A Formalization of Opaque Definitions for a Dependent Type Theory

A Formalization of Opaque Definitions for a Dependent Type Theory

Nils Anders Danielsson
nad@cse.gu.se
Department of Computer Science and Engineering
University of Gothenburg and Chalmers University of Technology
Gothenburg, Sweden

Eve Geng
me@evening.st
Chalmers University of Technology
Gothenburg, Sweden

Abstract

Definitions allow for reuse of code. Typical type-checkers for dependently typed programming languages automatically unfold definitions, but excessive unfolding can lead to types that are hard to read, or performance issues. Such problems can be mitigated through the use of opaque definitions, which give the programmer control over when unfolding is allowed. However, subject reduction fails to hold for certain designs.

We study the metatheory of a type theory with opaque definitions, inspired by Agda. We give typing and reduction rules and show that the type theory enjoys properties like subject reduction, normalization, consistency, and decidability of conversion. The development is fully mechanized in Agda.

CCS Concepts: • **Software and its engineering** → *Abstraction, modeling and modularity*; Abstract data types; *Formal language definitions*; • **Theory of computation** → *Logic and verification*; **Type theory**.

Keywords: opaque definitions, dependent types, formalization, Agda

ACM Reference Format:
Nils Anders Danielsson and Eve Geng. 2025. A Formalization of Opaque Definitions for a Dependent Type Theory. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759538.3759653

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 easy to reuse code without having to repeat oneself: one can simply write the name α in place of the code at any 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
double n = n + n

quadruple : ℕ → ℕ -- Quadruples a natural
quadruple n = double (double n)
```

This 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 use the name in a later type signature:

```
ℕ>0 : Set -- Positive naturals
ℕ>0 = Σ[ n ∶ ℕ ] n > 0

pred : ℕ>0 → ℕ -- Predecessor
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 equipped with setoid equality relations:[1]

```
ℤ : Set -- Integers as differences of naturals
ℤ = ℕ × ℕ

_≡ℤ_ : ℤ → ℤ → Set -- Equality on integers
x ≡ℤ y = fst x + snd y ≡ fst y + snd x

_*ℤ_ : ℤ → ℤ → ℤ -- Integer multiplication
x *ℤ y = fst x * fst y + snd x * snd y
       , fst x * snd y + fst y * snd x

0ℤ : ℤ -- Integer zero
0ℤ = 0 , 0

ℤ≠0 : Set -- Nonzero integers
ℤ≠0 = Σ[ x ∶ ℤ ] ¬ x ≡ℤ 0ℤ

ℚ : Set -- Rationals as the field of fractions
ℚ = ℤ × ℤ≠0
```

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

TyDe '25, Singapore, Singapore

ACM ISBN 979-8-4007-2163-2/25/10
https://doi.org/10.1145/3759538.3759653

```
_≡ℚ_ : ℚ → ℚ → Set -- Equality on rationals
x ≡ℚ y = fst x *ℤ fst (snd y) ≡ℤ fst y *ℤ fst (snd x)
```

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

Informally, the semantics of definitions is fairly obvious: a definition name is equal to its corresponding definiens. This is straightforward for execution, but it may pose some problems when programs are constructed or type-checked. For example, suppose that we want to show that equality on the rationals is symmetric:

```
sym-ℚ : ∀ {x y} → x ≡ℚ y → y ≡ℚ x
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. 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))
```

This type is arguably not very readable, and we might have preferred to see something like the following, where some of the "implementation details" are hidden:

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

Unfolding of definitions is a basic ingredient in typical intensional type theories, but it can lead to problems:

1. As seen above, excessive unfolding can impair usability by obfuscating types shown to users.
2. Unnecessary unfolding can also bog down type checking, which typically involves some kind of normalization of types.

One way to mitigate such problems is to use **opaque definitions** [7]. 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:[2]

```
opaque
  ℤ : Set -- Integers as differences of naturals
  ℤ = ℕ × ℕ
```

[2]Support for opaque definitions was added to Agda by Amélia Liao (https://github.com/agda/agda/pull/6628).

With ℤ now opaque, it is no longer definitionally equal to ℕ × ℕ.[3] As a consequence, the definition of `_≡ℤ_` no longer type-checks, since the pair projections `fst` and `snd` do not apply to ℤ. To fix this, we can use the `unfolding` keyword to tell the type checker that when the body of the definition of `_≡ℤ_` is checked, the definition of ℤ *can* be unfolded, thus re-establishing the lost equality:

```
opaque
  unfolding ℤ
  _≡ℤ_ : ℤ → ℤ → Set -- Equality on integers
  x ≡ℤ y = fst x + snd y ≡ fst y + snd x
```

Such definitions make up a sort of "interface" for ℤ, allowing us to reason about it without having to look into the underlying implementation details. Note that the `unfolding` directive may only be used in other opaque definitions: if `_≡ℤ_` were not opaque in the above example, then details of the implementation of ℤ could "leak" when `_≡ℤ_` is unfolded. Furthermore, the directive only applies to the *body* of the definition and not to its *type*, which is part of the "interface" and therefore must make sense without any unfolding.

When defining the rationals, we can use the interface to avoid unfolding ℤ at all:

```
opaque
  ℤ≠0 : Set -- Nonzero integers
  ℤ≠0 = Σ[ x ∶ ℤ ] ¬ x ≡ℤ 0ℤ

opaque
  ℚ : Set -- Rationals as the field of fractions
  ℚ = ℤ × ℤ≠0

opaque
  unfolding ℚ ℤ≠0
  _≡ℚ_ : ℚ → ℚ → Set -- Equality on rationals
  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. It declares unfolding for `_≡ℚ_`, which in turn implies unfolding for ℚ and ℤ≠0 (and everything those definitions declare unfolding for, and so on):

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

As a result of the use of opaque definitions, the normalized goal type is now the type that we gave earlier:

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

Here, the type checker has unfolded `_≡ℚ_` as asked, but has gotten stuck on the opaque `_*ℤ_` and `_≡ℤ_`, thus yielding a goal type that is arguably more readable. The proof can be

[3]The definitional equality holds in the body of the definition of ℤ.

completed using symmetry of `_≡ℤ_`, which could be part of the interface for `ℤ`.

Implicit unfolding of transitive dependencies as 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 suggests that the design of a type theory with opaque definitions and unfolding declarations is not entirely trivial.

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 [2], 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, 3] is then employed to establish a number of metatheoretic properties: normalization, consistency, decidability of conversion, and so on.

In this work, we extend the type theory of graded-type-theory with opaque definitions, also based on Agda (adding top-level definitions in general along the way), then show that many of 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 for a language with dependent types. However, we note a few limitations of our work:

- For simplicity, and following the Agda design, we only allow top-level definitions; see §3.3 for more discussion.
- As mentioned above the language that is formalized in the graded-type-theory project is a *graded modal* type theory. In this text we ignore the grades (but grades are, to some extent, supported in the formalization).
- The type theory optionally supports *equality reflection*. We support equality reflection in the presence of definitions, but not *opaque* definitions, and will largely be ignoring this feature.

The remainder of this paper is structured as follows:

- In §2, we discuss some related work in the area.
- In §3, we recall the methodology of the formalization and go over the general idea of our extensions.
- In §4, we detail our extensions to the type theory for supporting top-level definitions without opacity.
- In §5, we detail further modifications for supporting *opaque* definitions.

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. We have chosen to be a little lax in the text in order to avoid including too many details; for instance, we omit certain arguments to the logical relations. Interested readers can find the full source code in this paper's software artifact [6]. Additionally, in the digital version of this paper, definitions and results are accompanied by (possibly less stable) links, displayed in blue, to the relevant parts of the Agda code. Because we build up our extensions incrementally throughout the paper, such links point to a snapshot 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.

The full development includes results not presented in this text, and the last two snapshots mentioned above include parts that do not type-check. However, the software artifact [6] contains further changes which make all of the code type-correct.

2 Related Work

Opaque definitions have been studied and implemented in various dependently-typed settings:

- Gratzer et al. [7] present a formalism for opaque definitions based on an elaboration to a language with extension types, along with an implementation as part of cooltt [16]. The surface language supports top-level unfolding directives, as in the examples above, but also expression-level unfolding directives. In the target language definitions are represented as specially-typed variables[4] in the typing context, which allows for local definitions. Gratzer et al. show normalization for the system, but the proof does not seem to be mechanized. Our proof is different from theirs: our proof does not involve the use of extension types.
- We discussed the Agda implementation of opaque definitions above. It is loosely based on the theory by Gratzer et al. [7], but dispenses with the extension types. Moreover, there are only top-level definitions, and only top-level unfolding directives. We follow suit in our work here.
- In Rocq, opaque definitions are present in the "core language" as defined by the reference manual [17], but not in the calculus of (inductive) constructions on which it is originally based [5, 11]. There is a single context for definitions and variables, but unlike

[4] 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.

the target language in the work of Gratzer et al. [7], definitions are distinct from variables. There has been some effort towards a formalization/mechanization of Rocq's metatheory, including its extensions to the calculus of inductive constructions; the work of Sozeau et al. [13] and MetaRocq [12] are two such examples. However, they operate on simplified versions of the core language that exclude opacity.

- In Lean [15], an opaque definition can be introduced with the `opaque` command, which instructs the kernel to check the definiens, but to then discard it. This irreversibly bars the definition from δ-reduction.
  Lean definitions also have a separate notion of "reducibility", which functions as a hint for the tactic engine. When the "irreducible" attribute is applied to a definition, tactics will refuse to unfold it; this can then be locally undone by removing the attribute with the `unseal` command. However, reducibility is discarded in elaboration to the core (kernel) language. Carneiro [4] formalizes some metatheory for Lean; the formalization seems to only address opacity, not reducibility.
- Automath, one of the earliest systems for mechanized mathematics, supports definitions [18]. The original system does not seem to have featured opacity, but a modern re-implementation includes support for it [19].

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 [10]. Some examples:

- In many languages, the notion of an "abstract data type" or "interface" offers a similar form of separation of concerns: since the abstract type 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 [8] present a "translucent sum type" to represent 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.[5]
- 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* [9].

[5] 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 $\bot$.

3 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. We also briefly describe the general idea of our extensions.

3.1 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 3.1. A **term** is either a variable, represented as a de Bruijn index, or a term constructor applied to subterms:

$$\text{Variable } x ::= x_0 \mid x_1 \mid \dots$$
$$\text{Term } A, B, t, u ::= x \mid \Pi\ A\ B \mid t\ u \mid \mathbb{N} \mid \mathsf{zero} \mid \mathsf{suc}\ t \mid \dots$$

Here, x_i refers to the variable at index i.

Note that terms are well-scoped in the actual formalization.[6]

We also use typing contexts, weakenings and substitutions. We do not include all details of their definitions here:

Definition 3.2. 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.

Definition 3.3. A **weakening** ρ lifts a term t to a term $t[\rho]$ by incrementing free de Bruijn indices in t: weakenings are used to embed terms into larger contexts.

Definition 3.4. A **substitution** σ transforms a term t into a term $t[\sigma]$ by replacing each free variable in t with a specified term (which may be weakened to account for binders).

Typing contexts, weakenings and substitutions are also well-scoped in the formalization proper, but again we elide this.

We use the following basic typing judgements and reduction relations:

Definition 3.5 (Typing judgements)**.**

$\vdash \Gamma$ — Γ is a well-formed typing context.
$\Gamma \vdash A$ — A is a well-formed type in Γ.

[6] 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.

$\Gamma \vdash t : A$ — 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.
$\Gamma \vdash A \Rightarrow B$ — A reduces to B as a type in Γ.
$\Gamma \vdash t \Rightarrow u : A$ — t reduces 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. The reduction relations are of the small-step kind, and weak head reduction is used.

With these rules, we can immediately derive a handful of "direct" results (we use $\Gamma \vdash \mathcal{J}$ as notation for any of $\Gamma \vdash A$, $\Gamma \vdash t : A$, $\Gamma \vdash A \equiv B$ and $\Gamma \vdash t \equiv u : A$):

Lemma 3.6 (Weakening)**.** If $\Gamma \vdash \mathcal{J}$, then for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, we have $\Delta \vdash \mathcal{J}[\rho]$, and similarly for reduction.

Lemma 3.7 (Substitution)**.** If $\Gamma \vdash \mathcal{J}$, then for any well-formed substitution $\Delta \vdash \sigma : \Gamma$, we have $\Delta \vdash \mathcal{J}[\sigma]$, and similarly for reduction.

Lemma 3.8 (Well-formedness)**.** Most typing judgements require that their constituents are well-formed:

- 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.

Because the reduction relation for terms is typed, subject reduction reduces to well-formedness of reduction:

Theorem 3.9 (Subject Reduction)**.** If $\Gamma \vdash t \Rightarrow u : A$ and $\Gamma \vdash t : A$, then $\Gamma \vdash u : A$.

3.2 The Logical Relation

For more substantial results about reduction, we resort to a Kripke logical relation argument in the style of Abel et al. [3]. 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, as illustrated in Fig. 3.1.

We start by defining 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 3.10. A term is **neutral** if it has a variable in its head position.

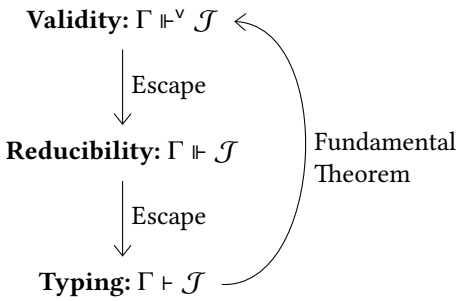

Figure 3.1. The validity-reducibility-typing "ladder".

Definition 3.11. A term is in **weak head normal form** (WHNF) if it is either neutral or a constructor application.

Definition 3.12. The **logical relation for reducibility** consists of the following relations:[7]

$\Gamma \Vdash A$ — 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.

Reducibility generally entails two things:

1. Everything in sight reduces (in zero or more steps) to a WHNF.
2. The WHNFs "behave correctly" under 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. 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 term),
- for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, and any $\Delta \Vdash a : A[\rho]$, we have $\Delta \Vdash f[\rho]\ a : B[\rho\uparrow][a/x_0]$, and
- for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, and any $\Delta \Vdash a : A[\rho]$ and $\Delta \Vdash b : A[\rho]$ such that $\Delta \Vdash a \equiv b : A[\rho]$, we have $\Delta \Vdash f[\rho]\ a \equiv f[\rho]\ b : B[\rho\uparrow][a/x_0]$.

Here $\uparrow$ is a "lifting" operation that takes a weakening $\rho : \Delta \supseteq \Gamma$ to a new lifted one $\rho\uparrow : \Delta \cdot A[\rho] \supseteq \Gamma \cdot A$ (if $\Delta \vdash A[\rho]$).

Note the generalization over weakenings in the above conditions. We will use reducibility weakening lemmas in later proofs, but because reducibility occurs in negative positions in its own definition (e.g. as premises to the two latter conditions), it is difficult to prove these lemmas directly by induction. Instead, by building weakening directly into

[7] In the formalization proper, the three latter cases are parametrized 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.

the definition of reducibility, we are essentially shifting the proof burden for weakening elsewhere—in particular, to the fundamental theorem (3.17). For now, we have:

Lemma 3.13 (Weakening)**.** If $\Gamma \Vdash \mathcal{J}$, then for any well-formed weakening $\rho : \Delta \supseteq \Gamma$, we have $\Delta \Vdash \mathcal{J}[\rho]$.

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

The proof of the forwards direction, on the other hand, is more involved. The general idea is to proceed by induction on the typing derivations $\Gamma \vdash \mathcal{J}$. However, the motive is strengthened from reducibility to what is known as *validity*:

Definition 3.15 (Validity judgements)**.**
$\Vdash^{\mathsf{v}} \Gamma$ — Every type in Γ is valid.
$\Gamma \Vdash^{\mathsf{v}} A$ — 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.
$\Delta \Vdash^{\mathsf{v}} \sigma : \Gamma$ — σ is a valid substitution from Γ to Δ.
$\Delta \Vdash^{\mathsf{v}} \sigma_1 \equiv \sigma_2 : \Gamma$ — 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 simplify weakening lemmas, baking substitution into validity simplifies substitution lemmas. As an example, consider validity for terms: $\Gamma \Vdash^{\mathsf{v}} t : A$ holds when $\Gamma \Vdash^{\mathsf{v}} A$ and, for any validly equal substitutions $\Delta \Vdash^{\mathsf{v}} \sigma_1 \equiv \sigma_2 : \Gamma$, we have $\Delta \Vdash t[\sigma_1] \equiv t[\sigma_2] : A[\sigma_1]$. Plain reducibility falls out of this definition as the special case for the trivial substitution $\sigma_1 = \sigma_2 = \mathsf{id}$:

Lemma 3.16 (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 for the logical relation*:

Theorem 3.17 (Fundamental Theorem)**.** If $\Gamma \vdash \mathcal{J}$, then $\Gamma \Vdash^{\mathsf{v}} \mathcal{J}$. Similarly, if $\vdash \Gamma$, then $\Vdash^{\mathsf{v}} \Gamma$.

The principal challenge of our work will be to repair the proof of the fundamental theorem after making our extensions. The original proof uses induction on the typing derivation $\Gamma \vdash \mathcal{J}$ (or $\vdash \Gamma$), and mainly amounts to showing that the premises of each typing rule, given suitable inductive hypotheses, imply the corresponding validity judgement. As an example, consider the successor typing rule for natural numbers:

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

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 $\Gamma \Vdash^{\mathsf{v}} \mathsf{suc}\ t : \mathbb{N}$. Later on, when we are 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 have obtained a way to reduce generalizations over all terms to generalizations over only normal forms. A simple example of this is the normalization theorem:

Theorem 3.18 (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 are done. Various other results also follow from the fundamental theorem, including the following ones:

Theorem 3.19 (Canonicity for $\mathbb{N}$)**.** If $\varepsilon \vdash t : \mathbb{N}$, where ε is the empty context, then t is judgementally equal to a canonical form of $\mathbb{N}$, i.e. suc applied zero or more times to zero.

Theorem 3.20 (Consistency)**.** In the empty context ε, the empty type $\bot$ is uninhabited.

The formalized normalization proof is parametrized in a certain way, and by instantiating it a second time, one can obtain decidability of conversion [3]:

Theorem 3.21 (Decidable Conversion)**.** Judgemental equality is decidable:

- 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 $\Gamma \vdash t \equiv u : A$.

The formalization that we build on [3] also includes results related to decidability of type checking. We do not list them here, but we build on them and present similar results in §5.3.

3.3 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. We will also

augment the logical relation accordingly, so that $\Gamma \Vdash \mathcal{J}$ becomes $\nabla \gg \Gamma \Vdash \mathcal{J}$ and $\Gamma \Vdash^{\vee} \mathcal{J}$ becomes $\nabla \gg \Gamma \Vdash^{\vee} \mathcal{J}$.

For a typical variable, one only needs to know its type. For a definition, on the other hand, one also needs to know its definiens, its opacity, as well as any associated unfolding directives in order to know how the definition should unfold. The responsibility of the definition context, then, is to store all of this information for later use by the typing rules. This design might be thought of as a simplified version of the way opaque definitions are implemented in Agda [14], where things are more complicated due to, for instance, the presence of mutually recursive definitions, which we do not provide direct support for.

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 is 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 depends on the definition context.

An alternative approach might be to use a single, heterogeneous context in which later entries can depend on earlier entries, regardless of whether they are variables or definitions. This is the approach taken by the Rocq proof assistant [17]. Because Agda itself only supports top-level opaque definitions, as well as for the sake of simplicity, we have opted to stick with a separate definition context in the present work.

4 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.

4.1 The Formalism

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

Definition 4.1. A **term** can also 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? With de Bruijn indices, if we were to append an entry to a definition context, then we would have to shift indices in order to ensure that they still point to the same thing. With de Bruijn levels, there is no need to do this.

Because definition names contain no variables, they are invariant under substitution. This will be useful for the proof of Lemma 5.16 later.

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

Definition 4.2. A **definition context** is a list of type annotations and definientia for definitions in the context:

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

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

Definition 4.3. The **maps-to 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 ∇.

These relations will be used in the typing rules for definitions, which we now move to.

We first state 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 4.4. **Well-formedness for definition contexts** is defined inductively by the following rules:

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

One might have expected to see $\gg \nabla$ as a premise of the EXTEND rule, but it turns out that this follows from the given premise by a well-formedness lemma (4.9).

Next, we augment the well-formedness of typing contexts $\vdash \Gamma$ to $\nabla \gg\!\vdash \Gamma$:

Definition 4.5. **Well-formedness for typing contexts** is defined inductively by the following rules:

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

None of the existing typing rules depend on ∇, and so they remain largely untouched (aside from slapping a ∇ onto the judgements). However, our new typing rules for definitions look into ∇ using the maps-to relations:

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

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

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

These rules state that a definition takes its type from its type annotation and is equal to (and reduces to) its definiens. We refer to the reduction step as "δ-reduction", following work on Automath [18].

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 free 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 4.6. 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 $\xi \mathbin{»} \nabla' \supseteq \nabla$ from ∇ to ∇' when each definition in the sequence is well-typed.

With these rules, we can derive some simple results about definitions and definition contexts:

Lemma 4.7 (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 4.8 (Well-formedness)**.** If $\mathbin{»} \nabla$ and $\alpha \mapsto t : A \in \nabla$, then $\nabla \mathbin{»} \varepsilon \vdash t : A$.

Note that the proof of the above well-formedness lemma (4.8) uses weakening for definitions (4.7): it amounts to extracting the proof from $\mathbin{»} \nabla$ that $\nabla' \mathbin{»} \varepsilon \vdash t : A$ for some sub-context ∇' of ∇, then weakening it up to ∇. This is simple enough for now, but our proof of an analogous lemma for validity (4.15) is more complicated.

The previous "direct" results from §3.1 still hold, albeit with minor changes to the proofs to account for the new typing rules. Our new well-formedness lemma also includes a new case for definition contexts:

Lemma 4.9 (Well-formedness)**.** Most typing judgements require that their constituents are well-formed:

- If $\nabla \gg\!\vdash \Gamma$, then $\mathbin{»} \nabla$.
- If $\nabla \mathbin{»} \Gamma \vdash A$, then $\nabla \gg\!\vdash \Gamma$.
- If $\nabla \mathbin{»} \Gamma \vdash t : A$, then $\nabla \mathbin{»} \Gamma \vdash A$.
- If $\nabla \mathbin{»} \Gamma \vdash A \equiv B$, then $\nabla \mathbin{»} \Gamma \vdash A$ and $\nabla \mathbin{»} \Gamma \vdash B$.
- If $\nabla \mathbin{»} \Gamma \vdash t \equiv u : A$, then $\nabla \mathbin{»} \Gamma \vdash t : A$ and $\nabla \mathbin{»} \Gamma \vdash u : A$.
- Similarly for reduction.

4.2 Updating the Logical Relation

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

Definition 4.10. The **logical relation for reducibility** consists of the following 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 4.11. The **validity judgements** are the following relations:

- $\nabla \gg\!\Vdash^{\mathrm{v}} \Gamma$ — Every type in Γ is valid in ∇.
- $\nabla \mathbin{»} \Gamma \Vdash^{\mathrm{v}} A$ — A is a valid type in ∇ and Γ.
- $\nabla \mathbin{»} \Gamma \Vdash^{\mathrm{v}} t : A$ — t is a valid term of type A in ∇ and Γ.
- $\nabla \mathbin{»} \Gamma \Vdash^{\mathrm{v}} A \equiv B$ — A and B are validly equal as types in ∇ and Γ.
- $\nabla \mathbin{»} \Gamma \Vdash^{\mathrm{v}} t \equiv u : A$ — t and u are validly equal as terms of type A in ∇ and Γ.
- $\nabla \mathbin{»} \Delta \Vdash^{\mathrm{v}} \sigma : \Gamma$ — σ is a valid substitution from Γ to Δ in ∇.
- $\nabla \mathbin{»} \Delta \Vdash^{\mathrm{v}} \sigma_1 \equiv \sigma_2 : \Gamma$ — 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 (with the definition context ∇ tacked on so that it can be passed into the underlying typing judgements), but with one key difference that we will discuss in connection with Lemma 4.16 below.

For now, note that no validity variant of definition context well-formedness is given above. As discussed in §3.2 the logical relation is focused on WHNFs, and definitions can always reduce: they are not WHNFs. It turns out that we can get away without validity for definitions as part of Definition 4.11, definition context well-formedness is sufficient for our purposes: $\nabla \gg\!\Vdash^{\mathrm{v}} \epsilon = \mathbin{»} \nabla$.

Nevertheless, validity for definitions will be used to prove the cases of the fundamental theorem related to definitions. Thus, we define it now, *after* the other validity judgements:[8]

Definition 4.12. **Validity for definition contexts** is defined by the following recursive equations:

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

Now, we can state the updated fundamental theorem:

[8] That is to say, $\mathbin{»}^{\mathrm{v}}\ \nabla$ is not defined mutually-recursively with any of the other validity judgements.

Theorem 4.13 (Fundamental Theorem)**.** All of the following hold:

- If $\gg \nabla$, then $\gg^{\vee} \nabla$.
- If $\nabla \gg\vdash \Gamma$, then $\nabla \gg\Vdash^{\vee} \Gamma$.
- If $\nabla \gg \Gamma \vdash \mathcal{J}$, then $\nabla \gg \Gamma \Vdash^{\vee} \mathcal{J}$.

Proceeding by mutual induction,[9] the first two statements can be shown using the third as a mutual-inductive hypothesis: 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 typing rules.

We will 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 hypotheses give us $\gg^{\vee} \nabla$ and $\nabla \gg\Vdash^{\vee} \Gamma$. Then, it suffices to show the following:

Lemma 4.14 (Validity of δ-reduction)**.** Given that $\gg^{\vee} \nabla$ and $\nabla \gg\Vdash^{\vee} \Gamma$, if $\alpha \mapsto t : A \in \nabla$, then $\nabla \gg\vdash \Gamma \Vdash^{\vee} \alpha \equiv t : A$.

To make use of the assumption $\alpha \mapsto t : A \in \nabla$, we will use a well-formedness lemma for valid definition contexts analogous to the one given earlier for well-formed ones (4.8):

Lemma 4.15 (Well-formedness)**.** If $\gg^{\vee} \nabla$ and $\alpha \mapsto t : A \in \nabla$, then $\nabla \gg \varepsilon \Vdash^{\vee} t : A$.

As with Lemma 4.8 before, we prove this using a weakening lemma for definitions—this time, for validity:

Lemma 4.16 (Weakening)**.** If $\nabla \gg \Gamma \Vdash^{\vee} \mathcal{J}$, then for any well-formed extension $\xi \gg \nabla' \supseteq \nabla$, we have $\nabla' \gg \Gamma \Vdash^{\vee} \mathcal{J}$.

In proving this, we run into the same problem we previously encountered with weakening of the typing context (3.13)—namely, that reducibility occurs negatively in its own definition. Fortunately, we can use the same trick again to work around the problem: we just bake weakening for definitions into validity. To illustrate, let us see how validity for terms has changed to accommodate these weakenings: $\nabla \gg \Gamma \Vdash^{\vee} t : A$ holds when

- $\nabla \gg \Gamma \Vdash^{\vee} A$ and
- for any well-formed extension $\xi \gg \nabla' \supseteq \nabla$ and any validly equal substitutions $\nabla' \gg \Delta \Vdash^{\vee} \sigma_1 \equiv \sigma_2 : \Gamma$, we have $\nabla' \gg \Delta \Vdash t[\sigma_1] \equiv t[\sigma_2] : A[\sigma_1]$.

[9] The first statement is proved using induction on the length of the definition context. "Recursive calls" in the proofs of the other two statements always use an unchanged definition context.

Returning to the proof of Lemma 4.14, a key insight is that because reducibility speaks principally about a term's WHNF, any term t that reduces to a reducible term u—and therefore has the same WHNF as u—is reducibly equal to u. By then generalizing over weakenings (for definitions) and substitutions, we can extend this result to validity:

Lemma 4.17 (Expansion)**.** Given that

- $\nabla \gg \Gamma \Vdash^{\vee} u : A$ and
- for any well-formed extension $\xi \gg \nabla' \supseteq \nabla$ and any valid substitution $\nabla' \gg \Delta \Vdash^{\vee} \sigma : \Gamma$, we have $\nabla' \gg \Delta \vdash t[\sigma] \Rightarrow u[\sigma] : A[\sigma]$,

we can conclude that $\nabla \gg \Gamma \Vdash^{\vee} t \equiv u : A$.

We can now complete the proof of Lemma 4.14: By well-formedness, we know that $\nabla \gg \varepsilon \Vdash^{\vee} t : A$, which gives us $\nabla \gg \Gamma \Vdash^{\vee} t : A$ by a weakening lemma for validity. Then, by expansion for δ-reduction, we can conclude that $\nabla \gg \Gamma \Vdash^{\vee} \alpha \equiv t : A$.

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

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

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

Lemma 4.18 (Validity of Definitions)**.** Given that $\gg^{\vee} \nabla$ and $\nabla \gg\Vdash^{\vee} \Gamma$, if $\alpha \mapsto\!: A \in \nabla$, then $\nabla \gg\vdash \Gamma \Vdash^{\vee} \alpha : A$.

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 $\alpha \mapsto t : A \in \nabla$. We thus 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.

5 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.

5.1 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 be associated with unfolding directives, we also include information about those:

Definition 5.1. An **opacity** for entry number $1 + n$ in a definition context is either tra, indicating that the entry is transparent; or $\mathsf{opa}(\varphi)$ for an n-bitvector φ, indicating that the entry is opaque, with unfolding directives for

those previous definitions that correspond to the one bits in φ:

$$\text{Opacity } \omega ::= \mathsf{tra} \mid \mathsf{opa}(\varphi)$$
$$\text{Unfolding Vector } \varphi ::= \varepsilon \mid \varphi 0 \mid \varphi 1$$

Definition 5.2. A **definition context** is a list of definition bindings, where each entry consists of a type annotation, a definiens, and an opacity:

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

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

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

Due to the presence of both transparent and opaque definitions we use three maps-to relations:

Definition 5.3. The **maps-to 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 ∇.

With a bit of easy induction, we can move from the last of these relations to one of the other two:

Lemma 5.4 (Dichotomy)**.** If $\alpha \mapsto\!\!: A \in \nabla$, then either $\alpha \mapsto \emptyset : A \in \nabla$ or there is a t for which $\alpha \mapsto t : A \in \nabla$.

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

Definition 5.5. The **glassify** operation is given by the following recursive equations:

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

Lemma 5.6 (Glassification)**.** If $\alpha \mapsto\!\!: A \in \nabla$, then there is some t for which $\alpha \mapsto t : A \in \mathsf{glassify}(\nabla)$.

As we will see in §5.3, certain results are stated only for glass (i.e. fully transparent) contexts, because they do not necessarily hold for general contexts.

When an opaque definition $\alpha \triangleq_{\mathsf{opa}(\varphi)} t : A$ is checked for well-formedness, the definitions marked in φ should be treated as transparent. This, along with the transitive unfolding discussed earlier, is encoded by the transparentization relation $\varphi \mathbin{»} \nabla' \mathbin{\backsim} \nabla$, where ∇ is the "input" context and ∇' is the "output":

Definition 5.7. The **transparentization relation** is defined inductively by the following rules:

$$\frac{}{\varepsilon \mathbin{»} \epsilon \mathbin{\backsim} \epsilon}\ \textsc{empty}$$

$$\frac{\varphi \mathbin{»} \nabla' \mathbin{\backsim} \nabla}{\varphi 0 \mathbin{»} \nabla' \cdot_{\omega} (t : A) \mathbin{\backsim} \nabla \cdot_{\omega} (t : A)}\ \textsc{no}$$

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

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

Note the $\sqcup$ in the YES-OPA rule, which stands for bitwise disjunction. The idea is to enforce transitive unfolding by merging the definition's unfolding vector into the running vector. Note also that the relation is deterministic: $\varphi \mathbin{»} \nabla' \mathbin{\backsim} \nabla$ and $\varphi \mathbin{»} \nabla'' \mathbin{\backsim} \nabla$ imply that $\nabla' = \nabla''$.

We can now update the well-formedness judgement:

Definition 5.8. **Well-formedness for definition contexts** is defined inductively in the following way:

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

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

Note that even in the opaque case, the type of the definition must check in the original context: the type signature of an "interface" definition must not leak the "implementation details" of any opaque definitions. For example, imagine if we had tried to extend the interface for $\mathbb{Z}$ in §1 with a definition 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:

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

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

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

However, note that the $\mapsto$ variant of the maps-to relations only holds for transparent definitions. Opaque definitions should not unfold, so the δ-RED rules use this variant. The DEFN rule uses the $\mapsto\!:$ variant, which is opacity-agnostic: uses of definitions are typed in the same way regardless of opacity.

There are two useful classes of context-modification lemmas which we can now prove for the typing judgements:

Lemma 5.9 (Weakening)**.** 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 5.10 (Glassification)**.** Typing judgements are preserved by glassification. That is, if $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, then $\mathsf{glassify}(\nabla) \mathbin{»} \Gamma \vdash \mathcal{J}$.

The "direct" results from §3.1 and §4 still hold (with the addition of "$\nabla \mathbin{»}$" wherever appropriate), including subject reduction:

Theorem 5.11 (Subject Reduction)**.** If $\nabla \mathbin{»} \Gamma \vdash t \Rightarrow u : A$ and $\nabla \mathbin{»} \Gamma \vdash t : A$, then $\nabla \mathbin{»} \Gamma \vdash u : A$.

We can also prove a theorem analogous to subject reduction for transparentization:

Theorem 5.12 (Well-formedness Preservation)**.** Given $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$, if $\mathbin{»} \nabla$, then $\mathbin{»} \nabla'$, and if $\nabla \mathbin{»} \Gamma \vdash \mathcal{J}$, then $\nabla' \mathbin{»} \Gamma \vdash \mathcal{J}$.

Gratzer et al. [7] observe that subject reduction is lost with non-transitive unfolding; we see a similar result when we employ a variant of $\sqcup$ that rejects transitive unfolding:

Counterexample 5.13 (Unfolding)**.** With the alternate definition $\varphi \sqcup \varphi' = \varphi$, there exist definition contexts ∇ and ∇' and an unfolding $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$ such that $\mathbin{»} \nabla$, but not $\mathbin{»} \nabla'$.

Our counterexample uses the context

$$\nabla = \epsilon \cdot_{\mathsf{opa}()} (\mathbb{N} : \mathcal{U}) \cdot_{\mathsf{opa}(1)} (0 : \alpha_0)$$

and an unfolding vector 01 that makes $\alpha_1 \triangleq 0 : \alpha_0$ transparent but not $\alpha_0 \triangleq \mathbb{N} : \mathcal{U}$. The resulting transparentized context is ill-formed: we cannot show that $0 : \alpha_0$, because we cannot deduce that $\alpha_0 \equiv \mathbb{N}$.

Is the counterexample above a problem in practice? Most of our results actually hold with either of the two definitions of $\sqcup$ above. However, type-checking might be less efficient with the second definition. In the presence of Theorem 5.12 a type-checker might check if the context $\nabla \cdot_{\mathsf{opa}(\varphi)} (t : A)$ is well-formed in the following way, aborting if any answer is "no" [6]:

1. Does $\mathbin{»} \nabla$ hold?
2. Does $\nabla \mathbin{»} \varepsilon \vdash A$ hold for the well-formed context ∇?
3. Let ∇' be the (unique) context such that $\varphi \mathbin{»} \nabla' \leftsquigarrow \nabla$. Does $\nabla' \mathbin{»} \varepsilon \vdash t : A$ hold for the well-formed context ∇' and the well-formed type A?

Counterexample 5.13 shows us that, if the alternative definition of $\sqcup$ is used, then it may be the case that the answer to the first two questions is yes, but ∇' is not well-formed. This could be addressed by adding the step "Does $\mathbin{»} \nabla'$ hold?", but that seems terrible from a performance perspective. Perhaps there is a better fix, but the problem can be avoided entirely through the use of transitive unfolding.

5.2 Updating the Logical Relation, Take Two

A term is, generally speaking, "neutral" if it cannot reduce but is not in canonical form. Before we added opacity, this was only the case for terms blocked on variables, but now we use the following definition:

Definition 5.14. A term is **neutral** if it has either a variable or an opaque definition in its head position.

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

Definition 5.15. A term is in **weak head normal form** (WHNF) if it is either neutral or a constructor application.

The logical relations, both for reducibility and validity, do 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 key exception is the case for well-typed definitions, which no longer reduces to δ-reduction.

Recall that to discharge the case for well-typed definitions, it suffices to prove the following validity lemma (previously Lemma 4.18):

Lemma 5.16 (Validity of Definitions)**.** Given that $\mathbin{»}^{\vee} \nabla$ and $\nabla \mathbin{»}\!\Vdash^{\vee} \Gamma$, if $\alpha \mapsto\!: A \in \nabla$, then $\nabla \mathbin{»}\!\Vdash \Gamma \Vdash^{\vee} \alpha : A$.

In §4.2, we noted that α must always unfold to some t, which allowed us to reuse the argument used for δ-reduction (4.14). 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 (5.4), we either have that $\alpha \mapsto \emptyset : A \in \nabla$ or that there is some t for which $\alpha \mapsto t : A \in \nabla$. The latter case reduces to validity of δ-reduction, and so it now suffices to show validity for only *opaque* definitions.

Recall that definitions are invariant under substitution. Using this, along with some weakening lemmas for definitions, we can conclude that it suffices to show *reducibility* for opaque definitions. To this end, we use the following lemma:

Lemma 5.17 (Reducibility of Neutrals)**.** If t is a neutral term of a reducible type $\nabla \mathbin{»} \Gamma \Vdash A$, then $\nabla \mathbin{»} \Gamma \Vdash t : A$.

Since the name of an opaque definition is neutral, it remains only to show that A is reducible, which we can do with a variant of well-formedness (previously Lemma 4.15):

Lemma 5.18 (Well-formedness)**.** If $\mathbin{»}^{\mathsf{v}} \nabla$ and $\alpha \mapsto\!: A \in \nabla$, then $\nabla \mathbin{»} \varepsilon \Vdash^{\mathsf{v}} A$.

5.3 Consequences of the Fundamental Theorem

The theorems listed in §3.2 still hold, with the caveat that canonicity for $\mathbb{N}$ is stated for a *glass* context to avoid situations where a term is stuck on an opaque definition:

Theorem 5.19 (Canonicity for $\mathbb{N}$)**.** If $\nabla \mathbin{»} \varepsilon \vdash t : \mathbb{N}$ holds, then t is judgementally equal to a canonical form of $\mathbb{N}$ in $\mathsf{glassify}(\nabla)$.

We get the following theorem from canonicity for the identity type:

Theorem 5.20 (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 type $\mathsf{Id}\ A\ t\ u$ must reduce to rfl, which allows us to syntactically unify t and u. However, if ∇ contains opaque entries, then the theorem might not hold. Consider the following Agda code:

```
opaque
  x : ℕ
  x = 0

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

This is well-typed because the unfolding directive allows x to reduce to 0, and so rfl is a valid constructor for $\mathsf{Id}\ \mathbb{N}\ \mathsf{x}\ 0 \equiv \mathsf{Id}\ \mathbb{N}\ 0\ 0$. However, if x is not unfolded, then x is neutral, and not judgementally equal to the canonical form 0:

Counterexample 5.21 (Non-glass Pseudo-reflection)**.** There exists a definition context ∇, type A, and terms t, u, and v such that $\nabla \mathbin{»} \varepsilon \vdash v : \mathsf{Id}\ A\ t\ u$ holds, but not $\nabla \mathbin{»} \varepsilon \vdash t \equiv u : A$.

A few other potentially interesting results:

Theorem 5.22 (Consistency)**.** In the empty context ε, the empty type $\bot$ is uninhabited.

Theorem 5.23 (Definition Consistency)**.** No definition in a well-formed definition context has the empty type $\bot$.

Theorem 5.24 (Type Normalization)**.** Any well-formed type $\nabla \mathbin{»} \Gamma \vdash A$ reduces to some WHNF.

Theorem 5.25 (Normalization)**.** Any well-typed term $\nabla \mathbin{»} \Gamma \vdash t : A$ reduces to some WHNF.

Theorem 5.26 (Decidable Conversion)**.** Judgemental equality is decidable:

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

We can also consider the question of whether type checking is decidable. The syntax is not fully annotated: lambda abstractions are, for instance, not annotated with types. For that reason we expect that type checking is not decidable, even though we have not proved this formally. However, if we restrict ourselves to a fragment of the language that excludes certain things (including β-redexes), then checking is decidable:

Theorem 5.27 (Decidable Type Checking)**.** Typing is decidable for a certain "checkable" fragment of the language:

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

6 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 and shown that the theory satisfies a number of metatheoretic properties.

We based the design on that of Agda, and do not support local definitions or expression-level unfolding directives: an obvious opportunity for further work is to try to include support for one or more of those features.

Acknowledgements

We would like to thank Amélia Liao, who added support for opaque definitions to Agda. We also thank Patrik Jansson for feedback on the second author's MSc thesis, which this text is based on.

Nils Anders Danielsson acknowledges financial support from Vetenskapsrådet (2023-04538).

References

[1] Andreas Abel, Nils Anders Danielsson, and Oskar Eriksson. 2023. A Graded Modal Dependent Type Theory with a Universe and Erasure, Formalized. *Proceedings of the ACM on Programming Languages* 7, ICFP, Article 220 (2023), 35 pages. doi:10.1145/3607862

[2] Andreas Abel, Nils Anders Danielsson, Oskar Eriksson, Naïm Favier, Gaëtan Gilbert, Ondřej Kubánek, Wojciech Nawrocki, Joakim Öhman, and Andrea Vezzosi. 2025. *An Agda Formalization of a Graded Modal Type Theory with a Universe Hierarchy and Erasure.* https://github.com/graded-type-theory/graded-type-theory

[3] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 23 (2017), 29 pages. doi:10.1145/3158111

[4] Mario Carneiro. 2024. *Lean4Lean: Towards a Verified Typechecker for Lean, in Lean.* arXiv:2403.14064v2 [cs.PL] doi:10.48550/arXiv.2403.14064

[5] Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2–3 (1988), 95–120. doi:10.1016/0890-5401(88)90005-3

[6] Nils Anders Danielsson and Eve Geng. 2025. *An Agda Formalization of a Graded Modal Type Theory with a Universe Hierarchy, Erasure and Opaque Definitions.* doi:10.5281/zenodo.16906631

[7] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2022. *Controlling unfolding in type theory.* arXiv:2210.05420v1 [cs.LO] doi:10.48550/arXiv.2210.05420

[8] Robert Harper and Mark Lillibridge. 1994. A Type-Theoretic Approach to Higher-Order Modules with Sharing. In *Conference Record of POPL '94: 21ST ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 123–137. doi:10.1145/174675.176927

[9] José Meseguer and Joseph A. Goguen. 1986. Initiality, induction, and computability. In *Algebraic methods in semantics.* Cambridge University Press, 459–541.

[10] D.L. Parnas. 1972. Information Distribution Aspects of Design Methodology. In *Information Processing 71.* North-Holland Publishing Company, 339–344.

[11] Christine Paulin-Mohring. 1993. Inductive Definitions in the system Coq; Rules and Properties. In *Typed Lambda Calculi and Applications, International Conference on Typed Lamda Calculi and Applications, TLCA '93.* 328–345. doi:10.1007/BFb0037116

[12] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* 64 (2020), 947–999. doi:10.1007/s10817-019-09540-0

[13] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2025. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM* 72, 1, Article 8 (2025), 74 pages. doi:10.1145/3706056

[14] The Agda Team. 2025. Agda User Manual, Release 2.8.0. https://agda.readthedocs.io/_/downloads/en/v2.8.0/pdf/

[15] The Lean Developers. 2025. The Lean Language Reference. https://lean-lang.org/doc/reference/4.21.0-rc3/

[16] The RedPRL Development Team. 2023. *cooltt.* https://github.com/RedPRL/cooltt

[17] The Rocq Development Team. 2025. The Rocq Prover Reference Manual, Release 9.0.0. https://github.com/coq/coq/releases/download/V9.0.0/rocq-9.0.0-reference-manual.pdf

[18] D.T. van Daalen. 1980. *The language theory of Automath.* Phd Thesis 2. Technische Hogeschool Eindhoven. doi:10.6100/IR85774

[19] Freek Wiedijk. 2002. A New Implementation of Automath. *Journal of Automated Reasoning* 29 (2002), 365–387. doi:10.1023/A:1021983302516

Received 2025-06-23; accepted 2025-07-23