *do* have sufficient type information, which we refer to as the "checkable" fragment:

Theorem 4.29 (Decidable Type Checking)**.** Typing is decidable for a certain "checkable" fragment of the language. That is:

- Given a well-formed context $\nabla \gg\vdash \Gamma$ and a checkable type A, it is decidable whether or not $\nabla \gg \Gamma \vdash A$.
- Given a well-formed type $\nabla \gg \Gamma \vdash A$ and a checkable term t, it is decidable whether or not $\nabla \gg \Gamma \vdash t : A$.
- Given a well-formed context $\nabla \gg\vdash \Gamma$ and an inferable term t, it is decidable whether or not there exists an A for which $\nabla \gg \Gamma \vdash t : A$.
- Given a definition context ∇ of checkable types and terms, it is decidable whether or not $\gg \nabla$.

- Given a well-formed definition context $\gg \nabla$ and a typing context Γ of checkable types, it is decidable whether or not $\nabla \gg\!\vdash \Gamma$.

There is some interesting interaction between the type checker and the choice of the $\sqcup$ operator for transitive unfolding, particularly in the procedure for checking definition contexts. The algorithm for deciding $\gg \nabla \cdot_{\mathsf{opa}(\varphi)} (t : A)$ is roughly as follows:

1. Check whether $\gg \nabla$; if not, return "no".
2. Check whether $\nabla \gg \varepsilon \vdash A$; if not, return "no".
3. Compute the unfolded ∇' such that $\varphi \gg \nabla' \backsim \nabla$.
4. Check whether $\gg \nabla'$; if not, return "no".
5. Check whether $\nabla' \gg \varepsilon \vdash A$; if not, return "no".
6. Check whether $\nabla' \gg \varepsilon \vdash t : A$; if not, return "no".
7. Return "yes" using the EXTEND-OPA rule.

Step *(4)* is of particular note: since we check for well-formedness of the unfolded context, we preclude any poor behaviour that might have been caused by an ill-conditioned choice of $\sqcup$ (a loss of subject reduction, for instance). Instead, if a bad unfolding ever occurs, this step will fail, and the type checker can terminate with an error message. In this sense, it's perfectly "safe" to use an ill-conditioned $\sqcup$ so long as the programmer is willing to put up with it.

On the other hand, step *(4)* has some very serious performance implications: since each well-formedness check for a definition context of length n now makes *two* recursive calls of length $n-1$, it worsens the time complexity *exponentially* in the length of ∇! This is a strong pragmatic argument for the use of a well-behaved operator like $\sqcup_{\text{transitive}}$ which, by virtue of stability under unfolding (4.13), can completely skip step *(4)*.

5 Conclusion

We have presented a fully-mechanized formal characterization of opaque top-level definitions in the style of Agda. We have given syntax and semantics for these definitions then shown, through a logical relation for reducibility, that they enjoy many desirable metatheoretic properties. We've also discussed certain design decisions related to opacity, especially with regard to transitive unfolding and equality reflection.

We note here some opportunities for future work based on this project, with particular attention to the limitations we listed all the way back in §1:

- **Support for local definitions.** Our system only models definitions at the top level, but it *should* be possible to expand to a more flexible form of definition. In §2.3, we speculated about a heterogeneous typing context interspersing definitions and variables; this could be one avenue of investigation. The extension type approach of Gratzer *et al.* [10] also has some appeal, especially because support for extension types would also be useful for a possible future formalization of cubical type theory [7].
- **Support for graded modal type theory.** The graded portion of the graded-type-theory project, which we have largely ignored in this work, gives a "resourcing

system" for tracking the usage of terms based on a semiring of grades [5][15]. This enables a wide range of static analyses—for example, an erasure analysis that justifies erasing compilation [1]. We expect that the function of the resourcing system will be mostly orthogonal to opaque definitions, but there is some amount of nuance required in defining "usage" for definition terms.

- **Support for equality reflection with opacity.** As discussed in §4.4, equality reflection poses a proof engineering problem in the proof of the validity lemma (4.20). In principle, it *should* be possible to use the same strategy of reducing every identity term to rfl since opaque definitions, unlike variables, must have a well-typed definiens. The problem is just that because opaque definitions are in WHNF, the reducibility relation gets "stuck" on them: it doesn't see any information about the definiens. It may be possible to get this information from a proof of validity for the definition context, but we suspect that it will require some sleight-of-hand with unfoldings. Another possibility might be to carry auxiliary information about definientia in the reducibility relation—something to the effect of "this term is in WHNF, but if we *could* unfold it, this is what we would get".
- **Structured refactoring tools for Agda.** In the course of this Agda formalization project, we encountered a great deal of "mechanical" work related to our restructuring of the original graded-type-theory formalisms. For example, after augmenting the typing judgements $\Gamma \vdash \mathcal{J}$ with a definition context $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, it became necessary to update every usage of the corresponding mixfix operators (⊢_, _⊢_, _⊢_∷_, etc.) with an additional parameter (as in _»⊢_, _»_⊢_, _»_⊢_∷_). We performed these kinds of refactorings with a large number of find-and-replace operations against a collection of complex regex patterns[9], fixing any edge cases by hand. Although we made this work well enough for this project, it's not particularly robust, and it's certainly not very user-friendly; a better alternative would be to use Agda's own parsing facilities to perform these kinds of refactorings in a more principled, structured way. While there has been some research in the area of refactoring for dependently-typed languages [26][21], there has not yet been much attention to mixfix operators and no production-ready tool is currently available for Agda.

[9] For instance, the following pattern identifies most usages of typing, reducibility, and validity judgements in the original graded-type-theory codebase:

```
(^|[\s\(])([HΓΔε][₁₂₃₄']?(?: • [\w\sˢʷ₁₂₃₄'ℕ]+)*\s+)([A-Z]+\.)?
((?:⊢(?:[≅~]|ˢʷ|n[ef]?)?|⊩\S*)(?:\s|$))
```

References

[1] A. Abel, N. A. Danielsson, and O. Eriksson, "A graded modal dependent type theory with a universe and erasure, formalized," *Proc. ACM Program. Lang.*, vol. 7, no. ICFP, Aug. 2023. DOI: 10.1145/3607862.

[2] A. Abel, J. Öhman, and A. Vezzosi, "Decidability of conversion for type theory in type theory," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017. DOI: 10.1145/3158111.

[3] A. Abel *et al.*, *Graded Type Theory*, 2025. [Online]. Available: https://github.com/graded-type-theory/graded-type-theory.

[4] Agda Developers, *Agda*, version 2.8.0. [Online]. Available: https://agda.readthedocs.io/.

[5] R. Atkey, "Syntax and semantics of quantitative type theory," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18, Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 56–65, ISBN: 9781450355834. DOI: 10.1145/3209108.3209189.

[6] J. Cockx. "Agda core: The dream and the reality. "[Online]. Available: https://jesper.sikanda.be/posts/agda-core.html.

[7] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, *Cubical type theory: A constructive interpretation of the univalence axiom*, 2016. arXiv: 1611.02108 [cs.LO]. [Online]. Available: https://arxiv.org/abs/1611.02108.

[8] T. Coquand and G. Huet, "The calculus of constructions," Ph.D. dissertation, INRIA, 1986.

[9] J. Dunfield and N. Krishnaswami, "Bidirectional typing," *ACM Computing Surveys*, vol. 54, no. 5, May 2021, ISSN: 0360-0300. DOI: 10.1145/3450952.

[10] D. Gratzer, J. Sterling, C. Angiuli, T. Coquand, and L. Birkedal, *Controlling unfolding in type theory*, 2022. arXiv: 2210.05420 [cs.LO]. [Online]. Available: https://arxiv.org/abs/2210.05420.

[11] F. Guidi, "The formal system λδ," *ACM Trans. Comput. Logic*, vol. 11, no. 1, Nov. 2009, ISSN: 1529-3785. DOI: 10.1145/1614431.1614436.

[12] R. Harper and M. Lillibridge, "A type-theoretic approach to higher-order modules with sharing," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '94, Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 123–137, ISBN: 0897916360. DOI: 10.1145/174675.176927.

[13] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Logic Colloquium '73*, ser. Studies in Logic and the Foundations of Mathematics, H. Rose and J. Shepherdson, Eds., vol. 80, Elsevier, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1.

[14] J. Meseguer and J. A. Goguen, "Initiality, induction, and computability," in *Algebraic Methods in Semantics*. USA: Cambridge University Press, 1986, pp. 459–541, ISBN: 0521263935.

[15] B. Moon, H. Eades III, and D. Orchard, "Graded modal dependent type theory," in *European Symposium on Programming*, Springer International Publishing Cham, 2021, pp. 462–490.

[16] D. L. Parnas, "Information distribution aspects of design methodology," 1971.

[17] C. Paulin-Mohring, "Inductive definitions in the system coq rules and properties," in *Typed Lambda Calculi and Applications*, M. Bezem and J. F. Groote, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 328–345, ISBN: 978-3-540-47586-6.

[18] M. Sozeau, Y. Forster, M. Lennon-Bertrand, J. Nielsen, N. Tabareau, and T. Winterhalter, "Correct and complete type checking and certified erasure for coq, in coq," *J. ACM*, vol. 72, no. 1, Jan. 2025, ISSN: 0004-5411. DOI: 10.1145/3706056.

[19] M. Sozeau *et al.*, *Metarocq/metarocq: Metarocq 1.4 for rocq 9.0*, version v1.4-9.0, Mar. 2025. DOI: 10.5281/zenodo.15088961.

[20] T. Streicher, "Investigations into intensional type theory," *Habilitiation Thesis, Ludwig Maximilian Universität*, p. 57, 1993.

[21] K. Struik, "Refactoring with confidence," Bachelor's Thesis, Delft University of Technology, 2023.

[22] The Lean Development Team, *The Lean language reference*, version 4.21.0-rc3. [Online]. Available: https://lean-lang.org/doc/reference/4.21.0-rc3/.

[23] The RedPRL Development Team, *cooltt*, 2020. [Online]. Available: https://github.com/RedPRL/cooltt.

[24] The Rocq Development Team, *The Rocq reference manual – release 9.0*, https://rocq-prover.org/doc/v9.0/refman, 2025.

[25] D. van Daalen, "The language theory of automath," English, Phd Thesis 2 (Research NOT TU/e / Graduation TU/e), Mathematics and Computer Science, 1980. DOI: 10.6100/IR85774.

[26] K. Wibergh, "Automatic refactoring for Agda," Master's Thesis, Chalmers University of Technology and University of Gothenburg, 2019.

[27] F. Wiedijk, "A new implementation of automath," *J. Autom. Reason.*, vol. 29, no. 3–4, pp. 365–387, Jan. 2003, ISSN: 0168-7433. DOI: 10.1023/A:1021983302516.