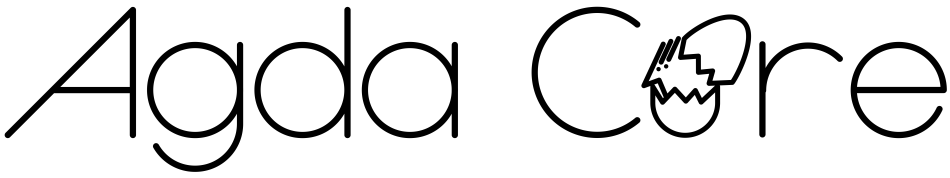
Correct-by-Construction eta-conversion for Agda Core

Master's Thesis



Ate-Jan de Vries

Correct-by-Construction eta-conversion for Agda Core

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ate-Jan de Vries

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, The Netherlands
www.ewi.tudelft.nl

Correct-by-Construction eta-conversion for Agda Core

Author: Ate-Jan de Vries
Student id: 4966422

Abstract

Software correctness is a hard problem. Dependently-typed programming languages like Agda help us to provide guarantees about software while writing it in a Correct-by-Construction (CbC) manner. However, the implementation of a dependently-typed programming language may contain bugs itself: Agda has bugs in its own implementation on a regular basis. Dependently typed-typed programming languages should therefore verify themselves in a Correct-by-Construction manner using their own components: *self-verification*. Agda Core aims to do this for Agda: it is a core language for Agda and provides a Correct-by-Construction type checker for itself derived from a trusted type theory, written in Agda.

Before Agda Core can be used as a true self-verifier for Agda and be integrated into Agda's main compiler and type checker, more features from Agda need to be supported by it. In this thesis, we focus on Agda's η-conversion for function types and record types. We formalize η-conversion for function types and record types with at least one field using *untyped conversion*: we also add support for records to Agda Core along the way. We also show progress towards a formalization of Agda's η-conversion using *typed conversion*, which allows for formalizing the often tricky-considered η-conversion for Agda's unit type. For all of these formalizations, we show that they can be added to Agda Core with reasonable effort. Overall, this work therefore provides an important step towards the ultimate goal of a self-verified type checker for Agda with support for all of Agda's features, which decreases the future potential for bugs in Agda's implementation with relation to η-conversion, and teaches us how to self-verify η-conversion for a dependently-typed language.

Thesis Committee:

Chair:	Dr. J.G.H Cockx, Faculty EEMCS, TU Delft
Committee Member:	Dr. Z. Erkin, Faculty EEMCS, TU Delft

Preface

Eight years of study at TU Delft are coming to an end. This thesis document is the result of the research that I conducted in the final year as a Master Computer Science student. The subject of the thesis is a reflection of why I originally became interested in Computer Science: the question of why software can fail in the way that it often does and also merely my interest in solving a complex puzzle with precise constrains and scope in a very specific niche.

I would like to thank my advisor: Jesper Cockx, who always encouraged me to try out the ideas that I was thinking of and for finding the time to give a thought on any of my questions, as well as for reviewing my work. I would also like to thank Nathaniel Burke, who was always happy to hear about what I was doing for my thesis project and who gave me an impression of just how large the world of type theory and Agda really is. I would also like to thank Arthur Jacques for exchanging ideas on Agda Core and for the opportunity of giving guidance on each other's thesis projects, which both targeted Agda Core.

I would furthermore like to thank Zeki Erkin for supporting my pursuit of this specific thesis project with Jesper Cockx and for acting as external committee member.

Finally, and perhaps most importantly, I would like to thank my family for always supporting me throughout my studies.

Ate-Jan de Vries
Delft, The Netherlands
June 25, 2026

Contents

Chapter 1

Introduction

Ensuring the correctness of software systems is a major challenge in computer science and it is hard. The economic impact of software bugs is well-known: inadequate software testing causes the U.S. economy to lose billions of dollars per year [1, 2]. Traditional methods such as manual testing and model checking [3] can increase confidence in software correctness, but they have fundamental limitations. As Edsger Dijkstra famously told, manual testing can only reveal the presence of bugs, it can never prove their absence [4], and extensive test suites provide no formal guarantee of correctness. Model checking, while more rigorous, is typically restricted to verifying certain classes of safety and liveness properties within finite or finitely abstracted systems. It is not well-suited for establishing full functional correctness according to a formal specification: properties such as those verified in formally verified systems like CompCert [5] and seL4 [6] would be out of reach for conventional model checking approaches.

In contrast, proof assistants , such as Rocq [7], Agda [8], Isabelle [9], F* [10], and Lean [11], allow for the formal verification of software according to such formal specification, meaning that there is a high degree of confidence that such software behaves entirely according to specification.

In the subset of proof assistants which support dependent types, such as Rocq, Agda, F* and Lean, programming and proving become synonymous thanks to the idea that types are propositions: the *Curry-Howard correspondence* [12]. Such proof assistants are in fact a programming language with a dependent type system, which enables expressing precise specifications as dependent types directly in the type system itself. The type checker then acts as a proof checker: if a program type checks, its implementation is provably correct with respect to its specification. This integration of programming and proving provides an unprecedented level of confidence in software correctness. It rules out all potential cases that violate the program's specification: the type tells us all that we want to know about the program's behaviour, and an "ok" from the type checker means that the program is *Correct-by-Construction* (CbC) [12]. Such Correct-by-Construction programming with dependent types allows for writing programs in which it is impossible to separate the behaviour of the program from its proof of correctness.

However, this raises an important question: since a type checker is itself just a piece of software, how can we trust the correctness of the type checker of a dependently-typed language? To address this, all proof assistants, dependently-typed or not, are designed around a small *trusted computing base* (TCB), minimizing the amount of software that must be assumed correct. This should increase the trustworthiness of a proof assistant, but it is not sufficient, as evidenced by the fact that there are still plenty of bugs being discovered in the TCB of proof assistants [13, 14, 15, 16]. This means we should strive to verify parts of the TCB of a proof assistant, using a proof assistant's own machinery. We call such verification *self-verification*, in order to communicate that it is trying to make a claim about the implementation of a proof

assistant using the proof assistant itself.

1.1 Agda core

The Agda Core project aims to develop a core language for Agda (Agda Core) together with a Correct-by-Construction type checker for it [17]. Agda Core's set of formally specified typing and conversion rules, in other words the *type theory*, is taken as the source of truth. Consequently, we derive the type checker implementation from this type theory, meaning that the type checker implementation is Correct-by-Construction with regards to the specified type theory. By translating every Agda term from main Agda to Agda Core, and subsequently running Agda core's type checker, such a Correct-by-Construction type checker can be used to verify the correctness of Agda programs using Agda itself, relative to this type theory. It currently supports a subset of Agda's language: features which are supported in Agda, such as eta-conversion, dependent pattern matching and universe polymorphism are missing, meaning that Agda programs using those features cannot be verified using Agda Core's type checker to provide increased trustworthiness.

1.2 Existing self-verification works

Using self-verification techniques to reduce the TCB of a proof assistant is not a new idea, and some existing works attempt to solve this problem, but each of them has a number of limitations. MetaRocq [18] aims to verify functional correctness of Rocq's type checking and extraction phase, using Rocq itself. What makes MetaRocq difficult to develop is the fact that they attempt to formalize the underlying meta-theoretical properties of the type theory which is used by the type checker. Agda Core does not aim to do this: we instead assume that the type theory satisfies all meta-theoretical properties that we want, and derive a correct implementation from that type theory. MetaRocq was started in 2014 and is still ongoing as of 2026. Formalizing the metatheory of a dependently-typed language such as Rocq is a large undertaking in terms of time and effort, as evidenced by the fact that development is still ongoing 12 years after its initiation: as of the latest paper on MetaRocq, it does not support η-conversion for functions and records, literals and Rocq modules and functors. With Agda Core: we strive to create a balance between guarantees provided and the verification effort of those guarantees.

Lean4Lean is an independent reimplementation of the Lean type checker written in Lean itself [15]. Overall, its goal is very similar to Agda core, but applied in the context of Lean. However, the current implementation of Lean4Lean fails to concretely give sound guarantees on the correctness of their proposed type checker. Furthermore, their approach relies on *extrinsic verification* rather than the *intrinsic* verification used by MetaRocq and Agda Core: with intrinsic verification, it becomes impossible to separate proofs of correctness from the implementation, while extrinsic verification allows such separation.

Strub et al. developed a type checker for F* which is verified using F*'s own machinery [19]. Their work requires a metatheory of F* in Rocq, thus relying on the soundness of Rocq. This, in effect, means that their work is not self-verification of F*. Furthermore, their approach relies on extrinsic verification rather than the intrinsic verification which is proposed by MetaRocq and Agda Core.

There also exist other works which provide more exhaustive guarantees of correctness on a type checker implementation such as McTT [20], Stitch [21] and CakeML [22], however, these are all works which rely on languages that are a lot simpler and less expressive than Agda, which means they do not face the same challenges as Agda Core, MetaRocq, Strub et al., Lean4Lean etc. In the case of McTT, it would require the programmer to program in plain Martin-Löf type theory (MLTT), which we consider to be non-feasible, and similar to Strub

et al., their work also relies on the soundness of Rocq. Stitch and CakeML do not support programming with dependent types, which neglects the advantages of programming with dependent types.

In summary, to the best of our knowledge, there exists no (fully implemented) intrinsically verified type checker for a core language of a practical dependently-typed language.

1.3 Disclaimer on Gödel's second incompleteness theorem

Because of Gödel's second incompleteness theorem, which tells that a system cannot show its own consistency, [23], it is impossible to prove the specification of a dependently-typed language correct within that same language. However, it is possible to take the language's specification as a source of truth [18], and prove correctness of the implementation of that specification in the same language, in effect moving from a trusted code base to a trusted theory base (TTB). In our case, the trusted part is the type theory that we specify in the form of typing and conversion rules, using the syntax of Agda for that purpose. So with Agda Core, we are not claiming to go against Gödel's incompleteness theorem, but we are merely minimizing the trusted parts of a dependently-typed programming language.

1.4 η-conversion

In programming languages and type theory, *η-conversion*, pronounced as eta-conversion, for functions, means adding an abstraction over a function f or eliminating such an abstraction over a function f. For functions in the well-known λ-calculus, it means that one can freely convert between $\lambda x.(f\ x)$ and f as shown in Equation 1.1.

$$\frac{}{\lambda x.(f\ x) \cong f}\ \eta\text{-FUNCTIONS} \tag{1.1}$$

where $\cong$ denotes reduction when read from left-to-right and denotes expansion when read from right-to-left, and means "*conversion*" or "*definitional equality*" in general. Consequently, for the remainder of this thesis, we call eliminating such an abstraction over a function *η-reduction*, while we call adding such an abstraction over a function *η-expansion*. We will mention the term *η-conversion* when it is not relevant to the context whether we are η-reducing or η-expanding.

η-conversion can also happen for *record types*. A record type is simply a type used for containerizing certain values into a single value: the *projections* of a record type allow us to re-access individual values of a certain element of a record type. For example, we can construct a record with *projections* `x` and `y` with values `True` and `1` by `record {x = True; y = 1}`. For records p which consist of two projections fst and snd, we can write down the η-conversion rule as shown in Equation 1.2:

$$\frac{}{record\ \{\mathit{fst} = \mathsf{fst}\ p\ ;\ \mathit{snd} = \mathsf{snd}\ p\} \cong p}\ \eta\text{-RECORDS2} \tag{1.2}$$

In general, for a record p which consists of n projections $p.1, ..., p.n$, we can write down the η-conversion rule as shown in Equation 1.3.

$$\frac{}{record\ \{p.1 = p.1\ p\ ;\ ...\ ;\ p.n = p.n\ p\} \cong p}\ \eta\text{-RECORDS} \tag{1.3}$$

Adding η-equality to a dependently-typed proof assistant makes proving using that proof assistant more straightforward, as the proof assistant is automatically able to convert terms to their η-reduced or η-expanded form, which means less proof steps are necessary to supply for the user of the proof assistant. As a secondary benefit, it can make code written in that proof assistant more efficient, since the ability for a proof assistant to do η-conversion

```
double : Nat → Nat
double x = x + x

f : List Nat → List Nat
f = map double

g : List Nat → List Nat
g xs = map double xs

example : f ≡ g
example = refl
```

Listing 1: Example program in Agda motivating the need for η-conversion

between functions, can decrease the amount of function calls necessary in order to evaluate a program. For example, consider the Agda example in Listing 1 involving the well-known `map` function, in which `f ≡ g` is well-typed because of η-conversion. Because Agda includes η-conversion, this saves the backend from having to use a closure whenever `g` is applied, and instead, any application of `g` can simply be substituted with `map double`. Finally, η-conversion is an important tool when producing computer-checked theorems in the context of Homotopy type theory [24]. In this context, η-conversion is necessary in order to be able to derive function extensionality from Homotopy type theory's Univalence axiom [25].

In the context of self-verification for major practical dependently-typed languages such as Rocq, Agda, Lean etc., we are not aware of a work or tool which has support for η-conversion for both function and record types. Lean4Lean should support η-conversion for both function and record types in theory, but as mentioned previously, the current implementation of Lean4Lean fails to concretely give sound guarantees on the correctness of their proposed type checker anyway [15]. Furthermore, the Lean prover does not support η-conversion for the *unit type*, which is the record type which contains zero fields. The η-conversion rule for the unit type is shown in Equation 1.4.

$$\frac{\Gamma \vdash a : \top \qquad \Gamma \vdash b : \top}{\Gamma \vdash a \cong b}\ \eta\text{-UNIT} \tag{1.4}$$

Agda is unique in that it supports η-conversion for the unit type, and so we would also like to have it within Agda Core, as η-conversion involving the unit type is a regular source of bugs [26, 27, 28, 29]. There appears to be no work which has given a formal specification of η-conversion for the unit type; it is in fact considered especially tricky to formalize when one wants to formalize η-conversion in a dependently-typed language [30].

MetaRocq is the most complete self-verification work currently available and supports the most features. Yet, it currently does not support η-conversion for both function and record types, because PCUIC, the core language of Rocq which MetaRocq is based on, is built on *untyped conversion*, meaning that the conversion rules of such a core language cannot refer to the type of the terms which they refer to. Since untyped conversion is fundamental to the approach that MetaRocq takes, this makes it difficult to support η-conversion according to Sozeau et al. [18]. In fact, as explained by Lennon-Bertrand, η-conversion is very difficult to support because MetaRocq wants to verify meta-theoretical properties, and adding η-equality to PCUIC consistently violates at least one of those meta-theoretical properties [25].

Agda Core also makes use of untyped conversion rules, however, we are currently not interested in verifying any meta-theoretical properties. Therefore, we think it is quite feasible

to implement such a formalized η-conversion for functions and record types with at least one field, since this can be done in a purely syntax-directed manner. However, it is impossible to do η-conversion on an element of the unit type, without knowing the type of that element. Therefore, we expect that supporting such a formalized η-conversion for the unit type would require a re-write of the conversion rules of Agda Core in order to use *typed conversion* instead. We expect that to be particularly challenging, and we are interested in the question if it can be realistically done.

Another work by Lennon-Bertrand certifies a conversion checker in Rocq, targeting a version of MLTT, which makes use of untyped conversion, supporting η-conversion for both function types and record types with at least one field [31]. One of their limitations is that they cannot certify a conversion checker which makes use of η-conversion for the unit type. This is consistent with our belief that formalizing a conversion checker which makes use of η-conversion for the unit type would need to make use of typed conversion rules.

For other major proof assistants, such as F* and Isabelle/HOL, we are not aware of an existing serious self-verification effort, and therefore, we are not aware of a work or tool which supports formalized η-conversion for both function and record types. Therefore, we now get to the main objective of this thesis.

1.5 Main objective of the thesis

In summary, related works illustrate multiple shortcomings of self-verified proof assistants. MetaRocq is lacking under the complexity of formalization, while other approaches either rely on other proof assistants such as Rocq (Strub et al.), or they are simply not mature yet (Lean4Lean). This thesis therefore attempts to advance the state of the art in self-verification, by extending Agda Core, benefiting from a manageable verification effort by means of adding features to an intrinsically verified Correct-by-Construction type checker for a core language.

Hence, we want to formalize η-conversion in the Agda Core self-verification effort, and we are interested in the question on why this has not been done before for a major dependently-typed language. Hence, we have formulated the following research questions:

1.6 Research Questions

RQ1 How can one implement η-conversion for a Correct-by-Construction type checker for a dependently-typed core language, for function types and record types with at least one field?

RQ2 How can one implement typed conversion for a Correct-by-Construction type checker for a dependently-typed core language, in order to support η-conversion for the unit type?

1.7 Thesis contributions

We make the following concrete contributions in this thesis:

- A formalization of η-conversion for function types in Agda Core using untyped conversion, including support for translation from Agda to Agda Core and practical verification on examples
- Addition of records, record projection and Correct-by-Construction type checking for record features and record projection to Agda Core, including practical verification on examples

- A formalization of η-conversion for record types with at least one field in Agda Core using untyped conversion, including practical verification on examples
- A description of how typed conversion can be added to Agda Core in order to support η-conversion for the unit type

The implementation artifact may be found on the Agda Core GitHub under tag `atejan_msc_thesis` [32], containing the implementation on untyped conversion, and under tag `atejan_msc_thesis_eta_unit` [33], containing the progress on typed conversion, both of which are discussed in Chapter 3 of the present thesis. Agda Core depends on the `scope` library, whose version part of this thesis can be found on GitHub under the specific commit referenced in the bibliography [34].

1.8 Thesis Overview

The remainder of the thesis is structured as follows: in Chapter 2, we introduce the necessary background information on Agda Core which is required in order to understand the contribution of the thesis. In Chapter 3, we introduce our results gradually, for each of the three main targets that we consider in our research questions, so for η-conversion for functions, η-conversion for record types with at least one field and η-conversion for the unit type. Chapter 4 shows to what extent we have answered our research questions from Section 1.6. Chapter 5 compares the present work with related work. Finally, we conclude and give a reflection on the present research in the broader field of Computer Science and society in Chapter 6.

Chapter 2

Background on Agda Core

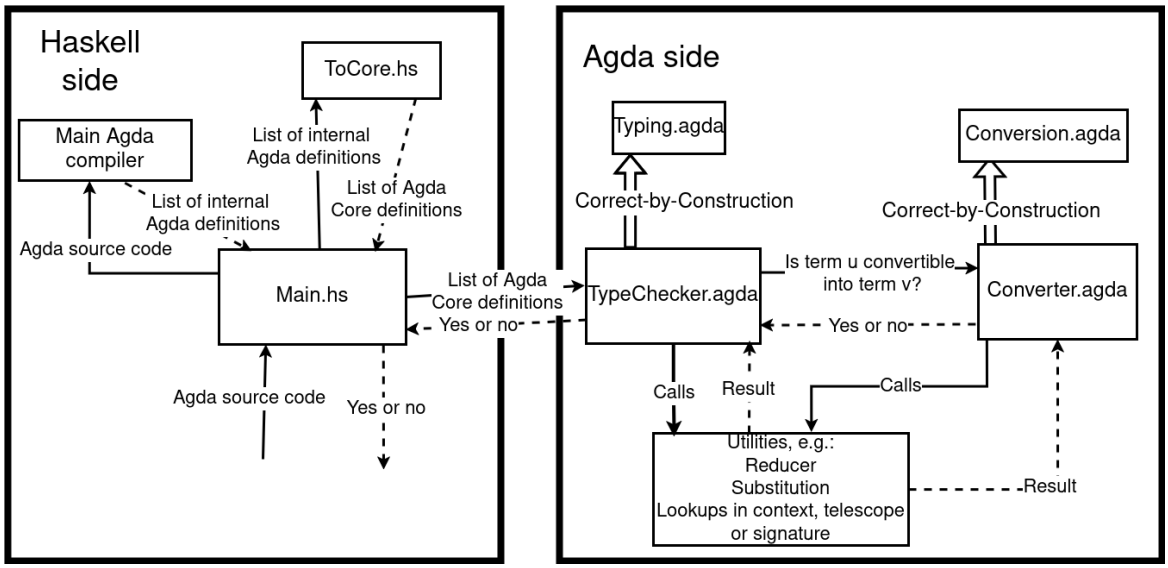


Figure 2.1: Simplified overview of Agda Core's architecture. "Correct-by-Construction" means that the file from the source node is Correct-by-Construction with respect to the target node. Solid lines represent a call relationship, while dotted lines represent the return of a result

This chapter is an updated summary of the paper by Liesnikov and Cockx [17] that introduced Agda Core: it summarizes the ideas of Agda Core that are necessary to understand the contribution of the present thesis. Figure 2.1 provides a simplified overview of the architecture of Agda Core, whose implementation is in Agda and partially in Haskell: the Haskell side acts as the entry point of the program and translates a given Agda program source to a list of Agda Core definitions. This list of definitions is type checked in `TypeChecker.agda`, the type checker implementation, and requires a conversion checker implementation `Converter.agda`. Both `TypeChecker.agda` and `Converter.agda` are Correct-by-Construction with respect to respectively `Typing.agda` and `Conversion.agda`. We cover the relevant details of the implementation in Agda from Sections 2.1 to 2.5 and the relevant details of the Haskell implementation in Section 2.6.

In Section 2.1, we give a basic overview of dependent types in Agda for the unfamiliar reader. In Section 2.2, we give an overview of Agda Core's term language. Sections 2.3 and 2.4 specify the trusted theory base of Agda Core: the Agda Core type theory and conversion theory. Section 2.5 gives an overview of the architecture of the type checker implementation and conversion checker implementation. Finally, we briefly discuss the Haskell side of Agda Core in Section 2.6.

2.1 Dependent types

Dependent types are a defining feature of programming with Agda and they are used extensively in the implementation of Agda Core. To the reader who is not familiar with Agda or dependent types, they deserve some explanation. In essence, a dependent type is a type that may depend on runtime values. They have the effect of further restricting the possible members of non-dependent types. A well-known example of such a type is the `Vector A n` type which is given in Agda as follows:

```
data Nat : Set where
  Zero : Nat
  Suc : (n : Nat) → Nat

data Vector (A : Set) : Nat → Set where
  Nil : Vector A Zero
  Cons : (n : Nat) → A → Vector A n → Vector A (Suc n)
```

For the unfamiliar reader, we first note that the given definition of `Nat` is the Peano encoding of natural numbers [35], which is the set of numbers greater than or equal to zero. `Zero` corresponds to 0, `Suc Zero` corresponds to 1, `Suc (Suc Zero)` corresponds to 2 etc. We will use this encoding of natural numbers occasionally throughout the remainder of the thesis.

As for the `Vector A n` type, one can think of this type as lists containing elements of type `A` that are of length `n`. For example, the element `Cons 1 False (Cons 0 False Nil)`, representing the list `[False ; False]` is of type `Vector Bool (Suc (Suc Zero)` but `Nil` or `Cons 0 True Nil`, representing the list `[True]` are not.

The general form of an Agda datatype definition is shown in Listing 2, where each `A_i` has the form as shown in Listing 3.

```
data D (x_1 : P_1) ... (x_k : P_k) : (y_1 : Q_1) → ... → (y_l : Q_l) → Set ℓ where
    c_1 : A_1
    ...
    c_n : A_n
```

Listing 2: Agda datatype general form [36]

Everything that appears between `data` and `Set ℓ` defines the parameters and indices of our datatype. Anything that appears *left* of the `:` is called a *parameter* whereas anything that appears *right* of the `:` is called an *index*. One can see in Listing 3 that parameters are forced to be equal between constructors, whereas the indices resulting from a constructor can differ between them.

Beyond `Vector A n`, we will now introduce some common Agda types which are used throughout the present work. These are shown in Listing 4. We note here that `_≡_` is the standard type encoding equality in Agda, i.e., an element of type `x ≡ y` is a proof that `x` is equal to `y`, and that `Σ` is the dependent pair type: the type of the second element `snd` may depend on the value of the first element `fst`.

`Vector A n` is a simple example, but in fact, dependent types can be used to embed powerful specifications directly in the type system itself, as we mentioned in Chapter 1. In the

```
    (z_1 : B_1) → ... → (z_m : B_m) → D x_1 ... x_k t_1 ... t_l
```

Listing 3: Agda datatype constructor general form [36]

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

data Bool : Set where
    True : Bool
    False : Bool

data Nat : Set where
    Zero : Nat
    Suc : (base : Nat) → Nat

record Σ (a : Set) (b : a → Set) : Set where
  field
    fst : a
    snd : b fst

data Vector (A : Set) : (length : Nat) → Set where
    Nil : Vector A Zero
    Cons : {n : Nat} → (el : A) → (vecSmaller : Vector A n) → Vector A (Suc n)
```

Listing 4: Common Agda types used throughout the present work

context of Agda Core, one important example of such a specification is the fact that the type checker only accepts terms that are *intrinsically well-scoped*. We will now explain how such well-scoped syntax is constructed in Agda Core in the next section.

2.2 Well-scoped syntax (`Term` α)

Well-scoped syntax representations [37, 38] allow terms of a programming language to capture the names of variables that can appear in those terms. The term language of Agda Core is defined as a `Term` α: it has a list of names α as a parameter, which is called a *scope*. In other words, a term of type `Term` α can only ever contain variable names that are part of the scope α. In Listing 5, we can see a subset of the term language of Agda Core.

```
data Term     (@0 α : Scope Name) : Set
data Term α where
  TVar  : (x : NameIn α) → Term α
  TDef  : (d : NameIn defScope) → Term α
  TData : (d : NameData)
        → (TermS α (dataParScope d))
        → (TermS α (dataIxScope d))
        → Term α
  TDataCon  : {d : NameData} (c : NameDataCon d)
        → (TermS α (dataFieldScope c)) → Term α
  TLam  : (@0 x : Name) (v : Term (α ▸ x)) → Term α
  TApp  : (u : Term α) (v : Term α) → Term α
  TSort : Sort α → Term α
```

Listing 5: Subset of `Term` α definition: the term language of Agda Core

We will now explain what each part in Listing 5 means what we have not explained so far.

2.2.1 Erasure notation (`@0`) and agda2hs

The implementation of Agda Core makes extensive use of *erasure notation* (`@0`) [39], which tells us that an argument or definition is *irrelevant* at runtime. These notations are necessary in order to translate the verified Agda source code to executable Haskell using the *agda2hs* library [40], which allows for the translation of Agda source code to executable Haskell source code that is easily readable by Haskell programmers. While one could use Agda's GHC backend [41] to make the source code executable, this allows for less optimization in the generated Haskell source code and is not readable to Haskell programmers. More specifically, using agda2hs makes the code on the Haskell side of Agda Core easier to develop, because one can inspect the generated Haskell code in a human-readable manner as opposed to the code generated by Agda's GHC backend. Thus, it was chosen to make Agda Core's implementation executable using agda2hs and erasure notation.

Because Haskell does not support programming with dependent types, we need to mark all indices in the Agda source code as erased using the `@0` notation, in order to communicate to agda2hs that they will be erased during translation to Haskell. For example, the `Vector A n` definition is translated to the definition of `Vector A` in Haskell as shown in Listing 6.

```
data Vector A where
    Nil :: Vector A
    Cons :: A -> Vector A -> Vector A
```

Listing 6: `Vector` in Haskell

`Term` α from Agda Code Listing 5 is translated the definition of `Term` in Haskell as shown in Listing 7.

```
data Term where
    TVar :: Int -> Term
    TDef :: Int -> Term
    TData :: Int -> [Term] -> [Term] -> Term
    TDataCon :: Int -> Int -> [Term] -> Term
    TLam :: Term -> Term
    TApp :: Term -> Term -> Term
    TSort :: Sort -> Term
```

Listing 7: `Term` in Haskell

2.2.2 Scopes and the `scope` library

Agda Core relies on the `scope` library [17, 42] for representing scopes which support well-scoped syntax. The purpose of the `scope` library is to provide a single and consistent abstract interface for scopes and their operations, such that the implementation of Agda Core does not have to deal with low-level details about scopes. We have `Scope` and `RScope` (reverse scope): in the end, both of these are just lists of erased names, but for `Scope`, items are added to the right while for `RScope` items are added to the left. We use `mempty` both for representing the empty `Scope` and the empty `RScope` whenever applicable, and we use ▶ for adding items to `Scope` while ◀ is used for adding items to `RScope`. For example, `mempty` ▶ x ▶ y is the `Scope` of list `[x ; y]` and x ◀ y ◀ `mempty` is the `RScope` of list `[x ; y]`.

```
Singleton x = ∃ _ (x ≡_)
```

Listing 8: `Singleton` definition from the `agda2hs` library

```
In : @0 name → @0 Scope name → Set
In x α = ∃ Index (λ n → IsNth x α n)
syntax In x α = x ∈ α

InR : @0 name → @0 RScope name → Set
InR x rα = ∃ Index (λ n → IsNthR x rα n)
syntax InR x rα = rα ∋ x

NameIn : (@0 α : Scope Name) → Set
NameIn α = Σ0 Name λ x → x ∈ α

NameInR : (@0 rα : RScope Name) → Set
NameInR rα = Σ0 Name λ x → rα ∋ x
```

Listing 9: Definition of `NameIn` and `NameInR`. `name` is any element of `Set`. `Σ0` corresponds to a version of the dependent pair type where the first argument is erased.

2.2.3 `Singleton`

While there are times when we need to erase argument with `@0`, as we explained in Section 2.2.1, this also has the side effect of only being able to use those arguments in a *compile-time position*, as explained in the Agda documentation [43]. To put it in simple words: it means that those arguments cannot be used in a runtime position in the generated Haskell code, because they aren't actually translated, which means that they also cannot be used in a runtime position in the original Agda code. Since there are situations where we need to use an erased argument at runtime, we have to supply a local copy of the erased argument at the site where we need it from other data that is available. This process is called *resurrection* [44]. We have the `Singleton` definition for this purpose, which is shown in Listing 8, whose type can be read as the proposition that there exists an element that is propositionally equal to `x`. We will give an example of how resurrection is commonly used throughout Agda Core in Section 2.3.2.

2.2.4 `NameIn` and `NameInR`

The type `NameIn` α corresponds to a dependent pair type of a runtime de Bruijn index of a variable and a proof that it is in scope, as shown in Listing 9. We also have a similar definition `NameInR` for `RScope` which is also shown in Listing 9. For example, if `[x ; y]` is a `Scope`, then `x` has type `NameIn [x ; y]` and if `[y ; x]` is an `RScope`, then `x` has type `NameInR [y ; x]`. Because the type checker takes terms of type `Term` α as input, it is not possible to give an ill-scoped term to the type checker. In effect, this makes it impossible to create a type checker implementation that has bugs related to scoping inside of it.

The definitions `NameIn` and `NameInR` make use of the types `IsNth`, `IsNthR`, `∃` and `Σ0`. We do not show the implementation details of these types in the present document, but they can be found in the implementation artifact [32]. We further note that elements of `∃` and `Σ0` are constructed with the syntax `a ⟨ b ⟩` and `⟨ a ⟩ b` respectively: this syntax is used whenever we need to construct an `In` and `InR` or `NameIn` and `NameInR` respectively.

2.2.5 Global scopes

```
data Vector (A : Set) : Nat → Set where
  Nil : Vector A Zero
  Cons : (n : Nat) → A → Vector A n → Vector A (Suc n)

vector0 : Vector Nat Zero
vector0 = Nil
```

Listing 10: Vector program example

```
dataParScope      : NameIn dataScope → RScope Name
dataIxScope       : NameIn dataScope → RScope Name
dataConstructors  : NameIn dataScope → RScope Name
```

Listing 11: Scopes for each datatype name

For each Agda program that is type checked using Agda Core, the type checker relies on a few global scopes. Two of these are `defScope` and `dataScope`: respectively, they contain all the names of an Agda program that are either global definitions or datatypes. For example, if we have the Agda program from Listing 10 being type checked by Agda Core, then `Vector` will be an element of `dataScope` and `vector0` will be an element of `defScope`. We also define three additional global scopes for datatype in the `dataScope`, which capture the parameters, indices and constructors of a datatype, as shown in Listing 11.

We use the shorthands `NameData` and `NameDataCon` to refer to a datatype name or constructor name part of the global `dataScope`, as shown in Listing 12. Then, we also have an additional scope for each constructor `dataFieldScope c` which is the scope of constructor argument names, as shown in Listing 13.

```
NameData : Set
NameData = NameIn dataScope
NameDataCon : NameData → Set
NameDataCon d = NameInR (dataConstructors d)
```

Listing 12: `NameData` and `NameDataCon`

```
dataFieldScope    : {d : NameIn dataScope } → NameInR (dataConstructors d)
  → RScope Name
```

Listing 13: `dataFieldScope`

2.2.6 `TermS` α $r\beta$

`TermS` is in effect a vector (from the start of Section 2.1) of `Term` α. Names which are added to the `TermS` are kept tracked of with an `RScope`, as one can see in Listing 14: therefore, we know exactly for which `RScope` of names the list of terms is supposed to be for. For example, if `f` is a global Agda definition of type `(n : Nat) -> (b : Bool) -> Nat`, then in the expression `f 0 False`, the list of arguments `0 False` is of type `TermS` α `(n ◀ b ◀ mempty)` when that list of arguments gets compiled to Agda Core.

```
data TermS    (@0 α : Scope Name) : (@0 rβ : RScope Name) → Set
data TermS α where
  TSNil  : TermS α mempty
  TSCons : {@0 rβ : RScope Name} {@0 x : Name}
    → Term α → TermS α rβ → TermS α (x ◂ rβ)

pattern ⌈⌉ = TSNil
pattern _↦_◂_ x t s = TSCons {x = x} t s
```

Listing 14: `TermS` α $r\beta$

2.2.7 `Sort` α

A *sort* or *universe* is a type whose members are also types. For example, `Set`, `Set_1`, defined as the Agda type containing `Set`, and `Set_2`, defined as the Agda type containing `Set_1`, etc. are all universes, because they are types whose members are also types.

A *sort system* is a method that avoids Russell's paradox: one cannot construct a set of all sets and assume that the result is a set [45, 12]. In the context of Agda, this means that the statement `Set : Set` is ill-typed; rather, one should have `Set : Set_1`, `Set_1 : Set_2` etc.: see the Agda 2.8 documentation on Agda's sort system for the complete explanation [46].

Since sorts are present in Agda, we must also support sorts in Agda Core and they are stated as in Listing 15, as simply being natural numbers. This is unchanged from the paper by Liesnikov and Cockx [17].

```
data Sort     (@0 α : Scope Name) : Set

data Sort α where
  STyp : Nat → Sort α
```

Listing 15: `Sort` α

2.3 Prerequisites for defining Agda Core's Type theory

In this section, we explain some prerequisites that are necessary to understand before giving the main definition of Agda Core's type theory in Section 2.4. In essence, a type theory for a programming language is just a collection of typing rules, specifying expected behaviour for a type checker, with form as shown in Equation 2.1.

$$\frac{ANTECEDENT}{\Gamma \vdash t : ty} \tag{2.1}$$

where $ANTECEDENT$ is a collection of zero or more logical statements that we assume to hold, and $\Gamma \vdash t : ty$ is the conclusion that we define to hold when all of $ANTECEDENT$ is true. In the notation $\Gamma \vdash t : ty$, Γ is a context that maps variables to concrete language terms, t is a term of the language and ty is a type of the language and we read this statement as "Under the context Γ, the term t has type ty".

We have already given the definition of what a term is in Agda Core in Section 2.2. We will now first discuss what a type and context is in Agda Core in Sections 2.3.1 and 2.3.2, then we will define other prerequisites that are necessary for modelling Agda Core's type theory, before introducing the main definition of Agda Core's type theory in Section 2.4.

2.3.1 `Type` α

In order to define our type theory, we must first define what a type is in Agda Core. Types are simply terms with a sort, as shown in Listing 16.

```
record Type   (@0 α : Scope Name) : Set

record Type α where
  inductive; no-eta-equality; pattern
  constructor El
  field
    typeSort : Sort α
    unType   : Term α
```

Listing 16: `Type` α

Since `Type` α is defined in terms of `Term` α, types are also well-scoped. An example of Agda syntax that we will compile to an element of `Type` α is `Vector Nat Zero` from Listing 10.

2.3.2 `Context` α

In order to define our type theory, we need a data structure to represent the Γ as we defined it in Equation 2.1. Contexts are mappings from variables to types as shown in Listing 17.

```
data Context : @0 Scope Name → Set where
  CtxEmpty  : Context mempty
  CtxExtend : Context α → (@0 x : Name) → Type α → Context  (α ▸ x)
pattern _,_:_ Γ x a = CtxExtend Γ x a
```

Listing 17: `Context` α

In Listing 17, the syntax `pattern _,_:_ Γ x a = CtxExtend Γ x a` means that the syntax `Γ , x : a` is equivalent to the syntax `CtxExtend Γ x a`. We will use the notation `Γ , x : a` to mean extending the context `Γ` with the name `x` being bound to the type `a` further in the present work. We also note that a `Context` α is *closed*: this means that for any context ctx and for each pair $(x, t) \in ctx$, it holds that that each variable name appearing in t is locally known in the same context. An simple example where we need to add a variable to the context in order to type check a definition is shown in Listing 19. In this case, we would need to add `x` to a context with type `Nat`. So when we would type check the `x + 2` part of `addTwo`, the context would be `CtxEmpty , x : Nat`.

Because a `Context` α is an indexed type containing a `Scope` α, we can use the earlier introduced `Singleton` type from Section 2.2.3 to resurrect a `Scope` from a `Context` α. This happens commonly throughout Agda Core's codebase and the function doing this procedure is shown with its typing signature in Listing 18.

```
singScope : (Γ : Context α) → Singleton α
```

Listing 18: Function `singScope` which produces a `Singleton` α from a `Context` α for some scope α

```
addTwo : Nat → Nat
addTwo = λ x → x + 2
```

Listing 19: Example Agda definition where a context is necessary to type check it

We use the familiar notation $\Gamma \vdash t : ty$ to denote that a $t \in$ `Term` α has a certain type $ty \in$ `Type` α under a certain $\Gamma \in$ `Context` α. We will later give an exact specification of $\Gamma \vdash t : ty$ in Section 2.4, which defines the trusted type theory of Agda Core.

2.3.3 `Telescope` α `r`β

```
data Telescope (@0 α : Scope Name) : @0 RScope Name → Set where
  EmptyTel  : Telescope α mempty
  ExtendTel : (@0 x : Name) → Type α → Telescope (α ▸ x) rβ  → Telescope α (x ◂ rβ)

pattern ⌈⌉ = EmptyTel
infix 6 _:_◂_
pattern _:_◂_ x t Δ = ExtendTel x t Δ
```

Listing 20: `Telescope` α `r`β

A telescope is a data structure used in Agda's backend [47], used in the declaration of datatypes and record types among others. To effectively reason about Agda Core types involving datatypes and records types in Agda Core's type theory, we need to give a definition of telescope in Agda Core as well, which telescopes from the Agda backend will be translated into.

A telescope is like a `Context` α: in effect they just map variable names to types, as shown in Listing 20.

An example of a telescope in Agda is the `(A : Set) : Nat` part of the `Vector` definition from Listing 10: since `Nat` is not bound to a name in this Agda definition, a name is automatically generated by the Agda backend before translation from Agda to Agda Core.

2.3.4 Signature

The signature acts as a global environment for the type checker, containing Agda definitions, datatype definitions, datatype constructor definitions and record definitions. It is defined as a record with three elements as shown in Listing 21.

We will not explain in detail all of the data that is contained within a `Datatype d` or a `DataConstructor c`. They are record types that contain data such as the telescope of a

```
record Signature : Set where
  no-eta-equality
  field
    sigData : (d : NameData) → Datatype d
    sigDefs : (f : NameIn defScope)  → Type mempty × SigDefinition
    sigCons : (d : NameData) (c : NameDataCon d) → DataConstructor c
```

Listing 21: `Signature` definition. The type `SigDefinition` is equivalent to the type `Term mempty`.

```
data Term     (@0 α : Scope Name) : Set
data Term α where
    TPi   : (@0 x : Name) (u : Type α) (v : Type (α ▸ x)) → Term α
```

Listing 22: Definition of `TPi` term in Agda Core, which models a dependent function type, which is a function type where the output type `v` may depend on the input `x`, an element of type `u`

```
data TyTerm  (@0 Γ : Context α) : @0 Term α → @0 Type α → Set
syntax TyTerm Γ u t = Γ ⊢ u : t

data TyTerm {α} Γ where
  TyTVar : {x : NameIn α}
    ---------------------------------
    → Γ ⊢ TVar x : lookupVar Γ x

  TyLam :
      Γ , x : a ⊢ u : b
    -------------------------------
    → Γ ⊢ TLam x u : El k (TPi x a b)

  TyApp : {b : Type α}

    → Γ ⊢ u : El k (TPi x b c)
    → Γ ⊢ v : b
    ------------------------------------
    → Γ ⊢ TApp u v : substTop (singScope Γ) v c

  TyConv :
      Γ ⊢ u : a
    → unType a ≅ unType b
    ----------------
    → Γ ⊢ u : b
```

Listing 23: Subset of Agda Core's trusted type theory (Part I). α is of type `Scope Name`

datatype and the telescope of arguments of a constructor. For the exact definitions, we refer the reader to the implementation available on GitHub [32].

2.4 `TyTerm` $\Gamma\ t\ ty$ $(\Gamma \vdash t : ty)$

We will now state the definition of $\Gamma \vdash t : ty$ for Agda Core. We mentioned in Chapter 1 that Agda Core uses its own type theory as its source of truth. We also mentioned in the introduction of Section 2.3 that a type theory is just a collection of typing rules which define the expected behaviour of a type checker implementation. A subset of Agda Core's trusted typing rules are shown in Listing 23 and Listing 24. In all cases where `sig` is shown, we mean an object with type `Signature` as discussed in the previous section.

For example, the typing rule `TyTVar` says that if x is a name in a certain scope α, then under context Γ , `TVar x` has type the of looking up x in context Γ. This is equivalent to the same typing rule in mathematical notation as shown in Equation 2.2.

```
data TyTerm {α} Γ where
  TyData :
      {d : NameData}
      {@0 pars : TermS α (dataParScope d)}
      {@0 ixs  : TermS α (dataIxScope  d)}
      (let dt  : Datatype d
           dt  = sigData sig d)
    → Γ ⊢ˢ pars : instDataParTel dt
    → Γ ⊢ˢ ixs  : instDataIxTel dt pars
    ------------------------------------------------
    → Γ ⊢ TData d pars ixs : sortType (instDataSort dt pars)

  TyDataCon :
      {d : NameData}
      {c : NameDataCon d}
      {@0 pars : TermS α (dataParScope d)}
      {@0 us  : TermS α (dataFieldScope c)}
      (let dt  : Datatype d
           dt  = sigData sig d
           con : DataConstructor c
           con = sigCons sig d c)

    → Γ ⊢ˢ us : instConIndTel con pars
    ------------------------------------------------------------
    → Γ ⊢ TDataCon c us : dataConstructorType dt con pars us
```

Listing 24: Subset of Agda Core's trusted type theory (Part II), containing the typing rules for `TData` and `TDataCon`. α is of type `Scope Name`

$$\frac{x \in \alpha}{\Gamma \vdash \texttt{TVar x} : \texttt{lookupVar}\ \Gamma\ x}\ (\textsc{TyTVar}) \tag{2.2}$$

Slightly more complicated examples of typing rules are `TyLam` and `TyApp`, which target the terms `TLam` and `TApp` respectively, and both make use of the `TPi` Agda Core term shown in Listing 22. `TPi` models a dependent function type as explained in Listing 22. Both typing rules are again equivalent to their mathematical representations shown in Equation 2.3 and 2.4. We note that `substTop (singScope Γ) v c` does substitution inside the term `c` with argument `v`: for details on how substitution is implemented, the reader is referred to the implementation artifact [32].

$$\frac{\Gamma, x : a \vdash u : b}{\Gamma \vdash \texttt{TLam x u} : \texttt{El k (TPi x a b)}}\ (\textsc{TyLam}) \tag{2.3}$$

$$\frac{\Gamma \vdash u : \texttt{El k (TPi x b c)} \qquad \Gamma \vdash v : b}{\Gamma \vdash \texttt{TApp u v} : \texttt{substTop (singScope Γ) v c}}\ (\textsc{TyApp}) \tag{2.4}$$

The rules `TyData` and `TyDataCon` from Listing 24 also deserve an explanation: we will first give the definition of $\Gamma \vdash^s ts : tel$ which we have not given so far, before explaining `TyData` and `TyDataCon` individually. Because a `tel : Telescope` α `r`β is a list of types for a specific `RScope r`β, and a `ts : TermS` α `r`β also contains an RScope rβ in its definition, we say that

a `TermS` α `r`β can have a certain `Telescope` α `r`β under a certain $\Gamma \in$ `Context` α. We denote this relation by $\Gamma \vdash^s ts : tel$.

TyData

One can read the rule for `TyData` as: "if the given `pars` is of the correct telescope, and if the given `ixs` is of the correct telescope, then `TData d pars ixs` is well-typed and has a defined sort type". We will now give an explanation for what each part of the stated typing rule means that we have not explained so far: `instDataParTel dt` has type `Telescope` α `(dataParScope d)`. This means it is the telescope of the parameters of the datatype `d`. For example, for the type `Vector`, this is the telescope of length 1 with element `(A : Set)`. `instDataIxTel dt pars` has type `Telescope` α `(dataIxScope d)`. This means that it is the telescope of the indices of a datatype `d`. For example, for the type `Vector`, this is the telescope with length 1 with element `n1 : Nat`, where `n1` is any automatically generated name by Agda. As for `instDataSort`, `(instDataSort dt pars)` is simply the sort of the datatype (a `Sort` α). Finally, `sortType` transforms a `Sort` α to a `Type` α and is defined as `sortType (STyp a) = El (STyp (a + 1)) (TSort (STyp a))`.

TyDataCon

One can read the rule for `TyDataCon` as "if `us` is of the correct telescope, then `TDataCon c us` is well-typed and has the type that the constructor returns given `pars` and `us`". We will now give an explanation for what each part of the stated typing rule means that we have not explained so far: `instConIndTel con pars` has type `Telescope α (dataFieldScope c)`. This means that it is the telescope of the input parameters of a constructor of a datatype. For example, for the `Cons` constructor of `Vector`, this is the telescope `(n : Nat) (a : A) (v : Vector A n)` where `a` and `v` are any automatically generated non-overlapping names by Agda. Furthermore, `dataConstructorType` gives the type that the constructor `con` returns given `pars` and `us`. For example, this is the type `Vector A (Suc n)` for the `Cons` constructor of `Vector`.

TyConv

`TyConv` from Listing 23 is the typing rule which tells that if a certain `u : Term` α has a certain `Type` α `a` under a certain Γ `: Context` α, and if `a`'s underlying term is convertible into the underlying term of another `Type` α `b`, then `u` also has the `Type` α `b` under the same Γ. This typing rule makes use of the conversion relation $\cong$, whose definition we will give now.

2.4.1 `Conv u v` ($\mathbf{u \cong v}$)

Since the type theory makes use of a conversion relation $\cong$, we are required to define a relation which tells whether two terms are equal, in other words, whether two terms can be converted into each other. In effect, such a relation is a set of rules that specify the behaviour of a function that decides equality between terms. We call the set of rules a *conversion theory*: a subset of these rules are shown in Listing 25. For example, the rule `CRefl` tells us that, without any additional assumptions, a `Term` α `u` can always be converted into itself. Similarly, the rules `CData` and `CDataCon` for terms `TData` and `TDataCon` tell us that if the `TermS`s of their sub-arguments can be converted into each other, then we can also convert `TData` and `TDataCon` using those sub-arguments into each other respectively.

The `convert` function, shown in Listing 29, that implements the equality check for terms is Correct-by-Construction with regards to the conversion theory, whose definition we defer to Section 2.5.3. We call such a function a *conversion checker*.

```
data Conv      {@0 α} : @0 Term α → @0 Term α → Set
syntax Conv x y        = x ≅ y

data ConvTermS {@0 α} : @0 TermS α rβ → @0 TermS α rβ → Set
syntax ConvTermS us vs = us ⇔ vs

data Conv {α} where
  CRefl  : u ≅ u
  CLam   : {@0 r : Singleton α}
         → renameTop {y = z} r u ≅ renameTop {y = z} r v
         → TLam y u ≅ TLam z v
  CApp   : u ≅ u'
         → w ≅ w'
         → TApp u w ≅ TApp u' w'
  CData  : (@0 d : NameData)
           {@0 ps qs : TermS α (dataParScope d)}
           {@0 is ks : TermS α (dataIxScope d)}
         → ps ⇔ qs
         → is ⇔ ks
         → TData d ps is ≅ TData d qs ks
  CDataCon : {@0 d : NameData} (c : NameDataCon d)
           {@0 us vs : TermS α (dataFieldScope c)}
         → us ⇔ vs
         → TDataCon c us ≅ TDataCon c vs
  CRedL  : @0 ReducesTo u u'
         → u' ≅ v
         → u  ≅ v
  CRedR  : @0 ReducesTo v v'
         → u  ≅ v'
         → u  ≅ v
```

Listing 25: Subset of Agda Core's trusted conversion theory. α is of type `Scope Name`

2.5 Type checker implementation

We first state the definition of TCM, which the type checker implementation relies on, before detailing the architecture of the type checker implementation as well as the conversion checker implementation.

2.5.1 TCM

Agda is a total language, which means that every function in Agda should both terminate and give a result, but since the Agda source is translated to Haskell to generate an executable type checker implementation, we need a way to define failure in Agda. We do this with a simple type checking monad, which we call TCM, which is unchanged from the paper by Liesnikov and Cockx [17] as shown in Listing 26.

We will use TCM in several code blocks using Agda's *do-notation*, i.e., we use it with the keywords `do`, `←` and `return`. For the reader who is not familiar with monads or do-notation, a good explanation for both can be found on the Haskell Wiki page about monads [48].

```
record TCM (a : Set) : Set where
  pattern; no-eta-equality
  constructor MkTCM
  field runTCM : TCEnv → Either TCError a
open TCM public

tcError : TCError -> TCM a
tcError = MkTCM ∘ const ∘ Left
```

Listing 26: TCM monad. "∘" is function composition.

2.5.2 Bi-directional type checker

The type checker implementation follows the standard framework of bidirectional type checking [49, 50]. This means the type checker implementation consists of two functions `checkType` and `inferType` that are defined in a mutually recursive manner. Both of these functions return typing derivations constructed from `TyTerm` from Listing 23, and as the name of these functions hint at, `checkType` is called against checkable terms, while `inferType` is called against inferrable terms, and additionally returns the inferred type of the term. In the standard framework of bidirectional type checking, datatypes of constructors, lambda terms and let statements, which Agda Core supports but we have not introduced explicitly, are non-inferrable terms and thus have to be supplied with a type to enable type checking. All other terms part of the term language of Agda Core are inferrable terms. The signature of both `checkType` and `inferType` is shown in Listing 27.

As mentioned in Chapter 1, the type checker implementation is Correct-by-Construction with regards to the type theory specified in Listing 23 and Listing 24, because the type checker cannot determine success unless it returns a typing derivation from the rules specified in `TyTerm`, as one can see from the typing signature of `checkType` and `inferType` in Listing 27. This property is more generally known in the literature as *positive soundness* [31]. We further note that, for now, the type checker implementation is *partial*, that is: there is no guarantee that if the type checker fails, there was actually no typing derivation. In other words, for now, the type checker implementation does not satisfy *negative soundness* [31].

```
checkType : ∀ (Γ : Context α) u (ty : Type α) → TCM (Γ ⊢ u : ty)
inferType : ∀ (Γ : Context α) u → TCM (Σ[ t ∈ Type α ] Γ ⊢ u : t)
```

Listing 27: `checkType` and `inferType`

2.5.3 Conversion checker implementation

```
step : (rsig : Singleton sig) (s : State α) → Maybe (State α)
```

Listing 28: Reducer for terms

```
convert : Singleton α → ∀ (t q : Term α) → TCM (t ≅ q)
```

Listing 29: `convert` signature. α is of type `Scope Name`

The conversion checker relies on a reducer for terms, which remains largely unchanged from the paper by Liesnikov and Cockx [17]. The only change is that `step` needs to be given a resurrected signature as input, as shown in Listing 28. The implementation of `convert` is then simply a wrapper which:

- Calls `convertCheck`, which checks whether there is sufficient *fuel* left for the conversion problem: fuel is a natural number that specifies the maximum combined recursion depth of the conversion checker and reducer, in order to ensure that conversion and reduction always terminates
- `convertCheck` reduces both input terms to a normal form
- `convertCheck` calls `convertTerms` on the reduced terms, which does pattern matching on the input terms. `convertTerms` is mutually recursive with `convertCheck`

This process can be seen in Listing 30. We further note that the reducer is currently not written according to a specification in a Correct-by-Construction manner, so it must be considered part of the trusted code base of Agda Core for now.

```
convertTerms : ⟦ fl : Fuel ⟧ → Singleton α → (t q : Term α) → TCM (t ≅ q)

convertCheck ⟦ None ⟧ r t z =
  tcError "not enough fuel to check conversion"
convertCheck ⟦ More ⟧ r t q = do
  t ⟨ tred ⟩ ← reduceTo r t
  q ⟨ qred ⟩ ← reduceTo r q
  (CRedL tred ○ CRedR qred) <$> convertTerms r t q

convert r t q = do
  I ⟦ fl ⟧ ← tcmFuel
  convertCheck r t q
```

Listing 30: Implementation of `convert`

2.6 Haskell side & Agda to Agda Core translation

All the concepts that we have explained so far in this Chapter are related to the Agda side of Agda Core. However, there is also a Haskell side, functioning as the main program that Agda Core supplies and which the user invokes. This Haskell side consists of a main module `Main.hs` and a translation module `ToCore.hs`, and it translates Agda terms and definitions [51] to Agda Core terms and definitions from Listing 5. This allows us to type check actual Agda programs using our Correct-by-Construction type checker, instead of only being able to type check Agda Core programs.

Chapter 3

Results

We intend to introduce the results gradually in the following fashion: first, we give example programs in Agda which we would like to type check with Agda Core. Then, we show which conversion rule(s) and/or typing rule(s) we need to add to Agda Core's type theory as introduced in Chapter 2. This amounts to adding to the trusted theory base. Finally, we show which modifications we need to add to the type checker implementation, using the rules that we have added, in order to type check the example programs that we have shown.

We aim to introduce the results of our implementation in increasing order of perceived complexity. Section 3.1 discusses how we have added support for η-conversion for functions to Agda Core. Section 3.2 discusses how we have added records to Agda Core and how we have added support for untyped η-conversion for records of at least one field. Finally, Section 3.3 discusses the progress on adding support for a conversion checker implementation that uses typed conversion to Agda Core in order to be able to support the unit type.

3.1 η for functions

We start by adding untyped η-conversion for functions to the converter of Agda Core, because this is the most straightforward addition to the type checker and conversion checker. We remind the reader that we would like to introduce the the Rule η-FUNCTIONS 1.1 from the introduction into the conversion system of Agda Core.

3.1.1 Example programs

An Agda definition which requires an η-conversion rule for functions in order to type check using Agda is the `eta-functions_expl` definition in Listing 31.

```
eta-functions_expl : (A B : Set) (f : A → B) → (f ≡ (λ x → (f x)))
eta-functions_expl = λ A B → λ f → refl
```

Listing 31: `eta-functions_expl` example

We would like to use the definition `eta-functions_expl` as a test case for testing η-conversion on functions, but it would not be a good test case in practice, because the type of `eta-functions_expl` is initially checked by the main Agda type checker during the translation of the definition from Agda to Agda Core. During this translation, the Agda type checker would itself apply η-reduction on `(λ x → (f x))`: this results in the Agda type `(A B : Set) (f : A → B) → f ≡ f` being given to Agda Core as the type of `eta-functions_expl` when the definition is translated. This means that `eta-functions_expl` does not actually require η-conversion within Agda Core to type check with Agda Core, meaning that we have to make the `eta-functions_expl` definition slightly more complicated.

A slightly more complicated Agda definition which does require an η-conversion rule for Agda Core is the `eta-functions_expl'` Agda definition in Listing 32.

```
const : (A : Set) → A → A → A
const = λ A → λ x → λ y → x

eta-functions_expl' : (A B : Set) (f : A → B) → (f ≡ (λ x → (f (const A x x))))
eta-functions_expl' = λ A B → λ f → refl
```

Listing 32: `eta-functions_expl'` example

With the call to `const`, Agda is not able to automatically infer that the term `λ x → (f (const A x x))` is η-reducible, thus, the main Agda type checker will not attempt such η-reduction. One can think of this `const` function as any identity function at compile time, for which it is not obvious for the Agda type checker that it is actually such an identity function.

Another definition which requires η-conversion in the converter is the `eta-app-1` definition, in which the λ-binder is on the other side of the equation, thus testing symmetry, as seen in Listing 33.

```
eta-app-1 : (f : Nat -> Nat -> Nat) ->
  (λ x -> (f zero x)) ≡ (f (const Nat zero (suc zero)))
eta-app-1 = λ f → refl
```

Listing 33: `eta-app-1` example

3.1.2 Conversion rules

In order to add η-conversion to the Agda Core's converter, we formulate two conversion rules which try to η-expand any Agda Core term to an η-expanded form; one for each side as shown in Listing 34.

```
  CEtaFunctionsLeft : (@0 x : Name) (f : Term α) (b : Term (α ▸ x)) →
    let subsetProof = subWeaken subRefl in
      b ≅ (TApp (weakenTerm subsetProof f) (TVar (VZero x)))
      → f ≅ (TLam x b)
  CEtaFunctionsRight : (@0 x : Name) (f : Term α) (b : Term (α ▸ x)) →
    let subsetProof = subWeaken subRefl in
      b ≅ (TApp (weakenTerm subsetProof f) (TVar (VZero x)))
      → (TLam x b) ≅ f
```

Listing 34: η-conversion rules for functions in Agda Core

There are a couple of concepts to explain for each of the arguments that these conversion rules take: Firstly, they take an erased name `x`, which is the name binder of the λ-term in which we are trying to expand. Secondly, they take an `f : Term α`, which is a well-scoped term of Agda Core. `α` is a Scope of names, and `f` being of type `Term α` means that any names that appear free in `α` can only refer to the names that are present in `α`. This is the function term that we are trying to expand, so it is any `Term α` which may reduce to a value that is a function. Thirdly, they take a `b : Term (α ▸ x)`, which means that it is an Agda Core term in which only names that are in the scope `α` can occur, and additionally `x`, as `α ▸ x` means

that the name x is added to the scope α. This will be the body of our λ-term in which we are trying to expand. Finally, they take a proof that b is convertible to an application of f with argument x. There are two additional concepts to explain here as well:

- We need to convince Agda that f can be used as a `Term (α ▸ x)`. This is trivial, because f only contains names of scope α and thus it also only contains names of scope α ▸ x. (subWeaken subRefl) serves as this proof (subsetProof) and the function weakenTerm transforms f from a `Term α` to a `Term α ▸ x`.
- We need to convince Agda that `TVar x` is also a `Term (α ▸ x)`. This is trivial, because `TVar x` clearly only contains x as a name, and `(VZero x)` is an Agda pattern that returns this proof.

If the caller of this CEtaFunctions rule can supply a values of all these types, then we can conclude that f can be η-expanded and thus the return type is `f ≅ (TLam x b)`. Another remark regarding the construction of this conversion rule is that we really need both CEtaFunctionsLeft and CEtaFunctionsRight, because we cannot add a generic CSym conversion rule for symmetry to the conversion theory as shown in Listing 35.

```
CSym : {a b : Term α}
  → b ≅ a
  → a ≅ b
```

Listing 35: Bad conversion rule CSym

Adding a rule such as CSym would mean that we could add behaviour to the conversion checker implementation that would apply CSym would go in an infinite loop if the two terms are not actually convertible; so we manually have to apply symmetry for each conversion problem where it is needed in order to get a trustworthy conversion theory.

3.1.3 Conversion checker implementation

We show how we update the conversion checker implementation after adding CEtaFunctionsLeft and CEtaFunctionsRight to the conversion theory in Listing 36.

Most of what is happening in Listing 36 is simply constructing the objects which we require to call the conversion rules CEtaFunctionsLeft and CEtaFunctionsRight. There are two concepts that deserve an explanation:

- To check whether b is actually convertible to an application of f with argument x, we simply recursively call convertCheck in monadic code, which fails if b is not convertible to such a term.
- We need to construct a new scope newScope for the convertCheck function because we are introducing a new variable x in the conversion problem. This involves calling a library function singBind that returns a new Singleton of the desired scope.

3.1.4 Updates to the Agda to Agda Core translation

During the implementation of η-conversion for functions, there was a need to update the translation module ToCore.hs in order to support the compilation of datatype applications correctly to TData correctly, which was not happening previously. For instance, the translation of the type `f ≡ (λ x → (f (const A x x)))` previously gave a translation error, which meant that we could not actually type check the definition from Listing 32. This problem

```
convertEtaFuncsGeneric : ⟦ fl : Fuel ⟧
                    → Singleton α
                    → (@0 x : Name)
                    → (f : Term α)
                    → (b : Term (α ▸ x))
                    → TCM (b ≅ (TApp (weakenTerm _ f) (TVar (VZero x))))
convertEtaFuncsGeneric r x f b = do
  let
    subsetProof = subWeaken subRefl
    newScope    = singBind r
    term        = TApp (weakenTerm subsetProof f)
                       (TVar (VZero x))
  convertCheck newScope b term

convertTerms : ⟦ fl : Fuel ⟧ → Singleton α → (t q : Term α) → TCM (t ≅ q)
convertTerms r functionTerm (TLam x b) =
  do
    conversionProof <- convertEtaFuncsGeneric r x functionTerm b
    return (CEtaFunctionsLeft x functionTerm b conversionProof)
convertTerms r (TLam x b) functionTerm =
  do
    conversionProof <- convertEtaFuncsGeneric r x functionTerm b
    return (CEtaFunctionsRight x functionTerm b conversionProof)
```

Listing 36: Change to `convertTerms` to enable η-conversion for functions in the conversion checker implementation

was spotted merely due to the fact that not a single term in the translation module was being compiled to a `TData`. Such problems with the translation highlight the need to write the translation module in Agda with a trusted specification as well, as opposed to Haskell.

3.1.5 Summary of the contribution

The presented modifications on the Agda side are accepted by the Agda type checker, and the updates to the translation module allows us to type check all examples from Section 3.1.1. This means that we have given a Correct-by-Construction implementation of η-conversion for functions in Agda within Agda Core, using a syntax-directed conversion style.

3.2 Untyped η-conversion for records with at least one field

We want to formalize within Agda Core the η-conversion rule as shown in Equation 1.3 in the introduction chapter, excluding the `n=0` possibility.

3.2.1 Example programs

In order to add support for untyped η-conversion for record types to Agda Core, we attempt to type check the Agda program which is shown in Listing 37. This means that we must add the following features to Agda Core:

- Records, record constructors and record projection in Agda Core's term language from Listing 5

- Correct-by-Construction type checking for record constructors and record projection
- Modification of the conversion theory and conversion checker implementation to support η-conversion for record types with at least one field

In particular, taking into account the example in Listing 37:

- `containerX` and `sigmaRecordElement` tests type checking for record constructors/records
- `sigmaRecordElementProjSnd` tests type checking record projection
- `eta-R-one_fixed` tests η-conversion for records

```
record ContainerRecord : Set where
    field
        theProj : Bool

eta-R-one_fixed : (c : ContainerRecord) →
    _≡_ ContainerRecord c (record { theProj = ContainerRecord.theProj c})
eta-R-one_fixed = λ c → refl

containerX : ContainerRecord
containerX = (ContainerRecord.constructor False)

sigmaRecordElement : Σ Nat (λ n → (Vector Bool n))
sigmaRecordElement = Σ.constructor (Suc (Suc Zero)) (Cons False (Cons False Nil))

sigmaRecordElementProjSnd : Vector Bool (Suc (Suc Zero))
sigmaRecordElementProjSnd = sigmaRecordElement .Σ.snd
```

Listing 37: Example record program, used as a positive test case for record features in Agda Core

Furthermore, we also use the program in Listing 38 as a negative test case: for now, the definition `etaExpandEmptyRec` should be rejected by Agda Core, even though it is accepted by Agda, as this definition requires typed conversion to type check, and we test that our updates still do not type check this definition. We will also use the `Pair` example shown in Listing 39 in order to explain some concepts related to the type checker implementation, but we do not use the `Pair` example as an example to evaluate our results.

We will now describe the design of the required changes in order to support the listed features from the start of the section.

```
record EmptyRecord : Set where
    constructor MkEmptyRecord

etaExpandEmptyRec : (a : EmptyRecord) → (a ≡ EmptyRecord.constructor)
etaExpandEmptyRec = λ a → refl
```

Listing 38: Example record program, used as a negative test case for record features in Agda Core

```
record Pair (A B : Set) : Set where
    field
        fst : A
        snd : B
```

Listing 39: Example of a record `Pair A B`

3.2.2 Addition of records to Agda Core

We add new scopes to the global scopes as shown in Listing 40. The three scopes `recScope`, `recParScope` and `recFieldScope` are similar to `dataScope`, `dataParScope` and `dataFieldScope` respectively. `recScope`, is the global scope that contains all the names of the Agda program that are records. `recParScope` is the scope of a record's parameters. For example, for the `Pair` record, `recParScope Pair = [A ; B]`. `recFieldScope` is the scope of the record constructor, and thus by definition also the list of projections of a record. For example, `recFieldScope Pair = [fst ; snd]`. We also define shorthand notation `NameRec` and `NameProj rn` to define a name in the record scope and a projection function of a record `rn` respectively.

```
recScope          : Scope Name
recParScope       : NameIn recScope → RScope Name
recFieldScope     : NameIn recScope → RScope Name
NameRec : Set
NameRec = NameIn recScope
NameProj : NameRec → Set
NameProj rn = NameInR (recFieldScope rn)
```

Listing 40: Record global scopes

We add support for two new terms in the term language: `TRec` for record types and `TRecCon`, serving as a record constructor for record types, as shown in Listing 41. We add a new field `sigRecs` to the signature with type `(recordName : NameRec) → Record recordName` for accessing record information in the type checker implementation.

```
TRec : (rn : NameRec) → TermS α (recParScope rn) → Term α
TRecCon : (r : NameRec) → (TermS α (recFieldScope r)) → Term α
```

Listing 41: Additions to Agda Core term language to support records

For our trusted type theory, we add typing judgments `TyRec` and `TyRecCon` for `TRec` and `TRecCon` as shown in Listing 43. These typing judgments function very similar to `TyData` and `TyDataCon` as explained in Section 2.4, and therefore we do not discuss them in detail. For `TyRec`, we note that we only need to check that `pars` is of the correct telescope, because records in Agda do not have indices. For our trusted conversion theory, we add conversion judgments `CRec` and `CRecCon` that our untyped conversion checker makes use of as shown in Listing 42. These are also very similar to `CData` and `CDataCon` as shown in Listing 25, thus we will not discuss them in detail.

```
CRec : (@0 rn : NameRec)
         (@0 pars1 pars2 : TermS α (recParScope rn))
         → pars1 ⇔ pars2
         → TRec rn pars1 ≅ TRec rn pars2
CRecCon : (rn : NameRec)
        {@0 args1 args2 : TermS α (recFieldScope rn)}
      → args1 ⇔ args2
      → TRecCon rn args1 ≅ TRecCon rn args2
```

Listing 42: Additions to Agda Core's untyped conversion theory: conversion rules `CRec` and `CRecCon` for terms `TRec` and `TRecCon` respectively

```
  TyRec :
    {rn : NameRec}
    {@0 pars : TermS α (recParScope rn)}
    (let rt : Record rn
         rt = sigRecs sig rn)
    → Γ ⊢ˢ pars : instRecParTel rt
    ------------------------------------------------
    → Γ ⊢ TRec rn pars : sortType (instRecSort rt pars)

  TyRecCon :
    {rn : NameRec}
    {@0 pars : TermS α (recParScope rn)}
    {@0 args : TermS α (recFieldScope rn)}
    (let sigRecord : Record rn
         sigRecord = sigRecs sig rn)
    → Γ ⊢ˢ args : instRecConArgTel sigRecord pars
    ------------------------------------------------------------
    → Γ ⊢ TRecCon rn args : recordConstructorType sigRecord pars
```

Listing 43: Additions to Agda Core's type theory: typing rules `TyRec` and `TyRecCon` for terms `TRec` and `TRecCon` respectively

3.2.3 Addition of record projection to Agda Core

We add a constructor `TProj` to Agda Core's term language that takes a `Term` α and a projection function of a record as shown in Listing 44

```
TProj : {rn : NameRec} (u : Term α) (x : NameProj rn) → Term α
```

Listing 44: `TProj`

We also add a rule to the conversion checker `CProj` as shown in Listing 45. This rule is very similar to `CRec` and `CRecCon`, thus, we will not discuss it in detail.

Correct-by-Construction type checking for TProj

We need to add Correct-by-Construction type checking for `TProj` such that any Agda programs that make use of projection are type checked correctly. Therefore, we add the rule as

```
  CProj :
          {rn : NameRec}
          {f : NameProj rn}
          {recTerm1 recTerm2 : Term α}
          → recTerm1 ≅ recTerm2
          → (TProj recTerm1 f) ≅ (TProj recTerm2 f)
```

Listing 45: `CProj` conversion rule for `TProj`

```
lookupNameRinTel : (rs : Singleton α) (rrs : Singleton rβ)
(cargs : TermS α rβ) (tel : Telescope α rβ) (n : NameInR rβ) → Type α
lookupNameRinTel _ _ _ EmptyTel x = nameInRemptyCase x
lookupNameRinTel {α} rs ((_ :: rrs') ⟨ rrseq ⟩) (TSCons argTerm smallerTermS)
    (ExtendTel y typ smallerTel) x =
    let
      result : Type (α ▸ y)
      result = nameInRBindCase x
        ((λ q → lookupNameRinTel
          (singBind rs)
          (singTermS smallerTermS)
          (weakenTermS (subBindDrop subRefl) smallerTermS)
          smallerTel (⟨ _ ⟩ q)))
        (λ proof → weakenType (subBindDrop subRefl) typ)
    in
    substTop rs argTerm result
```

Listing 46: Definition `lookupNameRinTel`

```
projectionType : {rn : NameRec}
                 (Γ : Context α)
                 (cargs : TermS α (recFieldScope rn))
                 (sigRecord : Record rn)
                 (instPars : TermS α (recParScope rn))
                 (projFunc : NameProj rn)
                 → Type α
projectionType ctx cargs sigRecord instPars projFunc = lookupNameRinTel
  (singScope ctx)
  (singTermS cargs)
  cargs
  (instRecConArgTel sigRecord instPars) projFunc
```

Listing 47: `projectionType` helper function of the typing rule `TyProj`

shown in Listing 48. Informally, we can read `TyProj` as follows: first, `recordTerm` needs to be of a record type. Then, we also need to make sure that `recordTerm` was constructed via `TRecCon` with some `cargs : TermS` α `(recFieldScope rn)`, because we need the arguments that the record was constructed with in the return type. We do this by checking that `recordTerm` is convertible into a `TRecCon rn cargs` for some `cargs : TermS` α `(recFieldScope rn)`. If those two conditions hold, then `TProj recordTerm projFunc` has the type of looking up the entry `projFunc` in the record constructor telescope instantiated with `instPars`, given record

```
TyProj : {rn : NameRec}
   {recordTerm : Term α}
   {rsort : Sort α}
   {projFunc : NameProj rn}
   {instPars : TermS α (recParScope rn)}
   (cargs : TermS α (recFieldScope rn))
   (let sigRecord : Record rn
        sigRecord = sigRecs sig rn)
   → Γ ⊢ recordTerm : (El rsort (TRec rn instPars))
   → recordTerm ≅ (TRecCon rn cargs)
   ------------------------------------------------------------------------
   → Γ ⊢ TProj recordTerm projFunc :
     projectionType Γ cargs sigRecord instPars projFunc
```

Listing 48: Typing rule `TyProj`. `lookupNameRinTel` is given in Listing 46. `projectionType` is given in Listing 47

constructor arguments `cargs`, as shown in the `projectionType` helper function of Listing 47. The function `lookupNameRinTel` is given in Listing 46: that function makes use of the added `scope` library functions `inRemptyCase` and `inRbindCase` through wrappers `nameInRemptyCase` and `nameInRBindCase`.

`lookupNameRinTel` works in the following manner: it iterates over the `rβ RScope` which is an index of the given telescope, and returns the value of the current entry in the telescope if it matches the given `x : NameInR rβ` and recurses on the smaller telescope otherwise. Because that type `result` is a `Type α ► y`, we need to transform it into a `Type α` by substituting in it any occurrences of `y` with `argTerm`, the value that `y` is equal to. We subsequently update the type checker implementation such that the type of `TProj` is inferred, since `TProj` is an inferrable term in the standard framework of bidirectional type checking.

The reducer is updated to support reducing a `TRecCon` when a projection argument is given on the evaluation stack. This requires a function `lookupNameRinTermS` with type `TermS α rγ → NameInR rγ → Term α` for some α (a `Scope`) and $r\gamma$ (an `RScope`). The implementation of `lookupnameRinTermS` is similar to and simpler than `lookupNameRinTel` since no substitution needs to happen in the resulting term, thus we will not discuss it in detail.

New additional functions in the `scope` library

In order to implement the functions `lookupNameRinTel` and `lookupNameRinTermS`, we require two new functions in the scope library: `inRemptyCase` of type `(mempty ∋ x) → a` and `inRbindCase` of type `(y ◂ rβ) ∋ x → (rβ ∋ x → a) → (@0 x ≡ y → a) → a` where `mempty` is an empty `RScope`, `rs ∋ n` is a proof that `n` is part of the RScope `rs`, the same `rs ∋ n` as in Listing 9 and `a` is any datatype. These functions have been implemented and are accepted by the Agda type checker.

Discussion

We could have chosen not to implement `inRbindCase` in `scope` and instead try to re-use the function `inBindCase` by translating a `NameInR rβ` to a `NameIn α`, where α is the Scope that with the same elements in the same order as the RSCope rβ. This would require a function of type `(y ◂ rα) ∋ x` to type `x ∈ (α ▸ y)`. This was found to induce too much additional proof labour: particularly, it was not obvious how to rewrite the goal by Agda into a form that could be easily proven. Therefore, it was chosen to implement the function `inRbindCase` directly instead.

```
lengthOfRScope : {@0 rscope : RScope Name} → Singleton rscope → Nat
lengthOfRScope ([] ⟨ refl ⟩) = zero
lengthOfRScope ((Erased name :: names) ⟨ refl ⟩) =
suc (lengthOfRScope (names ⟨ refl ⟩))

etaProjTermS : {@0 rscope : RScope Name} → Singleton rscope
→ (NameInR rscope → Term α) → TermS α rscope
etaProjTermS ([] ⟨ refl ⟩)                    _  = TSNil
etaProjTermS ((Erased name :: names) ⟨ refl ⟩) f =
name ↦ f (⟨ name ⟩ inRHere) ◂ etaProjTermS
  (names ⟨ refl ⟩)
  (λ where (⟨ x ⟩ p) → f (⟨ x ⟩ inRThere p))

data Conv {α} where
  CEtaRecordsLeft :
    (rn : NameRec)
    (rt : Term α)
    (argsTermS : TermS α (recFieldScope rn))
    (singScope : Singleton (recFieldScope rn))
    (proofNonEmpty : ∃ Nat (λ n → lengthOfRScope singScope ≡ suc n))
    → let func = (TProj {rn = rn} rt)
          termSToConvertInto = etaProjTermS singScope func
          in
      (argsTermS ⇔ termSToConvertInto)
    → rt ≅ (TRecCon rn argsTermS)
  CEtaRecordsRight :
    (rn : NameRec)
    (rt : Term α)
    (argsTermS : TermS α (recFieldScope rn))
    (singScope : Singleton (recFieldScope rn))
    (proofNonEmpty : ∃ Nat (λ n → lengthOfRScope singScope ≡ suc n))
    → let func = (TProj {rn = rn} rt)
          termSToConvertInto = etaProjTermS singScope func
          in
      (argsTermS ⇔ termSToConvertInto)
    → (TRecCon rn argsTermS) ≅ rt
```

Listing 49: Untyped η-conversion rules for records in Agda Core

3.2.4 η-conversion rule for records

Now that we have built in support for records and record projection in Agda Core, we can add an η-conversion rule for functions, similar to what we have done for functions in Section 3.1.2. These can be seen in Listing 49. These rules can be read as follows: If `rt` is a `Term` α and `argsTermS` is a `TermS` α `(recFieldScope rn)` for some `rn`, then if `argsTermS` is of non-zero length, by a proof that the underlying `RScope` is of non-zero length) and if `argsTermS` is convertible into the `TermS` α `(recFieldScope rn)` consisting of `p.1 rt, p.2 rt, ..., p.n rt`, the list of terms of all of `rn`'s projection functions applied to `rt`, then `rt` is convertible into a `TRecCon rn` with arguments `argsTermS`. This reads exactly as Conversion Rule 1.3 from the introduction, disregarding the `n=0` possibility. Note that we need to explicitly check that `argsTermS` is of non-zero length, otherwise, we would be saying that `TRecCon rn argsTermS` is

convertible into any `rt` of type `Term` α.

Like for `CEtaFunctionsLeft` and `CEtaFunctionsRight`, we cannot have a general case for symmetry in the conversion relation, so we must specify two different rules where the consequent has swapped sides.

3.2.5 Conversion checker implementation

```
checkNonEmpty : {@0 rscope : RScope Name} →
  (singScope : Singleton rscope) →
  TCM (∃ Nat (λ n → lengthOfRScope singScope ≡ suc n))
checkNonEmpty singScope with lengthOfRScope singScope
checkNonEmpty singScope    | zero  = tcError "Cannot apply untyped eta-conversion
  for records on a record whose
  constructor takes zero arguments"
checkNonEmpty singScope    | suc n = return (n ⟨ refl ⟩)

convertTerms r recTerm (TRecCon rn argsTermS) = do
    let singScope = singTermS argsTermS
    proofNonEmpty ← checkNonEmpty singScope
    convProof ← convertEtaRecsGeneric r recTerm argsTermS
    return (CEtaRecordsLeft rn recTerm argsTermS singScope proofNonEmpty convProof)
convertTerms r (TRecCon rn argsTermS) recTerm = do
    let singScope = singTermS argsTermS
    proofNonEmpty ← checkNonEmpty singScope
    convProof ← convertEtaRecsGeneric r recTerm argsTermS
    return (CEtaRecordsRight rn recTerm argsTermS singScope proofNonEmpty convProof)
```

Listing 50: Change to `convertTerms` to enable η-conversion for records in the conversion checker implementation

The addition to the conversion checker implementation is very similar to the addition that we made to support η-conversion for functions, as can be seen in Listing 50. Therefore, we do not discuss it in detail: the only difference is that we need to generate a proof at runtime that the underlying `RScope` of the `TermS` α `(recFieldScope rn)` is non-empty, and throw an error otherwise.

3.2.6 Testing

We test our record additions to Agda Core on the positive test case from Listing 37. Unlike when we added support for η for functions, we do not test the added features using Haskell translation, because in the course of attempting to test record features using the same method as described for η for functions, unexplained runtime behaviour during type checking of an Agda definition with Agda Core was common, such as abrupt halts of the program execution. Because we concluded that this behaviour could not occur on the Agda side, it must occur in the Haskell side. Therefore, for testing record features, we drop the Haskell side and work completely in the Agda side, by modelling the exact Agda program that we want to test manually in Agda Core.

We model the exact program from Listing 37 in Agda Core explicitly in a file `TestProjection.agda` [32]. This file tests for each Agda Core definition of the program from Listing 37 that it has the correct Agda Core type with a call to the type checker implementation. Since the Agda type checker successfully type checks the file, we can conclude that all tests pass and

```
record EmptyRecord : Set where
    constructor MkEmptyRecord

test : (a b : EmptyRecord) → a ≡ b
test = λ a b → refl
```

Listing 51: Agda program which requires the unit η-conversion rule to type check

that we have correctly added the features discussed in Section 3.2.1, so records, records constructors, record projections, Correct-by-Construction type checking for record features and untyped η-conversion for record types with at least one field.

We also test that the negative test case from Listing 38 still does not type check: we also do this by manually encoding the exact program in Agda Core explicitly in a file `TestEtaRecordsUnit.agda`. This file tests that `etaExpandEmptyRec` returns the correct error constructed with `tcError` which tells that untyped η-conversion cannot happen for record types with zero fields. Since this test successfully passes, we can conclude that our implementation of untyped η-conversion for records avoids doing any η-conversion of record types with zero fields.

3.2.7 Summary of the contribution

To re-iterate, we have added records, records constructors, record projections, Correct-by-Construction type checking for record features and untyped η-conversion for record types with at least one field to Agda Core and shown successful application on the example from Listing 37. We have also shown that η-conversion is still rejected for the example from Listing 38. This means that we have correctly given a formal specification of untyped η-conversion for records with at least one field within Agda Core, using a syntax-directed conversion style.

3.3 Typed conversion for Agda Core

We want to formalize within Agda Core the η-conversion for unit rule as shown in Equation 1.4 in Chapter 1.

3.3.1 Example program

In order to add support for type checking files which require the η-unit rule, we attempt to type check the Agda program which is shown in Listing 51: specifically we are interested in type checking the Agda definition `test`. Fundamentally, the definition `test` requires us to state the conversion theory and conversion checker implementation in a different method to what we have introduced in Section 2.4, because for the definition `test` to type check, we need to have a conversion rule with a pre-condition that two `Term` α's `a` and `b` have the same unit type, meaning we need to be able to mention that a term has a specific type, which the current conversion theory does not allow us to formulate.

We will now give an overview of the required changes to the design of Agda Core in order to both be able to type check an example such as `test` while also still supporting all the features that we have discussed up until now. We have not implemented the following changes as of yet, although we expect the following description of required implementation changes to be possible with reasonable amount of implementation effort.

```
data Conv      {@0 α} (@0 Γ : Context α) : @0 Term α → @0 Term α → Type α → Set
data ConvTermS {@0 α} (@0 Γ : Context α) :
    (@0 rβ : RScope Name) → @0 TermS α rβ → @0 TermS α rβ → Set

syntax Conv Γ x y ty        = Γ ⊢ x ≅ y : ty
syntax ConvTermS Γ rscope us vs = Γ [ rscope ]  ⊢ us ⇔ vs

data Conv {α} Γ where
  CRefl : {ty : Type α}
    → Γ ⊢ u : ty
    → Γ ⊢ u ≅ u : ty
  ...
```

Listing 52: Structure of typed conversion implementation

3.3.2 Structure of a typed conversion implementation

In effect, we need to modify the conversion relation $\cong$ to include a `Context` α, such that we can give a requirement in the pre-condition of any conversion rule that a specific `Term` α has a specific `Type` α under a specific `Context` α. We also add a type to the relation: this enforces an invariant that the conversion checker implementation can only be invoked on terms that we have already determined to have the same type. This models the same style as the "term-directed typed conversion" from the paper by Lennon-Bertrand [31]. We show how to do this in Listing 52.

We then need to:

- Modify all existing conversion rules to return the type $\Gamma \vdash u \cong v : ty$ rather than the type `u` $\cong$ `v`.
- Refactor the conversion checker in order to use the new conversion relation.

In our previous untyped conversion approach, the conversion theory was specified in a separate Agda module `Conversion.agda`. We now also need to ensure that we declare the conversion theory within the module that declares the type theory (`Typing.agda`), because we are now developing a theory in which the type theory may contain a conversion judgments and in which the conversion theory may contain typing judgments. In effect, this means we are now declaring the type theory and conversion theory, as well as the implementations of the type checker and conversion checker, in a mutually recursive manner.

3.3.3 Supporting the unit type in Agda Core using typed conversion

Once we have a typed conversion implementation as have described in the previous section, supporting η-conversion for the unit type will be as simple as adding a `CUnit` rule to the conversion relation as shown in Listing 53. In such a conversion rule `CUnit`, we take an input type `tUnit : Type` α and a proof that `tUnit` is a unit type. We define this proof as an Agda datatype with a single constructor, which tells that only `TRec rn mempty` is a unit type in Agda Core, where `mempty` is the empty `RScope`. This is consistent with what Agda defines to be a unit type in the Agda documentation [52], as also shown in Listing 54. We can then read `CUnit` as the same conversion rule described in Chapter 1 from Equation 1.4: if two terms `u` and `v` are both of the unit type under a certain context, then they are convertible under that context.

```
data IsUnitType : {@0 α : Scope Name} (@0 typ : Type α) → Set where
  TRecEmptyParsIsUnit : {@0 rn : NameRec}
    (emptyPars : TermS α (recParScope rn))
    → ((recFieldScope rn) ≡ mempty) → IsUnitType (El (STyp 0) (TRec rn emptyPars))

data Conv {α} Γ where
  CUnit : (tUnit : Type α)
          → IsUnitType tUnit
          → Γ ⊢ u : tUnit
          → Γ ⊢ v : tUnit
          → Γ ⊢ u ≅ v : tUnit
```

Listing 53: Proposal for an η-unit conversion rule in Agda Core

```
record ⊤ : Set where
```

Listing 54: Agda unit type

Finally, we expect that updating the type checker and conversion checker implementation to use the new `CUnit` rule will allow us to type check the example of Listing 51, assuming that we are going to test it by manually encoding the program from that Listing into Agda Core and testing that every Agda Core definition has the expected Agda Core type, in other words, in the same way as we have tested practicality of the record features in Section 3.2.6.

3.3.4 Discussion

An alternative to the presented structure would be to drop the type as an index in the conversion relation. This makes it very challenging to implement the conversion rule for `TLam`, because we do not know what the type of the binder in lambda would be, therefore, we did not choose to pursue this route.

Beyond this, the conversion relation can have different meanings, depending on how it is read. One can read $\Gamma \vdash t1 \cong t2 : ty$ as "$\Gamma \vdash t1 : ty$, $\Gamma \vdash t2 : ty$ and under context Γ, $t1$ and $t2$ are convertible at type ty" or one can read the relation as "Under context Γ, $t1$ and $t2$ are convertible at type ty if it holds that $\Gamma \vdash t1 : ty$ and $\Gamma \vdash t2 : ty$". The current presentation of the `CUnit` rule assumes the former, because we need to ensure in the pre-condition of `CUnit` that $\Gamma \vdash: tUnit$ and $\Gamma \vdash v : tUnit$. This means that we must produce a typing derivation by calling `inferType` from the conversion checker, which could cause termination issues in Agda, meaning we need to put a fuel limit on the combined calls of `convert` and `inferType`. The latter would mean that we drop the requirements $\Gamma \vdash t1 : ty$ and $\Gamma \vdash t2 : ty$ in all of our conversion rules for typed conversion, avoiding the aforementioned problem: we then need to ensure that both terms are of the same type in the type checker. Currently, it is not clear which approach would cause the least implementation challenges.

3.3.5 Summary of the contribution

We have given a description on how to implement typed conversion in Agda Core in order to support η-conversion for the unit type within Agda Core. While we have not shown practicality of this description on an example by implementing it, we expect to be able to implement it into Agda Core on top of the added record features with reasonable implementation effort.

Chapter 4

Evaluation

To evaluate the research presented in this thesis, we first re-iterate our research questions from Chapter 1:

RQ1 How can one implement η-conversion for a Correct-by-Construction type checker for a dependently-typed core language, for function types and record types with at least one field?

RQ2 How can one implement typed conversion for a Correct-by-Construction type checker for a dependently-typed core language, in order to support η-conversion for the unit type?

We give a separate evaluation for both research questions. In all of the discussion below, we take Agda Core to be the dependently-typed core language that we are referring to in the research questions.

4.1 Evaluation of RQ1: untyped conversion

We give a separate evaluation for η-conversion for function types and η-conversion for record types with at least one field.

4.1.1 Function types

In Chapter 3, we have shown how we have implemented η-conversion for function types within Agda Core by adding two conversion rules to the conversion theory, updating the conversion checker implementation, and updating the Haskell-side translation module. Hence, the implementation of η-conversion for functions is Correct-by-Construction with respect to the conversion theory, and because the Agda Core type checker makes use of the conversion checker as explained in Chapter 2, we have implemented η-conversion for a Correct-by-Construction type checker for Agda Core for function types.

We have also shown the practicality of this implementation on an example program, more specifically on the `eta-functions_expl` and `eta-app-1` definitions presented in Section 3.1.1.

Therefore, the combination of the stated implementation and the practicality of the implementation on the stated examples answer the "function types" part of RQ1.

Threats to validity

A mistake in the Haskell translation module could threaten the validity of the given answer of the "function types" of RQ1. We have already stated previously in Chapter 2 that the translation module is given in Haskell, and therefore it is not Correct-by-Construction. Furthermore, it is also not tested individually. Therefore, a mistranslation could induce false

positives in the conversion checker implementation, even if the conversion theory as specified is trustworthy, meaning that our observed output of the type checker outputting success for the stated examples would be incorrect. We postulate that any such mistranslation mistakes are fixable by formally verifying the translation module as well, in other words writing it in Agda according to a formal specification, which is a long-term goal of Agda Core anyway, so we do not think mistranslation issues could pose a long-term validity-threat to the stated answer of the "function types" part of RQ1.

We think it is unlikely that there could be a mistake in the implementation of η-conversion for functions as presented in Chapter 3 as long as one is willing to trust the conversion rules presented in Listing 34. We think those rules are trustworthy, because they model the mathematical η-conversion rule as presented in Chapter 1 in Equation 1.1, and since the implementation of η-conversion for functions is Correct-by-Construction according to those rules, we think it is extremely unlikely that there is a mistake in the implementation of η-conversion for functions in the conversion checker implementation.

4.1.2 Record types with at least one field

Our evaluation for η-conversion for record types with at least one field is largely similar to the evaluation for η-conversion for functions, with the exception that we have tested the practicality of the method by encoding an example Agda program manually within Agda Core.

In Chapter 3, we have shown how we have implemented η-conversion for record types with at least one field within Agda Core by having added the following features: records, record projection, Correct-by-Construction type checking for records and record projection and untyped η-conversion rules for record types with at least one field in a similar fashion of those added for function types. Hence, the implementation of η-conversion for records with at least one field is Correct-by-Construction with respect to the conversion theory, and because the Agda Core type checker implementation makes use of this updated conversion checker, we have implemented η-conversion for a Correct-by-Construction type checker for Agda Core for records types with at least one field.

We have also shown the practicality of this implementation on an example program, more specifically, the example in Listing 37. More specifically, the `eta-R-one_fixed` definition of the listing makes use of the capability of the conversion checker to do η-conversion for record types with at least one field. This was done, as stated in Section 3.2.6 by manually encoding the program from Listing 37 in Agda Core and testing that each Agda Core definition has the expected Agda Core type.

Therefore, the combination of the stated implementation and the practicality of the implementation on the stated example answer the "record types with at least one field" part of RQ1

Threats to validity

The validity of the given answer of the "record types with at least one field" part of RQ1 could be threatened by a mistake in encoding the program from Listing 37 into Agda Core manually. More specifically, when stating parameter, constructor and field scope of data and record types, for example `dataIxsScope`, `dataConstructors`, `recFieldScope` etc., and when stating the signature object of the type checking problem, one needs to give the exact de Bruijn index of each name instead of the name itself, as we cannot pattern match on the name itself using Agda as it is erased. This process can be error-prone, so we need to trust the fact that we have not made a mistake while writing this exact information in our test case. Furthermore, we could have also made a mistake when writing down the exact terms that we intended to write down in the test case, although we consider this to be less likely, as each term that we

create needs to be well-scoped anyway and the Agda type checker will refuse any ill-scoped term that we give.

Similarly for the evaluation for function types, we postulate that any such mistakes in encoding the test case are fixable by formally verifying the translation process as well, as this would outline any mistake that could be made in manually specifying an example Agda program in Agda Core's syntax.

Furthermore, as also stated in Section 4.1.1, we consider it unlikely that there is a mistake in the implementation of η-conversion for record types with at least one field as presented in Chapter 3, as long as one is willing to trust the conversion rules presented in Listing 49. We think those rules are trustworthy, because they model the mathematical η-conversion rule as presented in Chapter 1 in Equation 1.3, and since the implementation of η-conversion for record types with at least one field is Correct-by-Construction according to those rules, we think it is extremely unlikely that there is a mistake in the implementation of η-conversion for record types with at least one field in the conversion checker implementation.

4.2 Evaluation of RQ2: typed conversion

In Chapter 3, we have given a description of a possible implementation of typed conversion for Agda Core, and we have also stated how we aim to support the unit η-conversion rule. We have not shown that we can implement that description yet, though if it were implemented, we would have given an implementation of typed conversion for a Correct-by-Construction type checker for Agda Core which supports η-conversion for the unit type. Assuming that we have a passing test case when manually encoding the Agda program from Listing 51 and testing that each Agda Core definition has the expected Agda Core type, the description of the implementation serves as the as the answer to RQ2, on the basis that it shows feasibility towards formalizing η-conversion within Agda Core using typed conversion.

4.2.1 Threats to validity

The main threat to validity of the given answer to RQ2 would be the situation when further research results in the conclusion that the description of the proposed implementation would in fact be infeasible. Although we realize that we are making a strong claim that the description would be relatively simple to implement, we do not think this threat is very likely, as we have already made partial progress on the implementation of the description from Section 3.3 and as of the time of writing the present work, we do not see concrete implementation challenges that would prevent a description like the one from Section 3.3 to be implemented, other than that extra weeks of implementation time would be required.

Furthermore, after we would have implemented typed conversion into Agda Core, the threats to validity mentioned in the previous section on untyped conversion for record types with at least one field apply here as well, so their counterarguments apply as well.

Chapter 5

Related work

We are not aware of another work that directly tries to formalize η-conversion for functions and record types for a dependently-typed core language. Therefore, the related works mostly fall into the category of self-verification works that aim to provide a self-certified kernel for a proof assistant: which will always include a self-certified type checker and will include a self-certified conversion checker if the language in question is dependently-typed. For these related works, we will discuss their η-conversion capabilities against the present work provided in this thesis, as well as compare them in general against Agda Core, in decreasing order of perceived proximity.

MetaRocq by Sozeau et al. [18] supplies a type checker for PCUIC, a dependently-typed core language of Rocq, which is proven both sound and complete in Rocq itself, assuming strong normalization, but lacking any η-conversion capabilities. However, it would be assumable that if MetaRocq did not want to verify meta-theoretical properties about their type system within their development, they could speculatively also add η-conversion to their development as we have done in this thesis. This could pose a challenge in the present work: if in the future, we also want to verify meta-theoretical properties of the type theory of Agda Core, as presented in Listing 23 and Listing 24, then it is assumable that the formalization of η-conversion for function types and record types with at least one field as we have done in the current development requires more effort to suit verification of such meta-theoretical properties, although it should be far from impossible with our approach. We argue the novelty in the present work to be the demonstration of feasibility towards supporting a formalization for η-conversion for the unit type using an implementation of typed conversion with reasonable formalization complexity: Rocq does not support η-conversion for the unit type and thus MetaRocq also does not.

Lean4Lean by Carneiro [15] supplies a type checker for Lean, 4 written in Lean, which supports η-conversion for function types and record types with at least one field. However, Lean4Lean currently only shows partial progress towards verification of their type checker, while our existing new development on Agda Core is already verified, as it is Correct-by-Construction with regards to our type theory. Furthermore, Lean4Lean cannot support η-conversion for the unit type in the same way as Agda Core could. While Lean does support two features which behave like the unit type, namely the Prop universe [53] and K-like types [54, 55], the team behind Lean seems to be content with having an incomplete type checker implementation with regards to these features, meaning that any self-verifier for Lean can also not support eta for the unit type in the same way that the combination of Agda and Agda Core can.

Lennon-Bertrand [31] provides a certified conversion checker that makes use of untyped conversion, which they call term-directed typed conversion, which supports both η-conversion for function types and η-conversion for record types with at least one field. Their main limitation is that their conversion checker does not support η-conversion for the unit type, and they

also argue that it is a fundamental limitation of their chosen approach. That being said, they do prove negative soundness and termination of their conversion checker, assuming normalisation, in addition to positive soundness, while for now we only prove positive soundness of the type checker and conversion checker.

McTT by Jang et al. [20] build an implementation of a kernel for a core Martin-Löf type theory, verified in Rocq and extracted to OCaml, which supports η-conversion for functions, but does not support programming with records, meaning that it lacks η-conversion for records. Unlike Agda Core, they aim to verify metatheory, similar to MetaRocq, and while we only provide a type checker implementation with positive soundness, they verify all components of their implementation except the lexer and pretty-printer. We argue that the relevance in our work encompasses a formalization of η-conversion for records including a demonstration of feasibility for supporting the unit type, and that Agda Core more closely resembles the core language of Agda when compared to McTT, as well as light-weightedness of the formalization technique of our work.

Martin-Löf à la Coq by Adjedj et al. [56] mechanize the metatheory of Martin-Löf type theory within Rocq, building upon a previous work in Agda, and also obtain a certified and executable type checker for their version of MLTT. McTT is overall an improvement over their paper, so the discussed advantages and limitations discussed apply here as well.

Strub et al. [19] develop a type checker for F* written in F*, certified using Rocq. Their work requires a metatheory of F* in Rocq, which Agda Core does not require of Agda. In our opinion, this means that their work is not true self-certification, as the correctness of their results depend on the correctness of Rocq, not only of F*. Moreover, F* removed support for η-conversion for functions in 2021 [57], and the book on F* [10] as well as their GitHub [58] make no mention of η-conversion for records, so overall, this means that F* has no η-conversion capabilities, meaning that the work by Strub et al also does not have such capabilities.

Finally, there are also related works of self-verified type checkers for simpler languages: Stitch by Eisenberg et al. [21] and CakeML by Kumar et al. [22] fall into this category. A self-verifier for non-dependently typed languages does not face many of the implementation challenges of dependently-typed languages and also neglects the broader advantages of programming with dependent types, so we consider them to be less relevant than the other related works presented before.

Chapter 6

Conclusion

In this thesis, we have presented a formalization of Agda's η-conversion within Agda Core for function types and record types with at least one field, adding support for records along the way, using untyped conversion. We have also given a description on how to formalize Agda's often tricky considered η-unit conversion rule using typed conversion. Overall, it teaches us how to self-verify η-conversion for Agda, providing an important step towards the ultimate goal of a self-verified type checker for Agda and a valuable contribution for constructing self-verified type checkers for practical dependently-typed programming languages in general.

Immediate future work will be to implement the description of typed conversion as stated in Chapter 3 as well as to show practicality of it by encoding the example from Listing 38 in Agda Core and checking that each Agda Core term has the expected Agda Core type. After this, we want to add support for additional open features in Agda Core: these include dependent pattern matching, universe polymorphism, positivity checking, a verified reducer of Agda Core terms etc. [59]. After Agda Core supports these additional features, future work will include a verified translation module by writing it in a Correct-by-Construction manner in Agda according to a specification, in order to address the threats to validity presented in Chapter 4. After there exists a verified translation module, it could be interesting to see whether we could support *negative soundness* to get a complete type checker, in the same way as MetaRocq has. Further work after that will be to add Agda Core translation and type checking as a step to the main Agda compiler and type checker, for the ultimate goal of a self-verified implementation of the Agda type checker.

6.1 Reflection on the research in the broader field of Computer Science and society

As mentioned in Chapter 1, we consider the research presented in this thesis to be relevant to the broader field of Computer Science and society, primarily in the sense that it advances the knowledge on how to build a trustworthy, bug-free, self-verified dependently-typed proof assistant, namely Agda. By showing how to formally verify the η-conversion of Agda within Agda Core, we are showing more generally how we can verify the correctness of proof assistant features known to cause bugs within them, which provides lessons for verifying other features of proof assistants that are known to cause bugs.

The research of the present thesis is relevant for several different stakeholders: for the software engineer using a dependently-typed proof assistant, it provides increased confidence that there is not a bug in the proof assistant that would mean that the result of their verified software in Agda does not hold in practice. For a mathematician who uses Agda to computer-check mathematical theorems, it provides increased confidence in the correctness of their theorems which are not refuted due to a bug in the kernel of Agda. For developers of dependently-typed proof assistants, it provides advice on proof techniques that can be

used to further the ultimate goal of self-verification for a dependently-typed proof assistant. Finally, end-users of software written using a dependently-typed programming language benefit from software having less bugs due to miscompilation or a faulty type checker of those languages.

Bibliography

[1] Gregory Tassey. *NIST Planning Report 02-4*. June 2002.

[2] Herb Krasner. *Cost of Poor Software Quality in the U.S.: A 2022 Report*. 2022. URL: https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/ (visited on 06/11/2026).

[3] Edmund M. Clarke et al., eds. *Handbook of Model Checking*. Cham: Springer International Publishing, 2018. ISBN: 978-3-319-10574-1 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8. URL: http://link.springer.com/10.1007/978-3-319-10575-8 (visited on 12/16/2024).

[4] Professor P. Ercoli and Professor Dr. F. L. Bauer. *SOFTWARE ENGINEERING TECHNIQUES*. Apr. 1970. URL: https://repositories.lib.utexas.edu/server/api/core/bitstreams/1891a34f-a566-4ea4-b3e0-9c3c56791226/content.

[5] Xavier Leroy et al. "CompCert - A Formally Verified Optimizing Compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. Toulouse, France: SEE, Jan. 2016. URL: https://inria.hal.science/hal-01238879 (visited on 11/04/2025).

[6] Gerwin Klein et al. "seL4: formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. SOSP '09. New York, NY, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: https://dl.acm.org/doi/10.1145/1629575.1629596 (visited on 11/04/2025).

[7] *The Rocq Prover 9.1.0 documentation*. URL: https://rocq-prover.org/doc/V9.1.0/refman/index.html (visited on 11/04/2025).

[8] *Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/overview.html (visited on 11/04/2025).

[9] *Isabelle*. URL: https://isabelle.in.tum.de/ (visited on 01/13/2026).

[10] Nikhil Swamy, Guido Martínez, and Aseem Rastogi. *Proof-Oriented Programming in F**. URL: https://fstar-lang.org/.

[11] *Lean Programming Language*. Lean Language. URL: https://lean-lang.org (visited on 11/04/2025).

[12] Benjamin Pierce. *Types and Programming Languages*. Feb. 2002.

[13] *Compilation of critical bugs in stable releases of Coq*. GitHub. URL: https://github.com/rocq-prover/rocq/blob/master/dev/doc/critical-bugs.md (visited on 11/04/2025).

[14] *soundness bug: native_decide leakage*. URL: https://leanprover-community.github.io/archive/stream/270676-lean4/topic/soundness.20bug.3A.20native_decide.20leakage.html (visited on 11/04/2025).

[15] Mario Carneiro. *Lean4Lean: Verifying a Typechecker for Lean, in Lean*. Sept. 14, 2025. DOI: 10.48550/arXiv.2403.14064. arXiv: 2403.14064[cs]. URL: http://arxiv.org/abs/2403.14064 (visited on 11/03/2025).

[16] *Eta-contraction is not type-preserving · Issue #2732 · agda/agda*. GitHub. URL: https://github.com/agda/agda/issues/2732 (visited on 06/25/2026).

[17] Bohdan Liesnikov and Jesper Cockx. "Building a Correct-by-Construction Type Checker for a Dependently Typed Core Language". In: *Programming Languages and Systems*. Ed. by Oleg Kiselyov. Singapore: Springer Nature, 2025, pp. 63–83. ISBN: 978-981-97-8943-6. DOI: 10.1007/978-981-97-8943-6_4.

[18] Matthieu Sozeau et al. "Correct and Complete Type Checking and Certified Erasure for Coq, in Coq". In: *Journal of the ACM (JACM)* (Nov. 2024). Publisher: Association for Computing Machinery, pp. 1–76. DOI: 10.1145/3706056. URL: https://inria.hal.science/hal-04077552 (visited on 11/03/2025).

[19] Pierre-Yves Strub et al. "Self-certification: bootstrapping certified typecheckers in F* with Coq". In: *SIGPLAN Not.* 47.1 (Jan. 25, 2012), pp. 571–584. ISSN: 0362-1340. DOI: 10.1145/2103621.2103723. URL: https://dl.acm.org/doi/10.1145/2103621.2103723 (visited on 10/27/2025).

[20] Junyoung Jang et al. "McTT: A Verified Kernel for a Proof Assistant". In: *Verified Implementation of "McTT: A Verified Kernel for a Proof Assistant"* 9 (ICFP Aug. 5, 2025), 242:190–242:221. DOI: 10.1145/3747511. URL: https://dl.acm.org/doi/10.1145/3747511 (visited on 11/03/2025).

[21] Richard A. Eisenberg. "Stitch: the sound type-indexed type checker (functional pearl)". In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. Haskell 2020. New York, NY, USA: Association for Computing Machinery, Aug. 9, 2020, pp. 39–53. ISBN: 978-1-4503-8050-8. DOI: 10.1145/3406088.3409015. URL: https://dl.acm.org/doi/10.1145/3406088.3409015 (visited on 01/22/2026).

[22] Ramana Kumar et al. "CakeML: a verified implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: Association for Computing Machinery, Jan. 8, 2014, pp. 179–191. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535841. URL: https://dl.acm.org/doi/10.1145/2535838.2535841 (visited on 11/03/2025).

[23] Peter Smith. *An Introduction to Gödel's Theorems*. Cambridge University Press, Feb. 21, 2013. 405 pp. ISBN: 978-1-107-02284-3.

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

[25] Meven Lennon-Bertrand. "À bas l'η - Coq's troublesome η-conversion". 2022. URL: https://www.meven.ac/documents/22-WITS-abstract.pdf.

[26] *Rewrite rules and eta-expansion of the Unit type · Issue #5174 · agda/agda*. GitHub. URL: https://github.com/agda/agda/issues/5174 (visited on 05/06/2026).

[27] *Comparison of blocked terms doesn't respect eta · Issue #3785 · agda/agda*. GitHub. URL: https://github.com/agda/agda/issues/3785 (visited on 05/06/2026).

[28] *Forward declaration breaks eta for unit type · Issue #6417 · agda/agda*. GitHub. URL: https://github.com/agda/agda/issues/6417 (visited on 05/06/2026).

[29] *Occurs check does not properly handle singleton type · Issue #5837 · agda/agda*. GitHub. URL: https://github.com/agda/agda/issues/5837 (visited on 05/06/2026).

[30] András Kovács. "Eta conversion for the unit type (is still not that simple) (WITS 2025) - POPL 2025". Denver, Jan. 25, 2025. URL: https://popl25.sigplan.org/details/wits-2025-papers/4/Eta-conversion-for-the-unit-type-is-still-not-that-simple- (visited on 01/19/2026).

[31] Meven Lennon-Bertrand. "What Does It Take to Certify a Conversion Checker?" In: *LIPIcs, Volume 337, FSCD 2025* 337 (2025). Ed. by Maribel Fernández. Artwork Size: 23 pages, 838028 bytes ISBN: 9783959773744 Medium: application/pdf, 27:1–27:23. ISSN: 1868-8969. DOI: 10.4230/LIPICS.FSCD.2025.27. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2025.27 (visited on 11/03/2025).

[32] *jespercockx/agda-core at atejan_msc_thesis*. June 25, 2026. URL: https://github.com/jespercockx/agda-core/tree/atejan_msc_thesis (visited on 06/25/2026).

[33] *jespercockx/agda-core at atejan_msc_thesis_eta_unit*. June 25, 2026. URL: https://github.com/jespercockx/agda-core/tree/atejan_msc_thesis_eta_unit (visited on 06/25/2026).

[34] *jespercockx/scope@e12c62a*. June 25, 2026. URL: https://github.com/jespercockx/scope/commit/e12c62a932d649cd452d99a67a18f39f179aa1d2 (visited on 06/25/2026).

[35] *Peano numbers - HaskellWiki*. URL: https://wiki.haskell.org/Peano_numbers (visited on 05/14/2026).

[36] *Data Types — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/language/data-types.html (visited on 04/24/2026).

[37] Robin Adams. "Formalized Metatheory with Terms Represented by an Indexed Family of Types". In: *Types for Proofs and Programs*. Ed. by Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner. Berlin, Heidelberg: Springer, 2006, pp. 1–16. ISBN: 978-3-540-31429-5. DOI: 10.1007/11617990_1.

[38] Richard S. Bird and Ross Paterson. "de Bruijn notation as a nested datatype". In: *Journal of Functional Programming* 9.1 (Jan. 1999), pp. 77–91. ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796899003366. URL: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/de-bruijn-notation-as-a-nested-datatype/D8BFA383FDA7EA3DC443B4C42A168F30 (visited on 04/24/2026).

[39] Conor McBride. "I Got Plenty o' Nuttin'". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley et al. Cham: Springer International Publishing, 2016, pp. 207–233. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_12. URL: https://doi.org/10.1007/978-3-319-30936-1_12 (visited on 04/30/2026).

[40] Jesper Cockx et al. "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs". In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. ACM Conferences. Sept. 6, 2022, pp. 108–122. ISBN: 978-1-4503-9438-3. DOI: 10.1145/3546189.3549920. URL: https://dl.acm.org/doi/abs/10.1145/3546189.3549920 (visited on 04/24/2026).

[41] *Compilers — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/tools/compilers.html#ghc-backend (visited on 05/12/2026).

[42] *jespercockx/scope: An agda2hs-compatible library for well-scoped syntax*. URL: https://github.com/jespercockx/scope (visited on 04/25/2026).

[43] *Run-time Irrelevance — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/language/runtime-irrelevance.html (visited on 04/30/2026).

[44] Nils Anders Danielsson. *Logical properties of a modality for erasure*. 2021.

[45] Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. 1972. URL: https://girard.perso.math.cnrs.fr/These.pdf.

[46] *Sort System — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/language/sort-system.html (visited on 04/30/2026).

[47] *Telescopes — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/language/telescopes.html#telescopes.

[48] *Monad - HaskellWiki*. URL: https://wiki.haskell.org/Monad# (visited on 06/07/2026).

[49] Jana Dunfield et al. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 23, 2021), pp. 1–38. DOI: 10.1145/3450952. URL: https://dl.acm.org/doi/10.1145/3450952 (visited on 04/26/2026).

[50] William Pugh et al. "Local type inference". In: *ACM Transactions on Programming Languages and Systems* 22.1 (Jan. 2000), pp. 1–44. DOI: 10.1145/345099.345100. URL: https://dl.acm.org/doi/10.1145/345099.345100 (visited on 04/26/2026).

[51] *Agda.TypeChecking.Monad.Base*. URL: https://hackage-content.haskell.org/package/Agda-2.8.0/docs/Agda-TypeChecking-Monad-Base.html (visited on 04/28/2026).

[52] *Built-ins — Agda 2.8.0 documentation*. URL: https://agda.readthedocs.io/en/v2.8.0/language/built-ins.html#the-unit-type (visited on 05/06/2026).

[53] *Propositions*. URL: https://lean-lang.org/doc/reference/latest/The-Type-System/Propositions/#propositions (visited on 06/02/2026).

[54] *Propositional Equality*. URL: https://lean-lang.org/doc/reference/latest/Basic-Propositions/Propositional-Equality/#K (visited on 06/02/2026).

[55] Rishikesh Vaishnav. "Lean4Less: Eliminating Definitional Equalities from Lean via an Extensional-to-Intensional Translation". In: *ICTAC 2025 - International Colloquium on Theoretical Aspects of Computing* (Nov. 2025).

[56] Arthur Adjedj et al. "Martin-Löf à la Coq". In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2024. New York, NY, USA: Association for Computing Machinery, Jan. 9, 2024, pp. 230–245. ISBN: 979-8-4007-0488-8. DOI: 10.1145/3636501.3636951. URL: https://dl.acm.org/doi/10.1145/3636501.3636951 (visited on 12/02/2025).

[57] *Removing eta equivalence from F* by nikswamy · Pull Request #2294 · FStarLang/FStar*. GitHub. URL: https://github.com/FStarLang/FStar/pull/2294 (visited on 06/02/2026).

[58] *FStarLang/FStar: A Proof-oriented Programming Language*. GitHub. URL: https://github.com/FStarLang/FStar/ (visited on 06/02/2026).

[59] *Issues · jespercockx/agda-core*. GitHub. 2026. URL: https://github.com/jespercockx/agda-core/issues (visited on 05/09/2026).

Acronyms

CbC Correct-by-Construction

TCB Trusted code base

TTB Trusted theory base

MLTT Martin-Löf type theory