Algorithmic Conversion with Surjective Pairing: A Syntactic and Untyped Approach

YIYUN LIU, University of Pennsylvania, USA
STEPHANIE WEIRICH, University of Pennsylvania, USA

In a dependent type theory with β-equivalence as its equational theory, the confluence of untyped reduction and normalization immediately give us a proof of the decidability of type conversion, where the decision procedure for convertibility simply checks the equality of the β-normal forms of its inputs. This technique is not available in the presence of surjective pairing (i.e. the η-law for pairs) because $\beta\eta$-reduction is not confluent. In this work, we show that by adopting established syntactic techniques, we can resolve the issue with confluence caused by surjective pairing, and recover a confluence-based proof of decidability of type conversion. Compared to existing proof developments, which rely on semantic tools such as Kripke-style logical relations, our proof modularly composes a minimal semantic proof of untyped normalization and a syntactic proof of decidability. This modularity enables us to explore algorithmic conversion through syntactic methods without modifying the minimal semantic proof. We have fully mechanized our results using the Rocq theorem prover.

1 Introduction

A prerequisite to implementing a type checker for a dependent type theory is a decision procedure for type conversion. However, finding such an algorithm is nontrivial; the type theories' specification of definitional equality often does not directly induce an algorithm, due to the rules such as symmetry and transitivity. Furthermore, not only do we want an algorithm, but we also want to know that it is correct, *i.e.* we want to prove that it is sound and complete with respect to definitional equality. By mechanizing such a correctness proof in a proof assistant, we can extract a certified conversion routine and have the highest level of assurance in our implementation.

Designers and implementors of existing type theories take various approaches to this problem. When definitional equality is defined as *untyped* β-equivalence, the confluence of untyped β-reduction gives us a reliable formula for both finding algorithms and reasoning about them. To test the β-equivalence of two terms, confluence says it suffices to find a common term that both inputs normalize to through any reduction strategy. As a result, we obtain a class of algorithms that *reduce* their inputs to β-normal forms and then *compare* for α-equivalence. The completeness and soundness of these algorithms follows directly from confluence. To prove that these algorithms terminate, and thus that conversion is decidable, we can use a logical predicate [Girard et al. 1989] to show that all well-typed terms are normalizing under any reduction strategy.

We can also use the confluence-based method for type theories that use *typed* β-equivalence for conversion. Adams [2006] and Siles and Herbelin [2012] show the equivalence between pure type systems [Barendregt 1991] that use untyped and typed β-equality. Thus, we can transport decidability results for a system with untyped definitional equality to its equivalent system with typed definitional equality.

However, challenges arise when definitional equality includes of η-laws, especially the pair η-law, also known as *surjective pairing*. While confluence holds for Curry-style systems that include the function η-law, surjective pairing breaks confluence for both Curry and Church-style systems [Barendregt 1993; Geuvers 1993; Klop and de Vrijer 1989; Lennon-Bertrand 2022]. Without confluence, we are unable to justify the correctness of the reduce-and-compare algorithm.

Authors' Contact Information: Yiyun Liu, University of Pennsylvania, Philadelphia, USA, liuyiyun@seas.upenn.edu; Stephanie Weirich, University of Pennsylvania, Philadelphia, USA, sweirich@seas.upenn.edu.

Furthermore, η-reduction for pairs is not type preserving [Abel and Coquand 2007], making it difficult to show the equivalence between systems with typed $\beta\eta$-equality and untyped $\beta\eta$-equality. Because typed definitional equality only relates well-typed terms, it is more restrictive than untyped equality. In particular, in the transitivity rule, to derive $A = C$ from $A = B$ and $B = C$, typed equality also requires that the intermediate type B be well-typed. However, without type preservation, it is not obvious how the untyped algorithm that compares $\beta\eta$-normal forms is sound with respect to a typed definitional equality, as the $\beta\eta$-reduction sequence to the normal form may contain ill-typed terms.

These technical issues are unfortunate because the pair η-law is important for the expressiveness of the type system! In theorem provers, the η-law for pairs generalizes to the η-laws for non-empty dependent records, allowing us to type-check more terms.

In this work, we resolve these issues and show that reduce-and-compare algorithms with η-laws for functions and pairs correctly implement type conversion. Our key observation is that $\beta\eta$-reduction with the function η-law and surjective pairing is confluent on an inductively defined set of strongly normalizing terms (originally defined by Van Raamsdonk and Severi [1995] and denoted as **SN**). By applying the *SN*-method of Joachimski and Matthes [2003] to formulate our logical predicate, we can prove that all well-typed terms are in the **SN** set, allowing us to directly show the completeness of reduce-and-compare algorithms.

To show the soundness of any reduce-and-compare algorithm, we adopt the syntactic technique of Goguen [2005] and introduce Coquand's untyped conversion algorithm [Coquand 1991] as an intermediate step. Coquand's algorithm performs η-expansion based on the shape of the terms being compared. The shape-based η-expansion is type-preserving and therefore enables us to give a direct proof of its soundness with respect to the typed convertibility. By proving the completeness of Coquand's algorithm with respect to the untyped reduce-and-compare algorithm, we fully establish the equivalence between the typed convertibility, Coquand's algorithm, and the untyped reduce-and-compare algorithm.

Why yet another decidability proof of type conversion? To motivate our confluence-based algorithms, we start by reviewing the state-of-the-art proof techniques for the decidability of conversion.

Abel and Coquand [2007] and later efforts by Abel et al. [2017]; Abel and Scherer [2012] and Adjedj et al. [2024], prove the decidability of type conversion by interpreting types as partial equivalence relations (PER). Instead of modeling the definitional equality with untyped $\beta\eta$-reduction, this approach models definitional equality using PERs. The extensionality properties of the PER model, justifies the η-laws for functions and products. This approach sidesteps the need for the confluence, so works in situations where untyped $\beta\eta$-reduction cannot model definitional equality (*e.g.* the inclusion of η-laws for singleton types). However, the power of the PER model comes at a cost. Regardless of whether the algorithm being justified is typed [Abel and Scherer 2012; Harper and Pfenning 2005] or untyped [Abel and Coquand 2007], the PER method can only be carried out on type systems with a *typed* definitional equality. For systems that do not have an equivalent form with a typed definitional equality, such as ICC [Barras and Bernardo 2008; Miquel 2001], DDC [Choudhury et al. 2022], and DCOI [Liu et al. 2024b], this approach is not applicable.

Coquand [2019], Altenkirch and Kaposi [2016], and the more recent synthetic method by Sterling and Angiuli [2021] prove the decidability of type conversion by working with an algebraic structure where syntactic terms are quotiented by definitional equality. The algebraic approach completely eschews the notion of reduction and proves the correctness of the normalization by evaluation (NbE) algorithm [Berger et al. 1998] without reasoning about reduction at all. Similar to the PER approach, the algebraic reduction-free approach does not rely on the confluence of $\beta\eta$-reduction, but shares the limitation that the object language must have a typed definitional equality. While

the reduction-free approach leads to a simple proof on paper, it remains difficult to carry it out in existing intensional theorem provers [Adjedj et al. 2024].

Despite the many advantages of the PER method and the algebraic reduction-free method, we believe the confluence-based proof is appealing for the following reasons. First, we can directly apply the confluence-based method to a system with an untyped definitional equality without taking a detour to typed definitional equality. Avoiding such detours not only leads to simpler proofs, but is also necessary for systems that are not known to be compatible with typed equality.

Second, because the confluence result is untyped, we can reuse it across multiple different type systems. The presence of expressive type system features such as large eliminations, subtyping, or inductive datatypes is independent of our methodology.

Finally, the confluence-based method leads to a modular proof consisting of a minimal, localized semantic proof of normalization, and a purely syntactic proof of decidability parameterized by confluence and normalization. The clean decoupling between the semantic and syntactic arguments allows us to mechanize more of the decidability proof in a weak metatheory by assuming only normalization [Sozeau et al. 2019].

We have mechanized all of the results of this paper using the Rocq proof assistant [Team 2024] and the proof development is available in the supplementary material under the `source` subdirectory. By leveraging the method of encoding general recursion by Bove and Capretta [2005], we can extract the verified conversion checker as an executable OCaml program.

The structure of our proof scripts supports our claim about its modularity and reusability. The entire development has approximately 10,000 LoC. The logical predicate, which is used to prove normalization, consists of 1,500 LoC (15% of the total). In contrast, the logical relations found in both Abel et al. [2017] and Wieczorek and Biernacki [2018] take up more than 50% of their respective code bases. Our semantic proofs are therefore significantly more lightweight. Our proof for $\beta\eta$-confluence and various other untyped injectivity properties, all of which are agnostic to the type system, consists of 5,000 LoC (50%). The remaining proofs are syntactic metatheoretic results of the type theory and syntactic proofs that relate Coquand's algorithm to $\beta\eta$-reduction. Experimenting with new variations of the conversion algorithm would thus only require changes to the remaining 3,500 lines of syntactic proof that are specific to the algorithm.

Our contributions. In this work, we analyze an expressive dependent type theory, called $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$, that features large eliminations, a cumulative universe hierarchy with subtyping, and a typed convertibility relation with η-laws for functions and dependent pairs. Section 2 presents the syntax and typing specification of our object language and its basic syntactic results.

In Section 3, we prove the confluence of $\beta\eta$-reduction for $\mathbf{SN}$ and apply the SN-method in a dependently typed setting to prove that every well-typed term is in $\mathbf{SN}$ using an untyped logical predicate. The confluence result over $\mathbf{SN}$ allows us to model convertibility with untyped $\beta\eta$-reduction and conclude the completeness of the reduce-and-compare algorithm with respect to the definitional subtyping relation.

In Section 4, we introduce Coquand's algorithm for type conversion extended with subtyping and surjective pairing as bridge between the untyped reduce-and-compare algorithm and typed definitional subtyping. Leveraging the fact that the η-expansion performed in Coquand's algorithm is type-preserving, we prove the soundness of Coquand's algorithm with respect to the typed definitional subtyping. We prove the completeness of Coquand's algorithm with respect to untyped $\beta\eta$-equivalence using a termination metric adapted from Goguen [2005]. We are thus able to conclude our decidability proof by showing that Coquand's algorithm and the reduce-and-compare algorithm (with well-typed inputs) are both equivalent to the typed convertibility relation.

While our object language, $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$, mostly resembles the core languages of intensional type theories, there are a few subtleties in its design, which we discuss in Section 5. In particular, we include type constructor injectivity rules as part of the subtyping relation (Section 5.1), and use of Curry-style lambda terms (Section 5.2). We also deepen our comparison to related systems in Section 6 and conclude in Section 7.

2 Specification of $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$

$$
\begin{array}{rll}
\Gamma & ::= \cdot \mid \Gamma, x : A & \text{Typing contexts} \\
A, B, C, a, b, c, P & ::= & \text{Terms} \\
& \mid x \mid \mathcal{U}_i & \textit{variables, type universes} \\
& \mid \Pi x{:}A.\, B \mid \lambda x.\, b \mid b\ a & \textit{function types, abstractions, applications} \\
& \mid \Sigma x{:}A.\, B \mid (a, b) \mid \pi_1\ x \mid \pi_2\ x & \textit{pair types, pairs, projections}
\end{array}
$$

Fig. 1. Grammar, excluding natural numbers

$\boxed{\vdash \Gamma}$ *(Context well-formedness)*

$$\textsc{Wf-Nil}\ \frac{}{\vdash \cdot} \qquad \textsc{Wf-Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathcal{U}_i}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash a : A}$ *(Typing)*

$$\textsc{Wt-Var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Wt-Pi}\ \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \Pi x{:}A.\, B : \mathcal{U}_{i \vee j}} \qquad \textsc{Wt-Abs}\ \frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.\, b : \Pi x{:}A.\, B} \qquad \textsc{Wt-App}\ \frac{\Gamma \vdash b : \Pi x{:}A.\, B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B[a/x]}$$

$$\textsc{Wt-Univ}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \qquad \textsc{Wt-Sig}\ \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \Sigma x{:}A.\, B : \mathcal{U}_{i \vee j}} \qquad \textsc{Wt-Pair}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x] \qquad \Gamma \vdash \Sigma x{:}A.\, B : \mathcal{U}_i}{\Gamma \vdash (a, b) : \Sigma x{:}A.\, B} \qquad \textsc{Wt-Proj1}\ \frac{\Gamma \vdash a : \Sigma x{:}A.\, B}{\Gamma \vdash \pi_1\ a : A}$$

$$\textsc{Wt-Proj2}\ \frac{\Gamma \vdash a : \Sigma x{:}A.\, B}{\Gamma \vdash \pi_2\ a : B[\pi_1\ a/x]} \qquad \textsc{Wt-Conv}\ \frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash a : B}$$

Fig. 2. Typing rules

We start by presenting $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$, our dependently typed object language, including its grammar and typing rules (Figures 1 to 3). These rules specify *what* programs type check, but they do not directly describe a type checking algorithm.

The language includes functions ($\lambda x.\, a$), dependent pairs (a, b) with projection operators ($\pi_1\ a$ and $\pi_2\ a$), and a Russell-style infinite universe hierarchy ($\mathcal{U}_i$). Function abstractions do not include type annotations; we discuss in Section 5.2 our proofs can be easily adapted to a Church-style system.

The language in our mechanized proofs also includes natural numbers with an induction principle, but we omit these forms from the text. We include natural numbers in our development to provide an observable infinite data type so our language supports large eliminations. Furthermore, natural

$\boxed{\Gamma \vdash a = b : A}$ *(Equality)*

$$\textsc{E-Refl}\ \frac{\Gamma \vdash a : A}{\Gamma \vdash a = a : A} \qquad \textsc{E-Sym}\ \frac{\Gamma \vdash b = a : A}{\Gamma \vdash a = b : A} \qquad \textsc{E-Trans}\ \frac{\Gamma \vdash a = b : A \quad \Gamma \vdash b = c : A}{\Gamma \vdash a = c : A} \qquad \textsc{E-AbsExt}\ \frac{x \notin \Gamma \quad \Gamma, x : A \vdash a\,x = b\,x : B}{\Gamma \vdash a = b : \Pi x{:}A.\, B}$$

$$\textsc{E-PairExt}\ \frac{\Gamma \vdash \pi_1\ a = \pi_1\ b : A \quad \Gamma \vdash \pi_2\ a = \pi_2\ b : B[\pi_1\ a/x]}{\Gamma \vdash a = b : \Sigma x{:}A.\, B} \qquad \textsc{E-AppAbs}\ \frac{\Gamma \vdash b : A \quad \Gamma, x : A \vdash a : B}{\Gamma \vdash (\lambda x.\ a)\ b = a[b/x] : B[b/x]}$$

$$\textsc{E-ProjPair1}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \pi_1\ (a, b) = a : A} \qquad \textsc{E-ProjPair2}\ \frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[a/x]}{\Gamma \vdash \pi_2\ (a, b) = b : B[a/x]}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{L-Trans}\ \frac{\Gamma \vdash A \leq B \quad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \qquad \textsc{L-Eq}\ \frac{\Gamma \vdash A = B : \mathcal{U}_i}{\Gamma \vdash A \leq B} \qquad \textsc{L-Univ}\ \frac{\vdash \Gamma \quad i \leq j}{\Gamma \vdash \mathcal{U}_i \leq \mathcal{U}_j} \qquad \textsc{L-Pi}\ \frac{\Gamma \vdash A_1 \leq A_0 \quad \Gamma, x : A_0 \vdash B_0 \leq B_1}{\Gamma \vdash \Pi x{:}A_0.\, B_0 \leq \Pi x{:}A_1.\, B_1}$$

$$\textsc{L-Sig}\ \frac{\Gamma \vdash A_0 \leq A_1 \quad \Gamma, x : A_0 \vdash B_0 \leq B_1}{\Gamma \vdash \Sigma x{:}A_0.\, B_0 \leq \Sigma x{:}A_1.\, B_1} \qquad \textsc{L-PiProj1}\ \frac{\Gamma \vdash \Pi x{:}A_0.\, B_0 \leq \Pi x{:}A_1.\, B_1}{\Gamma \vdash A_1 \leq A_0} \qquad \textsc{L-PiProj2}\ \frac{\Gamma \vdash \Pi x{:}A_0.\, B_0 \leq \Pi x{:}A_1.\, B_1 \quad \Gamma \vdash a_0 = a_1 : A_1}{\Gamma \vdash B_0[a_0/x] \leq B_1[a_1/x]}$$

Fig. 3. Equality and subtyping rules (selected)

numbers help us ensure that our proof techniques are robust under the addition of positive types (i.e. types with pattern-matching as their elimination forms). This addition leads to no surprises and requires no special treatment during our development. The full set of rules that includes natural numbers can be found in the supplementary material.

The $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$type system includes subtyping. Rule Wt-Conv refers to the definitional subtyping relation $\Gamma \vdash A \leq B$, which is defined in Figure 3 along with the $\beta\eta$-rules for the definitional equality $\Gamma \vdash a = b : A$. The rules for equality are standard, except for the addition of the injectivity rules L-PiProjOne and L-PiProjTwo and the omitted counterpart for Σ types. These rules simplify our proofs and do not affect decidability; we discuss them further in Section 5.1.

This type system satisfies standard syntactic properties, such as weakening and substitution (not shown). In addition, our mechanization includes proofs of the following. We can recover the well-formedness of context from the mutually defined typing and conversion judgments.

Lemma 2.1 (Context regularity). *If* $\Gamma \vdash a : A$, $\Gamma \vdash a = b : A$, *or* $\Gamma \vdash A \leq B$, *then* $\vdash \Gamma$.

In the presence of rule Wt-Conv, the generation lemma is useful for recovering information about a typing derivation.

Lemma 2.2 (Generation (selected)).

- *If* $\Gamma \vdash \lambda x.\ a : C$, *then there exists some* A *and* B *such that* $\Gamma \vdash \Pi x{:}A.\, B \leq C$ *and* $\Gamma, x : A \vdash a : B$.

- *If* $\Gamma \vdash b\,a : C$, *then there exists some* A *and* B *such that* $\Gamma \vdash b : \Pi x : A.\, B$, $\Gamma \vdash a : A$ *and* $\Gamma \vdash B[a/x] \leq C$.
- *If* $\Gamma \vdash \Pi x : A.\, B : C$, *then there exists some* i *such that* $\Gamma \vdash A : \mathcal{U}_i$, $\Gamma, x : A \vdash B : \mathcal{U}_i$, *and* $\Gamma \vdash \Pi x : A.\, B : \mathcal{U}_i$ *and* $\Gamma \vdash \mathcal{U}_i \leq C$.
- *If* $\Gamma \vdash \mathcal{U}_i : C$, *then* $\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}$ *and* $\Gamma \vdash \mathcal{U}_{i+1} \leq C$.

Using the substitution lemma and rules L-PiProjOne and L-PiProjTwo, we can prove that β-reduction preserves typing. We defer the precise definition of β-reduction to the next section.

Lemma 2.3 (Subject Reduction). *If* $\Gamma \vdash a : A$ *and* $a \rightsquigarrow_\beta b$, *then* $\Gamma \vdash b : B$.

Finally, regularity states that if a term is well-typed, then its type must be well-formed.

Lemma 2.4 (Regularity).

- *If* $\Gamma \vdash a : A$, *then there exists* i *such that* $\Gamma \vdash A : \mathcal{U}_i$.
- *If* $\Gamma \vdash a = b : A$, *then* $\Gamma \vdash a : A$ *and* $\Gamma \vdash b : A$.
- *If* $\Gamma \vdash A \leq B$, *then there exists* i *such that* $\Gamma \vdash A : \mathcal{U}_i$ *and* $\Gamma \vdash B : \mathcal{U}_i$.

3 Confluence of $\beta\eta$-Reduction for Strong Normalizing Terms

Our goal in this section is to show that we can implement the subtyping relation $\Gamma \vdash A \leq B$. A key step in this process is to model the typed convertibility relation $\Gamma \vdash a = b : A$ with an untyped reduce-and-compare algorithm defined in terms of $\beta\eta$-reduction. This algorithm, which we introduce in Section 3.6, is not transitive by definition, a necessary property if we are to use it to model the transitivity rule of typed convertibility. Instead, we use the confluence of $\beta\eta$-reduction to show that our algorithmic relation is transitive.

However, the usual $\beta\eta$-reduction relation is not confluent, due to surjective pairing. (See Klop [1980] for a counterexample). We solve this issue by identifying a set of terms where confluence holds—an inductively characterized set, called SN, that contains only strongly normalizing terms [Van Raamsdonk and Severi 1995]—and then arguing that all terms that we care about are in this set. Following Joachimski and Matthes [2003], we use an untyped logical predicate to show that all well-typed terms are in SN. Because we only invoke the conversion algorithm on well-typed terms, the confluence result for SN is sufficient to justify the transitivity of the reduce-and-compare algorithm.

3.1 Reduction Relations and Normal Forms

$\boxed{a \rhd_\beta b}$ *(Primitive β-Reduction)*

$$\text{PB-AppAbs}\ \frac{}{(\lambda x.\, a)\ b \rhd_\beta a[b/x]} \qquad \text{PB-ProjPair1}\ \frac{}{\pi_1\ (a, b) \rhd_\beta a} \qquad \text{PB-ProjPair2}\ \frac{}{\pi_2\ (a, b) \rhd_\beta b}$$

$\boxed{a \rhd_\eta b}$ *(Primitive η-Reduction)*

$$\text{PE-AppEta}\ \frac{x \notin \mathbf{freevar}(a)}{\lambda x.\, a\, x \rhd_\eta a} \qquad \text{PE-PairEta}\ \frac{}{(\pi_1\ a, \pi_2\ a) \rhd_\eta a}$$

Fig. 4. Primitive β and η-Reductions

Before we introduce SN, we start by reviewing standard definitions that help us both motivate and state properties about this set.

First, in Figure 4, we define primitive β and η-reductions. Rule PE-PairEta is also known as surjective pairing. We refer to a lambda term that can be reduced by one of these primitive reduction rules as a β or η-redex respectively.

We use $\rightsquigarrow$ to denote the strong/full reduction relation, which can reduce anywhere in the term, including under constructor forms, and $\Rightarrow$ to denote its parallel variant [Barendregt 1993; Takahashi 1995]. To be specific about which primitive rules may participate in full reduction, we use subscripts. Thus, the relations $\rightsquigarrow_\beta$, $\rightsquigarrow_\eta$, and $\rightsquigarrow_{\beta\eta}$ denote the full β, η, and $\beta\eta$-reduction relations. We also use $\rightsquigarrow_{lo}$ to denote the leftmost-outermost β-reduction strategy and $\rightsquigarrow_h$ as its corresponding weak reduction variant, which does not reduce under constructors. Note that our proofs do not require versions of these relations that include η-reduction, so we omit the β-subscript.

Neutral forms (**ne**)	v	::=	$x \mid v\ u \mid \pi_1\ v \mid \pi_2\ v$
Normal forms (**nf**)	u	::=	$\Pi x{:}u.\ u \mid \Sigma x{:}u.\ u \mid \lambda x.\ u \mid (u, u) \mid \mathcal{U}_i$
Weak-head neutral forms (**whne**)	e	::=	$x \mid e\ a \mid \pi_1\ e \mid \pi_2\ e$
Canonical forms (**canf**)	h	::=	$\Pi x{:}A.\ B \mid \Sigma x{:}A.\ B \mid \lambda x.\ a \mid (a, b) \mid \mathcal{U}_i$
Weak-head normal forms (**whnf**)	f	::=	$e \mid h$

Fig. 5. Grammars for neutral and normal forms

Figure 5 defines β-neutral (v) and β-normal (u) forms and their weak-head counterparts. An expression that is in weak-head normal form but is not neutral is called *canonical*. It is straightforward to verify that neither neutral nor normal forms contain any β-redexes and that the η-reduction relation preserves β-normal forms.

Lemma 3.1. *If $a \in$ **nf** or $a \in$ **ne**, then $a \not\rightsquigarrow_\beta b$.*

Lemma 3.2. *If $u \rightsquigarrow_\eta a$, then $a \in$ **nf**. If $v \rightsquigarrow_\eta a$, then $a \in$ **ne**.*

Note that the converse of Lemma 3.1 is not true—the inductive definitions of neutral and normal forms rule out stuck terms. For examples, the terms $\pi_1\ (\lambda x.\ x)$ or $(x, y)\ z$ are unable to take any further β-steps but are *not* in normal form as defined by Figure 5.

3.2 An Inductive Characterization of Strongly Normalizing Terms

Following Van Raamsdonk and Severi [1995], we define the set **SN** of strongly normalizing terms mutually with the set of strongly neutral terms **SNe** and the relation $a \rightsquigarrow_{\text{SN}} b$, a subrelation of the weak-head reduction relation $a \rightsquigarrow_h b$. Ignoring N-Exp, the sets **SN** and **SNe** are exactly the sets of normal and neutral forms from Figure 5. By adding N-Exp, we can expand the sets to include terms that contain β-redexes but nevertheless reduce to a normal or neutral forms.

In rule N-AppAbs, the fact that $a[b/x]$ is strongly normalizing alone does not necessarily imply that $(\lambda x.\ a)\ b$ is also strongly normalizing. We can construct a counterexample by picking b to be an infinite loop and a an expression that does not contain the variable x. Thus, the $b \in$ **SN** constraint in N-AppAbs ensures that if the term b is strongly normalizing and $a \rightsquigarrow_{\text{SN}} b$, then a must also be strongly normalizing.

By forgetting the **SN** and **SNe** premises in $\rightsquigarrow_{\text{SN}}$, we can recover from each of these judgments a leftmost-outermost reduction ($\rightsquigarrow_{lo}$) sequence to normal or neutral forms.

Lemma 3.3 (SN leftmost-outermost reduction).

- *If $a \in$ **SN**, then there exists u such that $a \rightsquigarrow_{lo}^* u$.*
- *If $a \in$ **SNe**, then there exists v such that $a \rightsquigarrow_{lo}^* v$.*
- *If $a \rightsquigarrow_{\text{SN}} b$, then $a \rightsquigarrow_h b$.*

$\boxed{a \in \mathbf{SN}}$ *(Strong Normal Forms)*

$$\textsc{N-Abs}\ \frac{b \in \mathbf{SN}}{\lambda x.\, b \in \mathbf{SN}} \qquad \textsc{N-Pair}\ \frac{a \in \mathbf{SN} \qquad b \in \mathbf{SN}}{(a, b) \in \mathbf{SN}} \qquad \textsc{N-Pi}\ \frac{A \in \mathbf{SN} \qquad B \in \mathbf{SN}}{\Pi x{:}A.\, B \in \mathbf{SN}} \qquad \textsc{N-Sig}\ \frac{A \in \mathbf{SN} \qquad B \in \mathbf{SN}}{\Sigma x{:}A.\, B \in \mathbf{SN}}$$

$$\textsc{N-Univ}\ \frac{}{\mathcal{U}_i \in \mathbf{SN}} \qquad \textsc{N-SNe}\ \frac{a \in \mathbf{SNe}}{a \in \mathbf{SN}} \qquad \textsc{N-Exp}\ \frac{a \rightsquigarrow_{\mathbf{SN}} b \qquad b \in \mathbf{SN}}{a \in \mathbf{SN}}$$

$\boxed{a \in \mathbf{SNe}}$ *(Strong Neutral Forms)*

$$\textsc{N-Var}\ \frac{}{x \in \mathbf{SNe}} \qquad \textsc{N-App}\ \frac{a \in \mathbf{SNe} \qquad b \in \mathbf{SN}}{a\ b \in \mathbf{SNe}} \qquad \textsc{N-Proj1}\ \frac{a \in \mathbf{SNe}}{\pi_1\ a \in \mathbf{SNe}} \qquad \textsc{N-Proj2}\ \frac{b \in \mathbf{SNe}}{\pi_2\ a \in \mathbf{SNe}}$$

$\boxed{a \rightsquigarrow_{\mathbf{SN}} b}$ *(Strong Weak Head Reduction)*

$$\textsc{N-AppAbs}\ \frac{b \in \mathbf{SN}}{(\lambda x.\, a)\ b \rightsquigarrow_{\mathbf{SN}} a[b/x]} \qquad \textsc{N-AppCong}\ \frac{b \in \mathbf{SN} \qquad a_0 \rightsquigarrow_{\mathbf{SN}} a_1}{a_0\ b \rightsquigarrow_{\mathbf{SN}} a_1\ b} \qquad \textsc{N-ProjPair1}\ \frac{b \in \mathbf{SN}}{\pi_1\ (a, b) \rightsquigarrow_{\mathbf{SN}} a} \qquad \textsc{N-ProjPair2}\ \frac{a \in \mathbf{SN}}{\pi_2\ (a, b) \rightsquigarrow_{\mathbf{SN}} b}$$

$$\textsc{N-ProjCong1}\ \frac{a \rightsquigarrow_{\mathbf{SN}} b}{\pi_1\ a \rightsquigarrow_{\mathbf{SN}} \pi_1\ b} \qquad \textsc{N-ProjCong2}\ \frac{a \rightsquigarrow_{\mathbf{SN}} b}{\pi_2\ a \rightsquigarrow_{\mathbf{SN}} \pi_2\ b}$$

In the following, we prove that $\beta\eta$-reduction is confluent by using η-postponement [Barendregt 1993; Takahashi 1995]. A sequence of $\beta\eta$-reductions starting from an SN term can be factorized into a sequence of β-reductions followed by η-reductions.

An alternative proof of confluence is to use Newman's lemma with the fact that $\beta\eta$-reduction is locally confluent. A perhaps surprising result, shown by Joachimski and Matthes [2003], is that despite the absence of η-rules in the definition of $\rightsquigarrow_{\mathbf{SN}}$, the set SN also characterizes the set of strongly $\beta\eta$-normalizing terms. However, the proof that terms in SN are strongly $\beta\eta$-normalizing, as required by Newman's lemma, is tedious. It requires nested induction and repeated analysis of possible $\beta\eta$-redexes to show that the set of strongly $\beta\eta$-normalizing terms satisfies the same congruence rules as SN and SNe.

Our proof based on η-postponement avoids the headache of connecting SN to strong $\beta\eta$-normalization (defined in terms of the well-foundedness of $\rightsquigarrow_{\beta\eta}$). Furthermore, this proof is also more flexible as it allows us to generalize the $\beta\eta$-confluence result to weakly β-normalizing systems as we show in Section 5.3.

3.3 Properties of SN

Next, we show structural properties of SN to prepare for our η-postponement and confluence proof. Like the inductively defined normal and neutral forms (Figure 5), the sets SN and SNe do not contain stuck terms.

Lemma 3.4 (No Stuck Terms). *There is no terms a, b, and c such that $(a, b)\ c \in \mathbf{SN}$, $\pi_1\ (\lambda x.\, a) \in \mathbf{SN}$, or $\pi_2\ (\lambda x.\, a) \in \mathbf{SN}$.*

Proof. Immediate by case analysis over the derivation of SN. □

The inductive characterization of strong normalization satisfies the structural properties of renaming and inversion (not shown), as well as the antisubstitution property below.

Lemma 3.5 (SN antisubstitution).

- *If* $a[b/x] \in \mathbf{SN}$, *then* $a \in \mathbf{SN}$.
- *If* $a[b/x] \in \mathbf{SNe}$, *then* $a \in \mathbf{SNe}$.
- *If* $a[c/x] \leadsto_{\mathrm{SN}} b$, *then either* $a \in \mathbf{SNe}$ *or there exists* b_0 *such that* $a \leadsto_{\mathrm{SN}} b_0$ *and* $b = b_0[c/x]$.

The structural properties are proven by structural induction over the respective derivation except for the application case of inversion that requires Lemma 3.5.

Strong normalization is preserved by both β-reduction and η-reduction. However, because **SN** and **SNe** are mutually defined with the reduction relation $a \leadsto_{\mathrm{SN}} b$, we must simultaneously show that $\beta\eta$-reduction commutes with $a \leadsto_{\mathrm{SN}} b$. This proof of commutativity requires us to generalize the lemma statement to use parallel $\beta\eta$-reduction. Furthermore, we use $a \leadsto_{\overline{\mathrm{SN}}} b$ to denote the reflexive closure of $a \leadsto_{\mathrm{SN}} b$.

Lemma 3.6 (SN Preservation).

- *If* $a \in \mathbf{SN}$ *and* $a \Rightarrow_{\beta\eta} b$, *then* $b \in \mathbf{SN}$.
- *If* $a \in \mathbf{SNe}$ *and* $a \Rightarrow_{\beta\eta} b$, *then* $b \in \mathbf{SNe}$.
- *If* $a \leadsto_{\mathrm{SN}} b_0$ *and* $a \Rightarrow_{\beta\eta} b_1$, *then there exists some* c *such that* $b_0 \Rightarrow_{\beta\eta} c$ *and* $b_1 \leadsto_{\overline{\mathrm{SN}}} c$.

The proof for commutativity (the third case of Lemma 3.6) only works because $a \leadsto_{\mathrm{SN}} b$ is a subrelation of head reduction, which never reduces under constructors. The same commutativity property does not hold if we replace $a \leadsto_{\mathrm{SN}} b$ with full $\beta\eta$-reduction.

3.4 Postponement of η-Reduction for SN

While η-postponement is a classic result for the pure untyped lambda calculus, it does not hold when we add another constructor (such as pairing). A simple counterexample is the expression $\pi_1\ (\lambda x.\ (y, y)\ x)$, which normalizes to y by first η-contracting the lambda term $\lambda x.\ (y, y)\ x$ to (y, y) then β-reducing $\pi_1\ (y, y)$. More generally, the counterexamples involve a stuck term that becomes unstuck by one step of η-reduction. However, because **SN** contains no stuck terms, we can prove η-postponement for terms in **SN** without running into the problematic cases!

The η-postponement theorem that we prove is more precise than the corresponding theorem found in Barendregt [1993] and Takahashi [1995]. We show that every sequence of $\beta\eta$-reductions can be converted into a sequence of β-reductions followed by a sequence of η-reductions such that *the η-reduction sequence does not reduce any β-redexes*. We denote this restrictive version of η-reduction as $a \Rightarrow_{ne} b$, mutually defined with the helper reduction relation $a \Rightarrow_{e} b$ in Figure 6. Both relations are subrelations of parallel η-reduction.

These definitions track whether we are reducing the scrutinee of an elimination form. The relation $a \Rightarrow_{ne} b$ can be used when the term a is not used in this way. Thus, we can freely perform η-contractions through AppEta and PairEta without erasing any β-redex. The $a \Rightarrow_{e} b$ relation, on the other hand, is used for subterms that are being eliminated and therefore is not allowed to perform any η-reduction steps at the top-level since doing so would erase potential β-redexes.

The desired property that $a \Rightarrow_{ne} b$ does not reduce a β-redex can be stated as follows.

Lemma 3.7 (restrictive-η and nf). *If* $a \Rightarrow_{ne} b$ *or* $a \Rightarrow_{e} b$, *then* $b \in \mathbf{nf}$ *implies* $a \in \mathbf{nf}$.

Proof. Immediate by mutual induction over the derivations of $a \Rightarrow_{ne} b$ and $a \Rightarrow_{e} b$. □

The following lemma allows us to decompose one step of parallel η-contraction into a sequence of β-reductions followed by one step of restrictive η-contraction. This property requires the **SN** restriction.

$\boxed{a \Rightarrow_{ne} b}$ *(Restrictive η-reduction (not eliminated))*

$$\textsc{EP-AppEta}\ \frac{a_0 \notin \mathbf{canf} \qquad a_0 \Rightarrow_e a_1 \qquad x \notin \mathbf{freevar}(a)}{(\lambda x.\ a_0\ x) \Rightarrow_{ne} a_1} \qquad \textsc{EP-PairEta}\ \frac{a_0 \notin \mathbf{canf} \qquad a_0 \Rightarrow_e a_1}{(\pi_1\ a_0, \pi_2\ a_0) \Rightarrow_{ne} a_1} \qquad \textsc{EP-Embed}\ \frac{a \Rightarrow_e b}{a \Rightarrow_{ne} b}$$

$\boxed{a \Rightarrow_e b}$ *(eliminated)*

$$\textsc{NEP-AbsCong}\ \frac{a_0 \Rightarrow_{ne} a_1}{\lambda x.\ a_0 \Rightarrow_e \lambda x.\ a_1} \qquad \textsc{NEP-AppCong}\ \frac{a_0 \Rightarrow_e a_1 \qquad b_0 \Rightarrow_{ne} b_1}{a_0\ b_0 \Rightarrow_e a_1\ b_1} \qquad \textsc{NEP-PairCong}\ \frac{a_0 \Rightarrow_{ne} a_1 \qquad b_0 \Rightarrow_{ne} b_1}{(a_0, b_0) \Rightarrow_e (a_1, b_1)}$$

Fig. 6. Definitions of restrictive η-reductions (selected)

Lemma 3.8 (η-Decomposition). *If $a \Rightarrow_\eta b$ and $a \in \mathbf{SN}$, then there exists c such that $a \rightsquigarrow_\beta^* c$ and $c \Rightarrow_{ne} b$.*

Proof. By inducting over the derivation of $a \Rightarrow_\eta b$. Assuming $a \in \mathbf{SN}$, Lemma 3.6 and inversion ensure that every subterm of a or reduced term also belongs to $\mathbf{SN}$. The only interesting cases are EP-AppEta and EP-PairEta, which are similar.

We show the EP-AppEta case to demonstrate how the lemmas defined earlier are used. Suppose $a = \lambda x.\ a_0\ x$ where $x \notin \mathbf{freevar}(a)$ and $a_0 \Rightarrow_\eta b$. By induction, there exists some c_0 such that $a_0 \rightsquigarrow_\beta^* c_0$ and $c_0 \Rightarrow_{ne} b$. Our goal is to find some c such that $\lambda x.\ a_0\ x \rightsquigarrow_\beta^* c$ and $c \Rightarrow_{ne} b$.

From $a_0 \rightsquigarrow_\beta^* c_0$, we deduce that $\lambda x.\ a_0\ x \rightsquigarrow_\beta^* \lambda x.\ c_0\ x$. When c_0 is not in canonical form, then $a_0 \notin \mathbf{canf}$ holds, and we are allowed to use EP-AppEta to finish off the proof by picking $c = \lambda x.\ c_0\ x$. The precondition of EP-AppEta requires $c_0 \Rightarrow_e b$ rather than $c_0 \Rightarrow_{ne} b$, but we know that $a_0 \notin \mathbf{canf}$.

When c_0 is a canonical form, we know that it must take the form $\lambda x.\ c_1$ for some term c_1, since otherwise Lemmas 3.4 and 3.6 derive a contradiction. We can then pick $c = \lambda x.\ c_1$ and construct the reduction sequence $a = \lambda x.\ a_0\ x \rightsquigarrow_\beta^* \lambda x.\ (\lambda x.\ c_1)\ x \rightsquigarrow_\beta \lambda x.\ c_1 = c_0 \Rightarrow_{ne} b$. □

The parallel η-rules for functions and pairs are hard to analyze as they may reduce any finite number of outermost η-redexes. Lemma 3.8 addresses this problem by decomposing a single step of parallel η-reduction into a sequence of β-reduction followed by one step of $\Rightarrow_{ne}$, which can only reduce at most one outermost η-redex. Thus, Lemma 3.8 subsumes the notion of k-fold η-expansion from Takahashi [1995] as a tool to make inverting on the derivation of parallel η-reduction tractable.

Lemma 3.9 (η-postponement). *If $a \in \mathbf{SN}$, $a \Rightarrow_\eta b$, and $b \rightsquigarrow_\beta c$, then there exists some b_0 such that $a \rightsquigarrow_\beta^* b_0$ and $b_0 \Rightarrow_\eta c$.*

Proof. By induction on the derivation of $a \Rightarrow_\eta b$. Most cases are trivial except for the congruence rules for elimination forms, where Lemma 3.8 is required. We only consider the case for rule P-AppCong as the other cases are similar. Suppose $a = a_0\ a_1$ and $b = b_0\ b_1$ such that $a_0 \Rightarrow_\eta b_0$ and $a_1 \Rightarrow_\eta b_1$. We invert on the derivation of $b = b_0\ b_1 \rightsquigarrow_\beta c$ and consider two possible cases. In the first case, $b_0\ b_1$ steps into c through one of the congruence rules for application. The goal is then immediate by induction.

In the second case, we have $b_0\ b_1 = (\lambda x.\ b_2)\ b_1 \rightsquigarrow_\beta b_2[b_1/x] = c$ for some term b_2. Applying Lemma 3.8 to the hypothesis that $a_0 \Rightarrow_\eta b_0 = \lambda x.\ b_2$, we deduce that there exists some term a_2 such that $a_0 \rightsquigarrow_\beta^* a_2$ and $a_2 \Rightarrow_{ne} \lambda x.\ b_2$. The proof can be finished off by inverting over the derivation of $a_2 \Rightarrow_{ne} \lambda x.\ b_2$. Only EP-AppEta and EP-AbsCong could have been used to derive $a_2 \Rightarrow_{ne} \lambda x.\ b_2$

since η-contracting a pair into a lambda term through rule EP-PairEta implies the existence of a stuck term that does not belong to SN (Lemma 3.4). □

Corollary 3.1 (Strengthened η-postponement). *If $a \Rightarrow_\eta b$ and $b \rightsquigarrow_\beta c$, then there exists some b_0 such that $a \rightsquigarrow_\beta^* b_0$ and $b_0 \Rightarrow_{ne} c$.*

Proof. Immediate by composing Lemmas 3.8 and 3.9. □

By composing Lemma 3.7 and Corollary 3.1, we obtain the final version of our theorem.

Corollary 3.2 (η-postponement for normal forms). *If $a \rightsquigarrow_{\beta\eta}^* u$, then there exists some u_0 such that $a \rightsquigarrow_\beta^* u_0$ and $u_0 \rightsquigarrow_\eta^* u$.*

Corollary 3.2 states that if some term $\beta\eta$-reduces to some β-normal form, then we can postpone it in a way such that the intermediate term u_0 remains in β-normal form. The fact that u_0 is in normal form is crucial for deriving the confluence result.

3.5 Confluence of $\beta\eta$-Reduction for SN

Now, we put everything together and prove confluence using η-postponement. We start by stating the confluence properties of β and η-reductions on their own.

Lemma 3.10 (Confluence for β). *If $a \rightsquigarrow_\beta^* b_0$ and $a \rightsquigarrow_\beta^* b_1$, then there exists some c such that $b_0 \rightsquigarrow_\beta^* c$ and $b_1 \rightsquigarrow_\beta^* c$.*

Lemma 3.11 (Confluence for η). *If $a \rightsquigarrow_\eta^* b_0$ and $a \rightsquigarrow_\eta^* b_1$, then there exists some c such that $b_0 \rightsquigarrow_\eta^* c$ and $b_1 \rightsquigarrow_\eta^* c$.*

We omit the details of their proofs as they are standard. For our mechanization, we prove confluence for β-reduction using Takahashi's complete development [Takahashi 1995], and confluence for η-reduction through local confluence and Newman's lemma.

Now we can prove our main result.

Theorem 3.1 (Confluence of $\beta\eta$-Reduction for SN terms). *If $a \in \mathsf{SN}$, $a \rightsquigarrow_{\beta\eta}^* b_0$, and $a \rightsquigarrow_{\beta\eta}^* b_1$, then there exists some c such that $b_0 \rightsquigarrow_{\beta\eta}^* c$ and $b_1 \rightsquigarrow_{\beta\eta}^* c$.*

Proof. By Lemma 3.6, we know that $b_0 \in \mathsf{SN}$ and $b_1 \in \mathsf{SN}$. Therefore, there must exist some u_0 and u_1 such that $b_0 \rightsquigarrow_\beta^* u_0$ and $b_1 \rightsquigarrow_\beta^* u_1$. By Lemma 3.11, it suffices to show that u_0 and u_1 are η-equivalent. By Corollary 3.2, we can factorize the reduction sequence $a \rightsquigarrow_{\beta\eta}^* u_0$ into $a \rightsquigarrow_\beta^* u_2 \rightsquigarrow_\eta^* u_0$ for some u_2. Likewise, we can factorize the sequence $a \rightsquigarrow_{\beta\eta}^* u_1$ into $a \rightsquigarrow_\beta^* u_3 \rightsquigarrow_\eta^* u_1$ for some u_3. However, since both u_2 and u_3 are β-normal forms of a, they must be equal according to Lemma 3.10, the confluence of β-reduction. From $u_2 = u_3$, We can immediately conclude that u_0 and u_1 are η-equivalent. The result then follows from the confluence of η-reduction (Lemma 3.11). □

3.6 Reduce-and-Compare Algorithms for Equality and Subtyping

Next, we define our reduce-and-compare algorithms. Our algorithm for definitional equality is based on joinability.

Definition 3.1 (Joinability). *We say that two terms a and b are joinable (denoted as $a \downarrow b$) if there exists some term c such that $a \rightsquigarrow_{\beta\eta}^* c$ and $b \rightsquigarrow_{\beta\eta}^* c$.*

A direct consequence of the $\beta\eta$-confluence result is that $\beta\eta$-joinability is transitive.

Lemma 3.12 (Transitivity of Joinability). *If $a, b, c \in \mathsf{SN}$ and $a \downarrow b \downarrow c$, then $a \downarrow c$.*

Proof. Immediate by Theorem 3.1, Corollary 3.2, and Lemma 3.3. □

Furthermore, unlike $\beta\eta$-equivalence, the injectivity of neutral forms for joinability is immediate from its definition.

Lemma 3.13 (Injectivity of $\beta\eta$-joinability).

- *If* $e_0\ a_0 \downarrow e_1\ a_1$ *, then* $e_0 \downarrow e_1$ *and* $a_0 \downarrow a_1$.
- *If* $\pi_1\ e_0 \downarrow \pi_1\ e_1$ *or* $\pi_2\ e_0 \downarrow \pi_2\ e_1$*, then* $e_0 \downarrow e_1$.

Because joinability is reflexive and symmetric by definition, and transitive through the reasoning above, we can conclude that it is indeed an equivalence relation. Thus, we can use it as a syntactic model for declarative equality by simply erasing typing information.

Next, we define a reduce-and-compare algorithm for *subtyping*. First, consider the following untyped *one-step* subtyping relation, which compares its inputs without using $\beta\eta$-reduction.

$\boxed{A \leq_1 B}$ *(Untyped reductionless subtyping)*

$$\text{S-Refl}\ \frac{}{A \leq_1 A} \qquad \text{S-Univ}\ \frac{i \leq j}{\mathcal{U}_i \leq_1 \mathcal{U}_j} \qquad \text{S-Pi}\ \frac{A_1 \leq_1 A_0 \qquad B_0 \leq_1 B_1}{\Pi x{:}A_0.\, B_0 \leq_1 \Pi x{:}A_1.\, B_1} \qquad \text{S-Sig}\ \frac{A_0 \leq_1 A_1 \qquad B_0 \leq_1 B_1}{\Sigma x{:}A_0.\, B_0 \leq_1 \Sigma x{:}A_1.\, B_1}$$

We extend this definition to an algorithm for subtyping which first $\beta\eta$-reduces its inputs.

Definition 3.2 (Untyped Subtyping). *We say that* A *is an untyped subtype of* B*, which we denote as* $A \leq B$*, if there exists some terms* C_0 *and* C_1 *such that* $A \rightsquigarrow^*_{\beta\eta} C_0$*,* $B \rightsquigarrow^*_{\beta\eta} C_1$*, and* $C_0 \leq_1 C_1$.

We can also prove that untyped subtyping is transitive, after showing a few structural properties.

Lemma 3.14 (Sub1 Transitive). *If* $A \leq_1 B$ *and* $B \leq_1 C$*, then* $A \leq_1 C$.

Proof. We prove the following more general statement:

$$\forall A\ B, A \leq_1 B \implies \forall C, (B \leq_1 C \implies A \leq_1 C) \wedge (C \leq_1 A \implies C \leq_1 B)$$

The proof itself is straightforward by induction over the derivation of $A \leq_1 B$. □

Lemma 3.15 (Sub1 commutativity). *If* $A \leq_1 B$ *and* $A \rightsquigarrow_{\beta\eta} A_0$*, then* $\exists B_0, B \rightsquigarrow_{\beta\eta} B_0 \wedge A_0 \leq_1 B_0$.

Lemma 3.16 (Sub1 preserves SN). *Let* A*,* B*, and* C *be arbitrary lambda terms. The following statements hold.*

- $A \in \mathbf{SNe} \wedge (A \leq_1 B \vee B \leq_1 A) \implies A = B$
- $A \in \mathbf{SN} \wedge (A \leq_1 B \vee B \leq_1 A) \implies B \in \mathbf{SN}$
- $A \rightsquigarrow_{\mathbf{SN}} B \wedge (A \leq_1 B \vee B \leq_1 A) \implies A = B$

Proof. By mutual induction over the **SN** definitions. □

The transitivity of untyped subtyping then follows as a corollary of the confluence of $\beta\eta$-reduction and the above lemmas.

Corollary 3.3 (Transitivity of Subtyping). *If* $A, B, C \in \mathbf{SN}$ *and* $A \leq B \leq C$*, then* $A \leq C$.

The benefit of using the directed definition of untyped subtyping ($A \leq B$) over a declarative version obtained by erasing type annotations from $\Gamma \vdash A \leq B$ is that the following no-confusion and injectivity properties are immediate.

Lemma 3.17 (Noconfusion for Untyped subtyping). *Suppose* $A, B \in \mathbf{SN}$ *and* $A \leq B$*, then the following holds:*

- *A and B cannot be type constructors with distinct head forms.*
- *If A is in weak head neutral form, then B cannot be a type constructor.*
- *If A and B are both weak head neutral forms, then they cannot have distinct elimination forms.*

Lemma 3.18 (untyped injectivity of type constructors).

- *If* $\Pi x{:}A_0.\, B_0 \leq \Pi x{:}A_1.\, B_1$, *then* $A_1 \leq A_0$ *and* $B_0 \leq B_1$.
- *If* $\Sigma x{:}A_0.\, B_0 \leq \Sigma x{:}A_1.\, B_1$, *then* $A_0 \leq A_1$ *and* $B_0 \leq B_1$.
- *If* $\mathcal{U}_i \leq \mathcal{U}_j$, *then* $i \leq j$.

We use the the transitivity, injectivity, and no-confusion properties of joinability and untyped subtyping in the next subsection to show that they are adequate models of the declarative relations.

3.7 Strong Normalization

$\boxed{[\![A]\!]_i \searrow S}$ *(Logical predicate)*

$$\textsc{SWt-Pi}\quad \frac{F \in \mathbf{Term} \to \mathcal{P}(\mathbf{Term}) \qquad [\![A]\!]_i \searrow S_A \qquad \forall a, a \in S_A \to [\![B[a/x]]\!]_i \searrow F(a)}{[\![\Pi x{:}A.\, B]\!]_i \searrow \{b \mid b \in \hat{\Pi}(S_A, F)\}}$$

$$\textsc{SWt-Sigma}\quad \frac{F \in \mathbf{Term} \to \mathcal{P}(\mathbf{Term}) \qquad [\![A]\!]_i \searrow S_A \qquad \forall a, a \in S_A \to [\![B[a/x]]\!]_i \searrow F(a)}{[\![\Sigma x{:}A.\, B]\!]_i \searrow \{c \mid c \in \hat{\Sigma}(S_A, F)\}}$$

$$\textsc{SWt-Step}\quad \frac{A \rightsquigarrow_{\mathrm{SN}} B \qquad [\![B]\!]_i \searrow S}{[\![A]\!]_i \searrow S} \qquad \textsc{SWt-Univ}\quad \frac{j < i}{[\![\mathcal{U}_j]\!]_i \searrow \{A \mid \exists S, [\![A]\!]_j \searrow S\}} \qquad \textsc{SWt-Ne}\quad \frac{A \in \mathbf{SNe}}{[\![A]\!]_i \searrow \{a \mid a \in \mathbf{SNe}*\}}$$

(Auxiliary definitions)

$$a \in \mathbf{SNe}* \triangleq \exists b, a \rightsquigarrow_{\mathrm{SN}}^{*} b \wedge b \in \mathbf{SNe}$$
$$b \in \hat{\Pi}(S_A, F) \triangleq \forall a, a \in S_A \to b\, a \in F\,(a)$$
$$c \in \hat{\Sigma}(S_A, F) \triangleq c \in \mathbf{SNe}* \vee \exists a\, b, c \rightsquigarrow_{\mathrm{SN}}^{*} (a, b) \wedge a \in S_A \wedge b \in F(a)$$

Fig. 7. Logical Predicate

Finally, we use a logical predicate to show that all syntactically well-typed terms are contained in **SN**, i.e. are strongly normalizing.

Inspired by the inductive model for universes from Abel et al. [2008], we apply the *SN*-method [Joachimski and Matthes 2003] and define our logical predicate as an inductive relation over untyped lambda terms, closed by $\rightsquigarrow_{\mathrm{SN}}$ in rules SWt-Sigma, SWt-Step, and SWt-Ne.

The logical predicate (Figure 7) takes the form $[\![A]\!]_i \searrow S$, meaning that the raw term A is a semantically well-formed type from the ith universe and is interpreted as the set of lambda terms S. Note that Rocq does not support the definition from Figure 7 as is because it requires nesting an inductive definition inside a recursive function over the universe level i. Our Rocq encoding is based on Liu et al. [2024a], which defines the predicate inductively over a parameterized interpretation of lower universes and then ties the knot by well-founded recursion over the universe level i.

An immediate result we can show is adequacy, which implies that every well-formed type is inhabited by terms from **SNe**, and every term that lives in the logical predicate is in the set *SN*.

Lemma 3.19 (Adequacy). *If* $[\![A]\!]_i \searrow S$, *then* $\mathbf{SNe} \subseteq S \subseteq \mathbf{SN}$ *and* $A \in \mathbf{SN}$.

Proof. By nested induction over the universe level i and the derivation of $[\![A]\!]_i \searrow S$. The only interesting cases are SWt-Pi and SWt-Sigma, both of which follow from Lemma 3.5. □

The interpreted set S is always closed under $a \rightsquigarrow_{\mathrm{SN}} b$.

Lemma 3.20 (Backward closure). *If $[\![A]\!]_i \searrow S$, $b \in S$, and $a \rightsquigarrow_{\mathrm{SN}} b$, then $a \in S$.*

Lemma 3.21 (Logical predicate cases). *If $[\![A]\!]_i \searrow S$, then there exists some type constructor A_0 such that $A \rightsquigarrow^*_{\mathrm{SN}} A_0$ and $[\![A]\!]_i \searrow S$.*

Proof. By straightforward induction over the derivation of $[\![A]\!]_i \searrow S$. □

The following lemma semantically justifies the untyped subtyping relation.

Lemma 3.22 (Logical predicate preserved by subtyping).

$$\forall A\ B\ S_0\ S_1, [\![A]\!]_i \searrow S_0 \wedge [\![B]\!]_i \searrow S_1 \wedge A \leq B \implies S_0 \subseteq S_1$$

Proof. As in Lemma 3.14, we need to prove the following generalized statement.

$$\forall A\ B\ S_0\ S_1, [\![A]\!]_i \searrow S_0 \wedge [\![B]\!]_i \searrow S_1 \implies (A \leq B \implies S_0 \subseteq S_1) \wedge (B \leq A \implies S_1 \subseteq S_0)$$

The proof proceeds by induction over the derivation of $[\![A]\!]_i \searrow S_0$ followed by case analyzing the derivation of $[\![B]\!]_i \searrow S$ with the help of Lemma 3.21. Lemma 3.17 is used to rule out the contradictory cases where A and B evaluates to distinct head forms. Lemma 3.18 is used to obtain the subtyping premises required to apply the induction hypotheses.

□

The next corollary allows us to treat our logical predicate as a partial function that takes a type and universe as inputs, and returns a set of lambda terms as its output.

Corollary 3.4 (Logical predicate is functional).

$$\forall A\ S_0\ S_1, [\![A]\!]_i \searrow S_0 \wedge [\![A]\!]_i \searrow S_1 \implies S_0 = S_1$$

Proof. Immediate by instantiating Lemma 3.22 with $A \leq A$. □

We can prove that the logical predicate is cumulative.

Lemma 3.23 (Logical predicate cumulativity). *If $[\![A]\!]_i \searrow S$ and $i \leq j$, then $[\![A]\!]_j \searrow S$.*

Proof. Immediate by induction over the derivation of $[\![A]\!]_i \searrow S$. □

The cumulativity result allows us to strengthen Lemma 3.22 and Corollary 3.4 to not require the universe levels of the premises to be the same. We will continue to refer to the same lemmas to avoid duplicating the definitions, and we will implicitly compose those lemmas with the cumulativity result as needed.

We are now ready to define the notion of closing substitution, on top of which we build our notion of semantic typing, equality, subtyping, and context well-formedness.

Definition 3.3 (Closing substitutions). *Let ρ be a mapping from variables to lambda terms. We say that ρ is a closing substitution for the context Γ, denoted as $\rho \vDash \Gamma$, if for all $x : A \in \Gamma$, i, and S such that $[\![A]\!]_i \searrow S$, we have $\rho(x) \in S$.*

Definition 3.4 (Semantic well-typedness). *A term a is semantically well-typed with type A under the context Γ, denoted as $\Gamma \vDash a : A$, if for all $\rho \vDash \Gamma$, there exists some i and S such that $[\![A[\rho]]\!]_i \searrow S$ and $a[\rho] \in S$.*

Definition 3.5 (Semantic equality). *The terms a and b are semantically equal with type A under the context Γ, denoted as $\Gamma \vDash a = b : A$, when $a \downarrow b$ and $\Gamma \vDash a : A$ and $\Gamma \vDash b : A$.*

Definition 3.6 (Semantic subtyping). *The term A is a semantic subtype of B under the context Γ, denoted as $\Gamma \vDash A \leq B$, when $A \leq B$ and $\Gamma \vDash a : A$ and $\Gamma \vDash b : A$.*

Definition 3.7 (Semantic context well-formedness). *The context Γ is semantically well-formed, denoted as $\vDash \Gamma$, when for all $x : A \in \Gamma$, there exists some universe level i such that $\Gamma \vDash A : \mathcal{U}_i$.*

By unfolding the definitions above, we immediately obtain the following structural lemmas about semantic typing.

Lemma 3.24 (Semantic Weakening). *If $\Gamma \vDash a : A$ and $\Gamma \vDash B : \mathcal{U}_i$ then $\Gamma, x : B \vDash a : A$*

Lemma 3.25 (Semantic Substitution). *If $\Gamma \vDash a : A$ and $\Gamma, x : A \vDash b : B$ then $\Gamma \vDash b[a/x] : B[a/x]$*

Similar results for equality and subtyping are omitted as they directly follow from the above. Context well-formedness satisfies the following structural lemmas.

Lemma 3.26 (Structural rules for semantic well-formedness).

$$\vDash \cdot \qquad \vDash \Gamma \text{ and } \Gamma \vDash A : \mathcal{U}_i \text{ implies } \vDash \Gamma, x : A$$

Proof. The empty context case is trivial. The cons case requires Lemma 3.24. □

Note that all of the structural rules we have seen so far can be proven by unfolding the definition of semantic typing. Thus, they are useful for organizing the proof but are not necessary for deriving the fundamental theorem.

Theorem 3.2 (Fundamental Theorem).

- $\vdash \Gamma$ *implies* $\vDash \Gamma$
- $\Gamma \vdash a : A$ *implies* $\Gamma \vDash a : A$
- $\Gamma \vdash a = b : A$ *implies* $\Gamma \vDash a = b : A$
- $\Gamma \vdash A \leq B$ *implies* $\Gamma \vDash A \leq B$

Proof. By mutual induction over the typing judgments and the structural lemmas. □

Corollary 3.5 (Well-typed strong normalization). *If $\Gamma \vdash a : A$ then $a \in \mathsf{SN}$ and $A \in \mathsf{SN}$.*

Corollary 3.6 (Completeness of reduce-and-compare).

- $\Gamma \vdash A \leq B$ *implies* $A \leq B$
- $\Gamma \vdash a = b : A$ *implies* $a \downarrow b$

4 Total Correctness via Coquand's Algorithm

Corollary 3.6 gives us the completeness of the reduce-and-compare algorithm with respect to typed definitional equality and subtyping. To prove its soundness, a sufficient condition is to show that $\beta\eta$-reduction is type-preserving. Unfortunately, $\beta\eta$-reduction is known to violate subject reduction [Abel and Coquand 2007; Lennon-Bertrand 2022].

However, η-expansion, based on either the shape of the term or its type, is type-preserving. As a result, it is easier to show the soundness of algorithms that perform η-expansion with respect to typed convertibility. Therefore, we next use Coquand's algorithm extended with surjective pairing [Abel and Coquand 2007; Coquand 1991], which is based on η-expansion, as a bridge to show the soundness of our normalize-and-compare algorithm.

This detour through Coquand's algorithm is itself worthwhile as we gain a correctness proof for a second decision procedure, which can be more efficient.

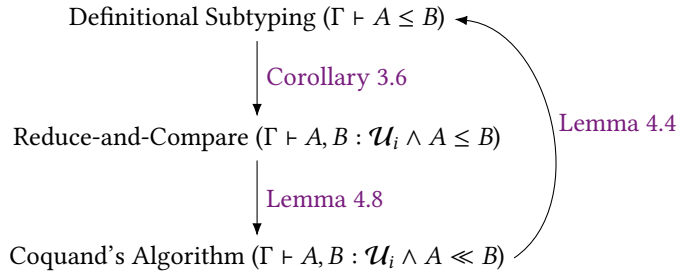

Fig. 8. Equivalence of Algorithmic and Declarative Subtyping Relations

In this section, we first extend the syntactic metatheory of Coquand's algorithm [Goguen 2005] with pairs and subtyping. We then show that this extended version of Coquand's algorithm is sound with respect to typed convertibility (Lemma 4.4), complete with respect to untyped subtyping (Lemma 4.8), and terminates on **SN** terms (Lemma 4.6). The soundness and completeness results allows us to show that Coquand's algorithm, the reduce-and-compare algorithm, and typed convertibility are equivalent on well-typed terms, following the diagram in Figure 8. This equivalence result, combined with termination, completes our main decidability proof.

The most involved part of this process is the completeness proof (Lemma 4.8). We need to convert a potentially non-type preserving $\beta\eta$-reduction sequence into a run of Coquand's algorithm: β-reductions interleaved with η-expansions. Goguen [2005] shows that Coquand's algorithm (with only function η-law) can be syntactically justified from the confluence of $\beta\eta$-reduction. We apply this technique to justify the completeness of both the subtyping and equality relation.

4.1 Coquand's Algorithmic Conversion by η-Expansion

Our version of Coquand's algorithmic equality, based on Coquand [1991] and Abel and Coquand [2007], is as follows.

$\boxed{a \leftrightarrow b}$ *(Algorithmic equality)*

$$\textsc{CE-Red}\ \frac{a \rightsquigarrow_h^* f_0 \qquad b \rightsquigarrow_h^* f_1 \qquad f_0 \sim f_1}{a \leftrightarrow b}$$

$\boxed{f_0 \sim f_1}$ *(Algorithmic equality for head normal forms)*

$$\textsc{CE-Var}\ \frac{}{x \sim x} \qquad \textsc{CE-Abs}\ \frac{a \leftrightarrow b}{\lambda x.\, a \sim \lambda x.\, b} \qquad \textsc{CE-AbsNeu}\ \frac{a \leftrightarrow e\, x \qquad x \notin \mathbf{freevar}(a)}{\lambda x.\, a \sim e} \qquad \textsc{CE-NeuAbs}\ \frac{e\, x \leftrightarrow b \qquad x \notin \mathbf{freevar}(a)}{e \sim \lambda x.\, b}$$

$$\textsc{CE-Pair}\ \frac{a_0 \leftrightarrow a_1 \qquad b_0 \leftrightarrow b_1}{(a_0, b_0) \sim (a_1, b_1)} \qquad \textsc{CE-PairNeu}\ \frac{a_0 \leftrightarrow \pi_1\, e \qquad b_0 \leftrightarrow \pi_2\, e}{(a_0, b_0) \sim e} \qquad \textsc{CE-NeuPair}\ \frac{\pi_1\, e \leftrightarrow a_0 \qquad \pi_2\, e \leftrightarrow b_0}{e \sim (a_0, b_0)} \qquad \textsc{CE-Univ}\ \frac{}{\mathcal{U}_i \sim \mathcal{U}_i}$$

$$\textsc{CE-Pi}\ \frac{A_0 \leftrightarrow A_1 \qquad B_0 \leftrightarrow B_1}{\Pi x{:}A_0.\, B_0 \sim \Pi x{:}A_1.\, B_1} \qquad \textsc{CE-App}\ \frac{e_0 \sim e_1 \qquad a_0 \leftrightarrow a_1}{e_0\, a_0 \sim e_1\, a_1} \qquad \textsc{CE-Fst}\ \frac{e_0 \sim e_1}{\pi_1\, e_0 \sim \pi_1\, e_1} \qquad \textsc{CE-Snd}\ \frac{e_0 \sim e_1}{\pi_2\, e_0 \sim \pi_2\, e_1}$$

We follow Goguen [2005] and split algorithmic equality into two mutually defined relations, where $a \leftrightarrow b$ is the top-level definition for algorithmic equality with a single rule that evaluates a and b into their respective weak head normal forms f_0 and f_1, and then check that $f_0 \sim f_1$ through the auxiliary relation defined only for weak head normal forms.

Inspired by Coquand's equality algorithm, we define the algorithmic subtyping relation below, which performs weak-head reduction and compares types in weak-head normal forms.

$\boxed{A \ll B}$ *(Algorithmic subtyping)*

$$\textsc{CLE-Red}\ \frac{A \rightsquigarrow_h^* f_0 \qquad B \rightsquigarrow_h^* f_1 \qquad f_0 \lesssim f_1}{A \ll B}$$

$\boxed{f_0 \lesssim f_1}$ *(Algorithmic subtyping for head normal forms)*

$$\textsc{CLE-Univ}\ \frac{i \leq j}{\mathcal{U}_i \lesssim \mathcal{U}_j} \qquad \textsc{CLE-Pi}\ \frac{A_1 \ll A_0 \qquad B_0 \ll B_1}{\Pi x{:}A_0.\, B_0 \lesssim \Pi x{:}A_1.\, B_1} \qquad \textsc{CLE-Sig}\ \frac{A_0 \ll A_1 \qquad B_0 \ll B_1}{\Sigma x{:}A_0.\, B_0 \lesssim \Sigma x{:}A_1.\, B_1} \qquad \textsc{CLE-NeuNeu}\ \frac{e_0 \sim e_1}{e_0 \lesssim e_1}$$

Above, $A \ll B$ is the top-level subtyping relation defined over arbitrary terms and $f_0 \lesssim f_1$ the auxiliary subtyping relation over weak head normal forms. For convenience, we refer to both the equality and subtyping relations as Coquand's algorithm.

Coquand's algorithm uses the shape of the two terms being compared to guide η-expansion. In rule CE-AbsNeu, we have a lambda term $\lambda x.\, a$ on the left-hand side and a weak neutral term e on the right-hand side. Thus, the only way to make progress is to η-expand e into $\lambda x.\, e\, x$ so we can check the equality by comparing its body $e\, x$ to a, the body of $\lambda x.\, a$.

The subtyping relation assumes that the terms being related are types. In cases where either side of $A \ll B$ reduce to a term constructor such as a lambda term or a pair, the algorithm returns false. When both A and B are in weak-head neutral form, CLE-NeuNeu falls back to checking whether A and B are equal neutral terms.

4.2 Soundness of Coquand's Algorithm

The soundness property states that well-typed terms related by algorithmic conversion must also be convertible by the type-annotated subtyping relation. The proof of soundness, at a high-level, amounts to showing that given $\Gamma \vdash A : \mathcal{U}_i$, $\Gamma \vdash B : \mathcal{U}_i$ and $A \ll B$, only well-typed terms appear in the derivation tree of $A \ll B$.

Before we dive into the soundness proof, we state two corollaries of the normalization property (Corollary 3.5) that we use to reason about subtyping.

Lemma 4.1 (Pi subtyping). *If $\Gamma \vdash \Pi x{:}A.\, B \leq C$ or $\Gamma \vdash C \leq \Pi x{:}A.\, B$, then there exists some A_0, B_0, and i such that $\Gamma \vdash C = \Pi x{:}A_0.\, B_0 : \mathcal{U}_i$.*

Lemma 4.2 (Univ subtyping). *If $\Gamma \vdash \mathcal{U}_i \leq C$ or $\Gamma \vdash C \leq \mathcal{U}_i$, then there exists some j and k such that $\Gamma \vdash C = \mathcal{U}_j : \mathcal{U}_k$.*

Now we can show the soundness of Coquand's equality with respect to definitional equality.

Lemma 4.3 (Soundness for Algorithmic Equality).

- *If $a \leftrightarrow b$ and $\Gamma \vdash a : A$ and $\Gamma \vdash b : A$, then $\Gamma \vdash a = b : A$.*
- *If $f_0 \sim f_1$, then*
  - *if $\Gamma \vdash f_0 : A$ and $\Gamma \vdash f_1 : A$, then $\Gamma \vdash f_0 = f_1 : A$.*
  - *if $f_0 \sim f_1$ and both f_0 and f_1 are neutral, then if $\Gamma \vdash f_0 : A$ and $\Gamma \vdash f_1 : B$, there exists some C such that $\Gamma \vdash C \leq A$, $\Gamma \vdash C \leq B$, and $\Gamma \vdash f_0 = f_1 : C$.*

Proof. By mutual induction over the derivations of $a \leftrightarrow b$ and $f_0 \sim f_1$. We consider only case CE-App to highlight complexities introduced by subtyping. Suppose we have $f_0 = e_0\, a_0 \sim e_1\, a_1 = f_1$ It suffices to show that the third bullet point holds as it subsumes the second bullet point when

both terms are neutral. Suppose also that $\Gamma \vdash e_0\ a_0 : A$ and $\Gamma \vdash e_1\ a_1 : B$ for some A and B. By the generation lemma (Lemma 2.2), there exists some terms A_0, B_0, A_1 and B_1 such that $\Gamma \vdash e_0 : \Pi x{:}A_0.\, B_0$, $\Gamma \vdash e_1 : \Pi x{:}A_1.\, B_1$, $\Gamma \vdash a_0 : A_0$, $\Gamma \vdash a_1 : A_1$, $\Gamma \vdash B_0[a_0/x] \leq A$ and $\Gamma \vdash B_1[a_1/x] \leq B$.

Applying the induction hypothesis to the neutral forms e_0 and e_1, we know that there exists some type C such that $\Gamma \vdash e_0 = e_1 : C$ where $\Gamma \vdash C \leq \Pi x{:}A_0.\, B_0$ and $\Gamma \vdash C \leq \Pi x{:}A_1.\, B_1$.

By Lemma 4.1, there exists types A_2, B_2 and i such that $\Gamma \vdash \Pi x{:}A_2.\, B_2 = C : \mathcal{U}_i$.

Therefore, we can replace the occurrences of C with $\Pi x{:}A_2.\, B_2$, obtaining the following statements.

- $\Gamma \vdash e_0 = e_1 : \Pi x{:}A_2.\, B_2$
- $\Gamma \vdash \Pi x{:}A_2.\, B_2 \leq \Pi x{:}A_0.\, B_0$
- $\Gamma \vdash \Pi x{:}A_2.\, B_2 \leq \Pi x{:}A_1.\, B_1$

By the injectivity rule L-PiProjOne, we deduce $\Gamma \vdash A_0 \leq A_2$ and $\Gamma \vdash A_1 \leq A_2$. This allows us to convert the type of both a_0 and a_1 to A_2, so we can obtain $\Gamma \vdash a_0 = a_1 : A_2$ from the induction hypothesis. By rule E-App, we have $\Gamma \vdash e_0\ a_0 = e_1\ a_1 : B_2[a_0/x]$. It now suffices to show that $\Gamma \vdash B_2[a_0/x] \leq B_0[a_0/x]$ and $\Gamma \vdash B_2[a_0/x] \leq B_1[a_1/x]$.

We can prove $\Gamma \vdash B_2[a_0/x] \leq B_0[a_0/x]$ immediately by L-PiProjTwo. To prove $\Gamma \vdash B_2[a_0/x] \leq B_1[a_1/x]$, we cannot directly apply rule L-PiProjTwo as a_0 is not necessarily typed under A_1. Instead, we need to first apply transitivity and show $\Gamma \vdash B_2[a_0/x] \leq B_2[a_1/x] \leq B_1[a_1/x]$. The two subgoals then follow from L-PiProjTwo. □

The soundness property for subtyping is formulated similarly, though we do not need to include the special case when both types A and B are neutral since subtyping for neutral types degenerate into equality, whose soundness we have already proven.

Lemma 4.4 (Soundness for Algorithmic Subtyping).

- *If $A \ll B$, and $\Gamma \vdash A : \mathcal{U}_i$ and $\Gamma \vdash B : \mathcal{U}_i$, then $\Gamma \vdash A \leq B$.*
- *If $f_0 \lesssim f_1$, and $\Gamma \vdash f_0 : \mathcal{U}_i$ and $\Gamma \vdash f_1 : \mathcal{U}_i$, then $\Gamma \vdash f_0 \leq f_1$.*

Proof. By mutual induction on $A \lesssim B$ and $A \ll B$. The only interesting case is CLE-NeuNeu, where A and B are neutral. We finish the proof by simply composing Lemma 4.3 and rule L-Eq. □

4.3 Termination of Coquand's Algorithm

To prove termination, we use a termination metric over the input terms a and b from Goguen [2005], extended to include pairs. (A similar metric with pairs but only for β-forms can be found in Abel and Coquand [2007].)

Definition 4.1 (Termination Metric for Coquand's Algorithm). *Given two β-normalizing terms a and b, there must exists some β-normal forms u_0 and u_1 such that $a \leadsto_{lo}^* u_0$ and $b \leadsto_{lo}^* u_1$. Let m and n be the number of steps it takes for a and b to reduce to their respective normal form, we define the termination metric $\mathcal{T}(a, b) = m + n + |u_0| + |u_1|$, where the $|\cdot|$ is a size function over lambda terms, defined as follows.*

$$
\begin{aligned}
|\lambda x.\, a| &= 3 + |a| \\
|a\ b| &= 1 + |a| + |b| \\
|(a, b)| &= 3 + |a| + |b| \\
|\pi_1\ a| &= 1 + |a| \\
|\pi_2\ a| &= 1 + |a| \\
&\cdots
\end{aligned}
$$

The $|\cdot|$ function weighs constructors more than elimination forms, allowing us to show that the inputs to recursive calls in CE-AbsNeu and CE-PairNeu are decreasing.

Goguen [2005] uses the definition of $\mathcal{T}$ as metric for both the termination and completeness proof. We take a more modular approach by introducing an intermediate inductive relation that characterizes the domain of the conversion algorithm. The intermediate relation serves two purposes. First, once we have shown that every pair of terms a and b such that $\mathcal{T}(a, b)$ is defined inhabit the domain, then we can immediately recover termination by the Bove-Capretta method [Bove and Capretta 2005]. Second, the intermediate relation abstracts away the low-level details of the termination metric and allows us to reason by induction over the call-graph of the algorithm, significantly simplifying our reasoning.

We write the relations encoding the domains for equality and subtyping as $(f_0, f_1) \in \mathcal{A}$ and $(f_0, f_1) \in \mathcal{S}$, which correspond to $f_0 \sim f_1$ and $f_0 \lesssim f_1$, and $(a, b) \in \mathcal{A}^*$ and $(A, B) \in \mathcal{S}^*$, which correspond to $a \leftrightarrow b$ and $A \ll B$. We omit the definition of these relations here as they are a straightforward application of the Bove-Capretta method. The definition can be found in the supplementary material.

The following properties can be proven by induction over the termination metric. We omit the proof, as it follows the exact same structure as the termination proof by Goguen [2005].

Lemma 4.5 (Metric implies Domain). *If there exists some n such that $\mathcal{T}(a, b) = n$, then $(a, b) \in \mathcal{A}^*$ and $(a, b) \in \mathcal{S}^*$.*

By Corollary 3.5, all well-typed terms are strongly normalizing, and therefore for every pair of terms a and b, $\mathcal{T}(a, b)$ is defined. By Lemma 4.5, we deduce that $(a, b) \in \mathcal{A}^*$ and $(a, b) \in \mathcal{S}^*$, and we can then define the equality and subtyping functions recursively over the derivations of the domains, concluding the proof of the following termination result.

Lemma 4.6 (Termination of Coquand's Algorithm). *If $\Gamma \vdash A : \mathcal{U}_i$ and $\Gamma \vdash B : \mathcal{U}_i$, then $A \ll B$ terminates.*

Furthermore, fixing a and b as inputs, we can show that the derivation of $(a, b) \in \mathcal{A}^*$ or $(a, b) \in \mathcal{S}^*$ does not affect the result of the algorithm. In our Rocq mechanization, we leverage this irrelevance property to place the domain definitions in the `Prop` sort so the extracted conversion algorithm does not have to compute the proofs or recurse over a gas parameter.

4.4 Completeness of Coquand's Algorithm w.r.t Untyped Subtyping

To show completeness, we reuse the domain definitions from Section 4.3 as our induction metric and formulate the completeness theorem as follows.

Lemma 4.7 (Completeness of Algorithmic Equality (auxiliary)).

- *If $(a, b) \in \mathcal{A}^*$, $\Gamma \vdash a : A$, $\Gamma \vdash b : A$ and $a \downarrow b$, then $a \leftrightarrow b$.*
- *If $(f_0, f_1) \in \mathcal{A}$, then*
  - *if $\Gamma \vdash f_0 : A$, $\Gamma \vdash f_1 : A$ and $f_0 \downarrow f_1$, then $f_0 \sim f_1$.*
  - *if f_0 and f_1 are both weak head neutral forms, then given $\Gamma \vdash f_0 : A$, $\Gamma \vdash f_1 : B$ and $f_0 \downarrow f_1$, we have $f_0 \sim f_1$.*

Proof. By mutual induction over the derivations of $(a, b) \in \mathcal{A}^*$ and $(f_0, f_1) \in \mathcal{A}$. The proof obligations can be split into two categories: proving the well-typedness of the subterms to be able to invoke the inductive hypotheses, and showing that the recursive calls are made on terms are still $\beta\eta$-joinable so we can meet the prerequisite for applying the induction hypotheses.

We consider the case for rule A-AppCong as an example. In case A-AppCong, we have $(e_0\ a_0, e_1\ a_1) \in \mathcal{A}$ where $(a_0, a_1) \in \mathcal{A}^*$, $(e_0, e_1) \in \mathcal{A}$, $\Gamma \vdash e_0\ a_0 : C_0$, $\Gamma \vdash e_1\ a_1 : C_1$, and $e_0\ a_0 \downarrow e_1\ a_1$. By the Generation Lemma (Lemma 2.2), there exists some $A_0\ A_1\ B_0\ B_1$ such that $\Gamma \vdash e_0 : \Pi x{:}A_0.\ B_0$, $\Gamma \vdash a_0 : A_0$, $\Gamma \vdash e_1 : \Pi x{:}A_1.\ B_1$, $\Gamma \vdash a_1 : A_1$.

By the injectivity of application forms for $\beta\eta$-joinability (Lemma 3.13), we deduce from $e_0\ a_0 \downarrow e_1\ a_1$ that $e_0 \downarrow e_1$ and $a_0 \downarrow a_1$. Since we also know that $\Gamma \vdash e_0 : \Pi x{:}A_1.\, B_1$, $\Gamma \vdash e_1 : \Pi x{:}A_0.\, B_0$ from the generation lemma, we can invoke the induction hypothesis and derive $e_0 \sim e_1$.

By Lemma 4.3, we know that there exists a common subtype C for $\Pi x{:}A_0.\, B_0$ and $\Pi x{:}A_1.\, B_1$. By Lemma 4.1 and rule L-PiProjOne, there exists some A_2 such that $\Gamma \vdash A_0 \le A_2$, $\Gamma \vdash A_1 \le A_2$. Thus, we have $\Gamma \vdash a_0 : A_2$, $\Gamma \vdash a_1 : A_2$, and $a_0 \downarrow a_1$. By the induction hypothesis, we deduce $a_0 \leftrightarrow a_1$.

From $e_0 \sim e_1$ and $a_0 \leftrightarrow a_1$, we conclude that $e_0\ a_0 \sim e_1\ a_1$. □

The completeness of algorithmic subtyping is stated as follows.

Lemma 4.8 (Completeness of Coquand's Algorithmic Subtyping (auxiliary)).

- *If* $(f_0, f_1) \in \mathcal{S}$, $\Gamma \vdash f_0 : \mathcal{U}_i$, $\Gamma \vdash f_1 : \mathcal{U}_i$, *and* $f_0 \le f_1$, *then* $f_0 \lesssim f_1$.
- *If* $(A, B) \in \mathcal{S}^*$, $\Gamma \vdash A : \mathcal{U}_i$, $\Gamma \vdash B : \mathcal{U}_i$, *and* $A \le B$, *then* $A \ll B$.

Proof. By mutual induction over the derivations of $(A, B) \in \mathcal{S}^*$ and $(A, B) \in \mathcal{S}$. The neutral case (CLE-NeuNeu) requires Lemma 4.7. □

Composing the completeness of Coquand's algorithm with respect to untyped subtyping (Lemma 4.8), the soundness of Coquand's algorithm with respect to definitional subtyping (Lemma 4.4), and the completeness of untyped subtyping with respect to definitional subtyping (Corollary 3.6), we conclude our decidability proof with the following equivalence results.

Lemma 4.9 (Reduce-and-compare and Coquand's Algorithm are Correct). *Given* $\Gamma \vdash A : \mathcal{U}_i$ *and* $\Gamma \vdash B : \mathcal{U}_j$, *the statements* $\Gamma \vdash A \le B$, $A \le B$, *and* $A \ll B$ *are all equivalent.*

Theorem 4.1 (Type Conversion is Decidable). *Given* $\Gamma \vdash A : \mathcal{U}_i$ *and* $\Gamma \vdash B : \mathcal{U}_j$, *the relation* $\Gamma \vdash A \le B$ *is decidable.*

5 Discussion

5.1 Admissibility of Π-Injectivity

One nonstandard feature of $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$ is the inclusion of the injectivity rules L-PiProj1 and L-PiProj2. We explicitly add these rules as part of the definitional subtyping relation so that we can prove subject reduction syntactically, simplifying the definition of our logical predicate in Section 3.7.

These injectivity rules are sound and justified by our algorithm so they do not compromise the implementation of type checking. Indeed, the results from Section 4 allow us to replace conversion rule Wt-Conv with the following rule based on algorithmic subtyping.

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B : \mathcal{U}_i \qquad A \ll B}{\Gamma \vdash a : B}\ \textsc{Wt-AConv}$$

Algorithmic subtyping does not depend on definitional equality, so the inclusion of injectivity rules in the latter does not affect the implementation of the type checker.

That said, it is difficult to remove the injectivity rules from the specification of the type system. While the rules do not affect the operation of algorithm, they are needed to show its correctness. The justification of rule Wt-AConv ultimately relies on subject reduction, which in turn depends on Π-injectivity. Therefore, the correctness of rule Wt-AConv does not imply that Π-injectivity is admissible.

Why would we want to show that Π-injectivity is admissible? After all, they causes no issues with our implementation and we have shown that the type system with these rules is decidable. The real issue lies in model construction. While the reducibility model in Section 4 satisfies the injectivity of type constructors, there also exist models of dependent type theory where injectivity

of type constructors is not true. For example, the standard set theoretic models by Miquel and Werner [2003] and Timany and Sozeau [2017] forget too much information about the syntax of the type theory for injectivity to hold semantically. But, the contravariant subtyping rule for function types already rules out set theoretic models for $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$. Therefore, we see no little reason to avoid the inclusion of the injectivity rules that simplify our proofs.

Alternatively, if the admissibility of type injectivity is needed, it is possible to replace the logical predicate in Section 3.7 with a Kripke-style logical relation, from which injectivity of type constructors and strong normalization can both be derived at once. The Kripke-style logical relation can remain oblivious of the algorithmic conversion rules as the correctness of the algorithm can be obtained by composing Π-injectivity and strong β-normalization with our syntactic proof in Section 4. Thus, while we would still need to pay the cost of constructing a PER model to obtain Π-injectivity, we can avoid the intricate setup of a parameterized generic equality found in Abel et al. [2017]; Adjedj et al. [2024] and keep our Kripke-style logical relation minimal.

5.2 Annotated Lambda Terms

Another slightly nonstandard feature of $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$is that function abstractions are not annotated with the types of their arguments. Such annotations are known to invalidate the confluence property for $\beta\eta$-reduction. Suppose x, y, and z are distinct variables, the term $\lambda x\!:\!A.(\lambda y\!:\!B.z)x$ can either η-reduced to $\lambda y\!:\!B.z$ or β-reduced to $\lambda x\!:\!A.z$. Thus, when A and B are distinct types, we obtain a counterexample to confluence.

However, the type system is designed such that annotating the lambda terms with types should not lead to significant changes to our proof development. Suppose the terms $\lambda x\!:\!A.a$ and $\lambda x\!:\!B.b$ share the same type. By the generation lemma, we can find some common subtype C of the types A and B. If we continue to use rule E-AbsExt to subsume both the congruence and η-law for abstractions, then there is no need to explicitly compare the annotations A and B as the congruence rule implied by rule E-AbsExt identifies lambda terms whose domains share a common subtype. Thus, the $\beta\eta$-confluence result over erased lambda terms is sufficient to justify the correctness of the algorithm, even though the syntax involves type-annotated lambdas. In our work, we consider only unannotated lambda terms so we do not have to prove more simulation/commutativity lemmas between erasure, one-step subtyping, and reduction.

What happens if the declarative system explicitly requires the annotations to be equal in the congruence rule for abstractions, in a way that is not supported by the generation lemma? For example, suppose we are working with a pure-type system without uniqueness of sorts, and we replace rule E-AbsExt with the following congruence rule for type-annotated abstractions (cf. Siles and Herbelin [2012]).

$$\text{E-AbsCong}\quad \frac{\Gamma \vdash A = A_0 : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma, x : A \vdash a = b : B}{\Gamma \vdash \lambda x\!:\!A.\ a = \lambda x\!:\!A_0.\ b : \Pi x\!:\!A.\ B}$$

In that case, we could not use the $\beta\eta$-confluence result over erased lambda terms. The conversion algorithm would be forced to compare annotations, but $\beta\eta$-joinability over erased terms does not include enough information about the annotations.

5.3 Relaxing the SN Condition

Our confluence property for $\beta\eta$-contraction holds only for the terms in the set **SN**. However, this condition is stronger than necessary.

We use **SN** in two ways. To derive η-postponement, we use the **SN** predicate to rule out terms that can step into stuck forms. To derive $\beta\eta$-confluence, we also use the **SN** predicate to find β-normal forms.

However, to rule out terms that can get stuck, it suffices to find a predicate P that satisfies following properties, which do not imply that terms are strongly normalizing:

- If $P(a)$ holds, then a is not a stuck term.
- If $P(a)$ holds, then $P(b)$ holds if b is a subterm of a.
- If $P(a)$ holds and $a \rightsquigarrow_{\beta\eta} b$, then $P(b)$ holds.

In our Rocq development, our η-postponement property is parameterized over such a predicate P.

An alternative to **SN** is a predicate that holds precisely when terms cannot reduce to a stuck state. Consider the following *coinductively* defined set.

Definition 5.1. *We define* **okay** *as the the largest set of untyped lambda terms that satisfies the following properties.*

- *If* $a \in$ **okay**, *then* a *does not contain any stuck subterms.*
- *If* $a \in$ **okay** *and* $a \rightsquigarrow_{\beta\eta} b$, *then* $b \in$ **okay**.

Note that **okay** contains the non-terminating term $(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$ since it loops without ever reaching a stuck state.

Proposition 5.1. *Terms in the set* **okay** *satisfy the postponement of* η*-reductions.*

Proof. It suffices to show that **okay** satisfies the three condition we specify above.

The first and third properties are immediate by definition. The second property can be verified by the coinduction principle. □

Thus, the η-postponement proof can be extended to systems that are not necessarily terminating as long as the terms do not get stuck. In particular, showing that all well-typed terms are **okay**, a similar property to type safety, is sufficient.

To derive $\beta\eta$-confluence from η-postponement, we do not need *all* reduction sequences to result in a β-normal form, we only need *one* reduction sequence to terminate. Thus, assuming η-postponement, we can obtain $\beta\eta$-confluence from weak β-normalization.

6 Related Work

6.1 Syntactic Methods for Type Conversion

In this section, we compare our approach with other syntactic proofs of decidable type conversion for typed $\beta\eta$-equality. (In systems with *untyped* β-equivalence as definitional equality, the decidability of type conversion follows immediately from strong normalization and confluence [Barendregt 1993; Geuvers 1994; Luo 1990].)

Goguen [2005] proves the decidability of a logical framework with a typed $\beta\eta$-equality as its convertibility relation. This system contains only the η-law for functions; surjective pairing is not considered. Unlike $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$, the logical framework does not support large eliminations or polymorphism. Goguen shows that the soundness, completeness, and termination of Coquand's untyped algorithm [Coquand 1991] and Pfenning and Harper's typed algorithm [Harper and Pfenning 2005] can be justified by a syntactic proof parameterized over the confluence of $\beta\eta$-reduction and metatheoretic properties about the type system, including normalization and subject reduction (for $\beta\eta$-reduction). Instead of proving completeness with respect to the typed definitional equality, he proves the completeness of algorithmic conversion with respect to untyped $\beta\eta$-equivalence. That is, if two well-typed terms $\beta\eta$-reduce to the same term, then one can show their equivalence by performing η-expansion and β-reduction.

In our work, we make the observation that we can perform Goguen's completeness proof without requiring η-reduction to be type-preserving. This allows us to use the completeness proof as a bridge to relate untyped $\beta\eta$-reduction, which can relate ill-typed terms, to the typed convertibility relation, which only relates well-typed terms. Furthermore, our development applies to a more expressive language, which includes large eliminations, surjective pairing, and subtyping. The latter two features mean that we must rely on **SN** to show confluence and that our development cannot rely on the uniqueness of typing.

Abel and Coquand [2007] use a PER model to justify two conversion algorithms for a Curry-style logical framework that includes surjective pairing. Like later work by Abel and Scherer [2012], Abel and Coquand [2007] avoid the confluence problem that comes with surjective pairing by using the extensionality of the PER to model the η-laws from definitional equality. Instead, the PER model directly justifies a specific reduce-and-compare algorithm that compares the η-equivalence of β-normal forms. Similar to our development, the correctness of Coquand's algorithm is proven syntactically by showing its completeness with respect to the reduce-and-compare algorithm. The lack of a confluence result means that other variants of reduce-and-compare algorithm that may interleave β and η-reduction are not justified. Furthermore, to prove the completeness of Coquand's algorithm, Abel and Coquand start by proving it complete with respect to β-normal forms and then lift the result to include β-normalizing terms. In our development, the confluence result allows us to directly prove completeness with respect to β-normalizing terms. Finally, our proof is simpler as it only involves a logical predicate. Because convertibility in our system is modeled as untyped $\beta\eta$-reduction, our proof is compatible with untyped definitional equality and we can add subtyping to our system without any no changes to the logical predicate.

Lennon-Bertrand [2025] proves in Rocq the positive soundness, negative soundness, and termination of algorithmic conversion for a Martin-Löf type theory with one universe. The negative and positive soundness results ensure that the algorithm, upon termination, returns true precisely when its inputs are definitionally equal. This relaxation of the termination condition allows Lennon-Bertrand [2025] to formulate correctness properties for type systems that are not normalizing.

Similar to our work, Lennon-Bertrand [2025] identifies several injectivity and normalization properties that are needed to carry out a syntactic correctness proof. However, the injectivity properties required by Lennon-Bertrand [2025] differ from ours in that they are about the typed definitional equality, whereas the injectivity properties we prove to justify our algorithm are about joinability, an untyped relation.

Proving typed injectivity and normalization properties has the benefit of being compatible with singleton η-laws. However, the typed properties not only requires a more complicated logical relation, but can also cause problems when trying to separate the semantic and syntactic proofs. Lennon-Bertrand [2025] prove the *deep normalization* property, which is needed for termination, using the logical relation by Adjedj et al. [2024]. The same fundamental theorem used to show deep normalization, however, already implies the completeness of algorithmic conversion! Thus, when total correctness is concerned, the proof development by Lennon-Bertrand [2025], composed with the logical relation by Adjedj et al. [2024], does not completely decouple the logical relation from the syntactic proof about the termination of the algorithm. As part of future work, Lennon-Bertrand [2025] suggests alternative ways of showing neutral injectivity or termination that do not require the detour to the completeness result of algorithmic conversion, such as the domain theoretic method by Coquand and Huber [2019].

6.2 Mechanized Metatheory for Dependent Types

All of our results have been mechanized using the Rocq proof assistant [Team 2024], leading to a preference for syntactic reasoning, which is convenient and well supported by this tool. We

leverage the rich ecosystem of Rocq to aid our proof development; we use CoqHammer [Czajka and Kaliszyk 2018] to automate our proof scripts, and Autosubst [Dapprich and Dudenhefner 2021] to generate and reason about the de Bruijn representation of our syntax. However, we are not the first to use a proof assistant to reason about dependent type theories and we draw on prior efforts.

Barras [1996] uses the Rocq (Coq) proof assistant to mechanize the strong normalization the Calculus of Constructions (CoC) and extract a decidable type checker. CoC, as an instance of Barendregt's Pure Type Systems (PTS) [Barendregt 1991], uses untyped β-equivalence as its definitional equality. The conversion algorithm, which compares β-normal forms of its input, is thus directly justified by the confluence of β-reduction.

Liu et al. [2025] mechanize in Rocq the decidability of conversion for DCOI, a dependently typed language that supports proof-irrelevance. The definitional equality of DCOI replaces α-equivalence with a notion of syntactic indistinguishability, which ignores components of the term that cannot be distinguished by a specified observer level. The conversion algorithm checks the syntactic indistinguishability of the β-normal forms of its inputs. They prove the correctness of the algorithm syntactically by composing normalization, β-confluence, and simulation properties which relate β-reduction and syntactic indistinguishability.

Our proof shares a similar architecture to that found in Barras [1996] and Liu et al. [2025]. Both of these works use confluence-based approach to show the decidability of algorithmic conversion. This paper shows that the same approach works for systems with η-laws for functions and pairs.

Other projects that mechanize results about dependent-type theory use methods that are less related to this work.

Abel et al. [2017] use Agda to show the decidability of type conversion for a dependent type theory with one fixed universe. The algorithm used to decide type conversion is type directed and is similar to the one found in Harper and Pfenning [2005]. This work, like Harper and Pfenning [2005] and Abel and Scherer [2012], uses a Kripke-style logical relation to show the total correctness of the algorithm. To avoid having to define two separate logical relations, one for soundness and one for completeness, Abel et al. parameterize their logical relation by a relation called *generic equality*. A generic equality consists of an equality between types, an equality between terms, and another equality specifically for neutral terms. They then prove that the fundamental theorem holds for generic equalities that satisfy an interface of required properties. Thus, the soundness and completeness proofs now require proving that different equalities satisfy the interface, and the fundamental theorem only needs to be proven once. Adjedj et al. [2024] adopt the same technique of a parameterized logical relation in Rocq to not only show the decidability of type conversion, but also show the decidability of type checking through a bidirectional type checker.

Wieczorek and Biernacki [2018] mechanize the correctness of a normalization by evaluation (NbE) algorithm for a dependent type theory with one universe in Rocq, following the pen and paper proof by Abel [2013]. Similarly, Hu et al. [2023] mechanize NbE for a modal dependent type theory with a cumulative universe in Agda. Building on the proof development by Hu et al. [2023], Jang et al. [2025] mechanize the correctness of an NbE algorithm for a more expressive dependent type theory that also supports subtyping, similar to $\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$. Jang et al. [2025], like Adjedj et al. [2024], also include a decidable type checker on top of the decidable conversion algorithm. The subtyping relation in Jang et al. [2025] is covariant on functions, whereas our subtyping relation is contravariant on functions.

Altenkirch and Kaposi [2016] mechanize the correctness of NbE for a dependent type theory with a single universe in Agda. Instead of working on explicit syntax, they employ an algebraic representation of the type system as categories with families [Hofmann and Hofmann 1997], encoded as a quotient inductive type [Altenkirch et al. 2018], where terms, which are always well-typed, are quotiented by definitional equality. Unlike the proofs we have discussed so far

(including our own), Altenkirch and Kaposi [2016] use a *proof-relevant* logical predicate, where the evidence that a term is reducible is not an irrelevant proof object, but a computationally relevant structure. The proof-relevance of the logical predicate sidesteps the need for an extensional PER model to justify the η-laws [Coquand 2019]. In our work, we show that a *proof-irrelevant* logical predicate can also help avoid an extensional model by leveraging the confluence of $\beta\eta$-reduction.

Barras [2012] axiomatizes set theory in Rocq, and then models Calculus of Inductive of Constructions (CIC) using the formalized set theory. The axiomatization of set theory requires adding additional axioms to Rocq, but also allows Barras to model the universe hierarchy with an impredicative `Prop` sort, which cannot be inductively defined.

Sozeau et al. [2019] prove the correctness of a type checker for a subset of Rocq that contains only β-rules. Due to Gödel's incompleteness theorem, Sozeau et al. [2019] formulate their correctness result syntactically by assuming strong normalization.

Kravchuk-Kirilyuk et al. [2020] extend the core language for Dependent Haskell [Weirich et al. 2017] with function η-laws in Rocq. The decidability of conversion for System DC, an annotated version of the core calculus, has a direct syntactic proof since the type checker only needs to validate traces of reduction sequences that are part of the term syntax. The confluence of $\beta\eta$-equivalence is used to justify the consistency its extensionally flavored equational theory.

7 Conclusion and Future Work

In this work, we show the decidability of type conversion for an expressive dependent type theory with function and pair η-laws. To show the confluence of untyped $\beta\eta$-reduction, we use the inductively defined strongly normalizing terms of Van Raamsdonk and Severi [1995]. From confluence and the normalization of well-typed terms, we derive a syntactic proof of the total correctness of Coquand's algorithm (extended with subtyping) and of a reduce-and-compare algorithm. Our object language ($\lambda^{\Pi,\Sigma,U_i,\mathbb{N}}$) is expressive and demonstrates that the features of modern proof assistants, such as subtyping, large eliminations, and inductive datatypes, are compatible with this approach. We have mechanized all of our proofs using the Rocq theorem prover.

Our proof method is novel as it uses a syntactic confluence proof to directly model typed convertibility as untyped $\beta\eta$-reduction. Compared to existing proof methods, our confluence-based approach is compatible with untyped convertibility and modularly combines a minimal semantic proof of normalization with a syntactic proof of decidability. The modularity gives us the option to carry out the bulk of the decidability proof in a weak metatheory and the ability to explore different algorithms with minimal changes to the whole development.

In future work, we hope to leverage the modularity of our proof development to prove the correctness of other type conversion algorithms through syntactic means. For example, we could replace the iterated weak-head β-reduction in Coquand's algorithm with an efficient untyped NbE algorithm, which can be proven correct syntactically through confluence [Grégoire and Leroy 2002]. We also would like to apply our techniques to systems, such as ICC [Barras and Bernardo 2008; Miquel 2001] and DCOI [Liu et al. 2024b], that require untyped equality judgments and so have previously not included η-laws for dependent pairs.

References

Andreas Abel. 2013. *Normalization by evaluation: Dependent types and impredicativity.* Ph. D. Dissertation. Ludwig-Maximilians-Universität München.

Andreas Abel and Thierry Coquand. 2007. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae* 77, 4 (2007), 345–395.

Andreas Abel, Thierry Coquand, and Peter Dybjer. 2008. Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Philippe Audebaud and Christine Paulin-Mohring (Eds.). Springer, Berlin, Heidelberg, 29–56. doi:10.1007/978-3-540-70594-9_4

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2017. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Dec. 2017), 29 pages. doi:10.1145/3158111

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (2012), 1–36. doi:10.2168/lmcs-8(1:29)2012

Robin Adams. 2006. Pure type systems with judgemental equality. *Journal of Functional Programming* 16, 2 (2006), 219–246.

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs* (London, UK) *(CPP 2024)*. Association for Computing Machinery, New York, NY, USA, 230–245. doi:10.1145/3636501.3636951

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*. Springer International Publishing Cham, 293–310.

Thorsten Altenkirch and Ambrus Kaposi. 2016. Normalisation by Evaluation for Dependent Types. (2016), 16 pages. doi:10.4230/LIPICS.FSCD.2016.6 Artwork Size: 16 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany.

Henk Barendregt. 1991. Introduction to generalized type systems. *Journal of Functional Programming* 1, 2 (1991), 462–490. doi:10.1017/s0956796800020025

Henk P. Barendregt. 1993. *Lambda Calculi with Types*. Oxford University Press, Inc., USA, 117–309.

Bruno Barras. 1996. *Coq en coq*. Ph. D. Dissertation. INRIA.

Bruno Barras. 2012. Semantical investigations in intuitionistic set theory and type theories with inductive families. Thèse l'habilitation à diriger des recherches, Université Paris 7 - Denis Diderot.

Bruno Barras and Bruno Bernardo. 2008. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *Foundations of Software Science and Computational Structures*, Roberto Amadio (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 365–379. doi:10.1007/978-3-540-78499-9_26

Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. 1998. Normalization by evaluation. *Prospects for Hardware Foundations: ESPRIT Working Group 8533 NADA—New Hardware Design Methods Survey Chapters* (1998), 117–137.

Ana Bove and Venanzio Capretta. 2005. Modelling general recursion in type theory. *Mathematical Structures in Computer Science* 15, 4 (2005), 671–708. doi:10.1017/S0960129505004822

Pritam Choudhury, Harley Eades III, and Stephanie Weirich. 2022. A Dependent Dependency Calculus. In *Programming Languages and Systems, ESOP 2022 (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 403–430. doi:10.1007/978-3-030-99336-8_15 Artifact available.

Thierry Coquand. 1991. An algorithm for testing conversion in type theory. *Logical frameworks* 1 (1991), 255–279.

Thierry Coquand. 2019. Canonicity and normalization for dependent type theory. *Theoretical Computer Science* 777 (July 2019), 184–191. doi:10.1016/j.tcs.2019.01.015

Thierry Coquand and Simon Huber. 2019. An Adequacy Theorem for Dependent Type Theory. *Theory of Computing Systems* 63, 4 (May 2019), 647–665. doi:10.1007/s00224-018-9879-9

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of automated reasoning* 61 (2018), 423–453.

Adrian Dapprich and Andrej Dudenhefner. 2021. Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml. *Bachelor thesis, Saarland University* (2021).

Herman Geuvers. 1994. A short and flexible proof of strong normalization for the calculus of constructions. In *International Workshop on Types for Proofs and Programs*. Springer, Berlin, Heidelberg, 14–38. doi:10.1007/3-540-60579-7_2

Jan Herman Geuvers. 1993. *Logics and type systems*. [Sl: sn].

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and types*. Vol. 7. Cambridge University Press, Cambridge.

Healfdene Goguen. 2005. Justifying Algorithms for $\beta\eta$-Conversion. In *Foundations of Software Science and Computational Structures*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Vladimiro Sassone (Eds.). Vol. 3441. Springer Berlin Heidelberg, Berlin, Heidelberg, 410–424. doi:10.1007/978-3-540-31982-5_26 Series Title: Lecture Notes in Computer Science.

Benjamin Grégoire and Xavier Leroy. 2002. A compiled implementation of strong reduction. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. 235–246.

Robert Harper and Frank Pfenning. 2005. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic (TOCL)* 6, 1 (2005), 61–101.

Martin Hofmann and Martin Hofmann. 1997. Syntax and semantics of dependent types. *Extensional Constructs in Intensional Type Theory* (1997), 13–54.

Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by evaluation for modal dependent type theory. *Journal of Functional Programming* 33 (2023), e7. doi:10.1017/S0956796823000060

Junyoung Jang, Jason ZS Hu, Antoine Gaulin, and Brigitte Pientka. 2025. McTT: Building A Correct-By-Construction Proof Checker For Martin-Löf Type Theory. (2025).

Felix Joachimski and Ralph Matthes. 2003. Short proofs of normalization for the simply- typed λ-calculus, permutative conversions and Gödel's T. *Archive for Mathematical Logic* 42, 1 (Jan. 2003), 59–87. doi:10.1007/s00153-002-0156-9

Jan Willem Klop. 1980. *Combinatory Reduction Systems.* Mathematisch centrum, Amsterdam.

J. W. Klop and R. C. de Vrijer. 1989. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation* 80, 2 (Feb. 1989), 97–113. doi:10.1016/0890-5401(89)90014-X

Anastasiya Kravchuk-Kirilyuk, Antoine Voizard, and Stephanie Weirich. 2020. Eta-Equivalence in Core Dependent Haskell. *Leibniz international proceedings in informatics* 175 (2020).

Meven Lennon-Bertrand. 2022. À bas l'η—Coq's troublesome η-conversion. The first Workshop on the Implementation of Type Systems (WITS). https://www. meven. ac/documents/22-WITS-abstract. pdf.

Meven Lennon-Bertrand. 2025. What Does It Take to Certify a Conversion Checker?. In *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 337)*, Maribel Fernández (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 27:1–27:23. doi:10.4230/LIPIcs.FSCD.2025.27

Yiyun Liu, Jonathan Chan, Jessica Shi, and Stephanie Weirich. 2024b. Internalizing Indistinguishability with Dependent Types. *Proc. ACM Program. Lang.* 8, POPL, Article 44 (Jan. 2024), 28 pages. doi:10.1145/3632886

Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2024a. Functional Pearl: Short and Mechanized Logical Relation for Dependent Type Theories. (2024).

Yiyun Liu, Jonathan Chan, and Stephanie Weirich. 2025. Consistency of a Dependent Calculus of Indistinguishability. *Proc. ACM Program. Lang.* 9, POPL, Article 7 (Jan. 2025), 27 pages. doi:10.1145/3704843

Zhaohui Luo. 1990. *An extended calculus of constructions.* Ph. D. Dissertation. University of Edinburgh.

Alexandre Miquel. 2001. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. In *Typed Lambda Calculi and Applications*, Samson Abramsky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–359. doi:10.1007/3-540-45413-6_27

Alexandre Miquel and Benjamin Werner. 2003. The Not So Simple Proof-Irrelevant Model of CC. In *Types for Proofs and Programs*, Herman Geuvers and Freek Wiedijk (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 240–258.

Vincent Siles and Hugo Herbelin. 2012. Pure type system conversion is always typable. *Journal of Functional Programming* 22, 2 (2012), 153–180. doi:10.1017/S0956796812000044

Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. 2019. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 8 (Dec. 2019), 28 pages. doi:10.1145/3371076

Jonathan Sterling and Carlo Angiuli. 2021. Normalization for cubical type theory. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–15.

Masako Takahashi. 1995. Parallel Reductions in λ-Calculus. *Information and Computation* 118, 1 (1995), 120–127. doi:10.1006/inco.1995.1057

The Coq Development Team. 2024. The Coq Proof Assistant. doi:10.5281/zenodo.14542673

Amin Timany and Matthieu Sozeau. 2017. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC).* Research Report RR-9105. KU Leuven, Belgium ; Inria Paris. 30 pages. https://inria.hal.science/hal-01615123

Femke Van Raamsdonk and PG Severi. 1995. On normalisation. (1995).

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. doi:10.1145/3110275

Paweł Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 266–279. doi:10.1145/3167091