What Does It Take to Certify a Conversion Checker?

Meven Lennon-Bertrand
University of Cambridge, UK

Abstract

We report on a detailed exploration of the properties of conversion (definitional equality) in dependent type theory, with the goal of certifying decision procedures for it. While in that context the property of normalisation has attracted the most light, we instead emphasize the importance of *injectivity* properties, showing that they alone are both crucial and sufficient to certify most desirable properties of conversion checkers. We also explore the certification of a fully untyped conversion checker, with respect to a typed specification, and show that the story is mostly unchanged, although the exact injectivity properties needed are subtly different.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures; Software and its engineering → Data types and structures

Keywords and phrases Dependent types, Bidirectional typing, Certified software

Digital Object Identifier 10.4230/LIPIcs.FSCD.2025.27

Related Version *Extended version, with complete typing rules*: `https://arxiv.org/abs/2502.15500` [34]

Supplementary Material *Software (GitHub)*: `https://github.com/CoqHott/logrel-coq/tree/fscd25` [29], archived at `swh:1:dir:175ef91ad4724a9348081efb37ae93dd8c9082ba`
Software (Zenodo): `https://zenodo.org/records/15359259`

Funding This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277).

Acknowledgements Many thanks to Kenji Maillard and Thibaut Benjamin for their feedback on drafts of this paper, and to the anonymous reviewers for their helpful comments.

1 Introduction

Certifying the certifier. Proof assistants are more and more popular tools, that have now been used to formally certify many critical software systems and libraries. The trust in these relies on the proof assistant, which itself generally concentrates trust on a *kernel*, a well-delimited and relatively simple piece of the system, in charge of (re)checking all proofs generated outside it. This so-called de Bruijn criterion [9] is a first step toward safety. However, how can we make the kernel, the keystone of the whole ecosystem, trustworthy?

We should certify the kernel itself! Since it is small and well specified, this surely should not be hard? In higher order logic, Candle [4] already provides such a certified kernel for HOL Light. An equivalent achievement for dependent type theory is however still missing, despite 30 years of efforts since Barras [10], and the recent Agda-Core [37], Lean4Lean [13] and MetaCoq [45] projects. The reason is that kernels for dependent type theories, although relatively simple, rely on many invariants. Establishing these invariants requires many properties of the type system that the kernel is supposed to decide.

Yet, showing these properties is a tremendous undertaking,[1] when not altogether impossible: due to Gödel's second incompleteness theorem, any property implying a system's consistency cannot be proven in the system itself. This applies in particular to termination of the kernel, which relies on the infamous *normalisation* property, which in turn implies consistency. The full certification of a dependently-typed proof assistant in itself is thus unfeasible. But what can be salvaged? We should aim for graceful failure: even if we cannot reach normalisation, we would like to certify *something* about our type checker, and identify the key *meta-theoretic* properties of the type system necessary for this.

By clearly divorcing the meta-theory, taken as an opaque toolbox, from the certification of the algorithm, we obtain a good separation of concerns. This should help in getting bolder on *both* sides: we can modularly adopt advanced proof techniques on the meta-theory side [47, 12], and independently aim for the certification of complex, optimised implementations on the certification one [18, 27]. For this, it is crucial that the meta-theory is not tailored to a particular implementation, and vice-versa.

Let us also emphasise that, while a fully certified kernel proves that typing is decidable, mere decidability is in this view not central: we care about the actual content of the typing algorithm(s). This is even clearer in case we cannot – or do not wish to – prove normalisation, and therefore decidability. We still want to talk about a typing algorithm, which cannot be extracted from a decidability proof that does not exist.

Injectivity. As often with dependent types, the main source of meta-theoretic complications is conversion, the equational theory up to which types are compared, which we write $\cong$.[2] Out of the many properties that can be demanded of it, we want to put forward *injectivity* – as well as the accompanying *no-confusion*. For the Π type constructor, injectivity says that if $\Pi\, x\colon A.B \cong \Pi\, x\colon A'.B'$, then $A \cong A'$ and $B \cong B'$, while no-confusion says, for instance, that $\Pi\, x\colon A.B \cong \mathbb{N}$ is impossible. These cannot be shown by mere inversion of the conversion derivation, which can go through a long chain of transitivities. Injectivity shortcuts this, saying that, despite transitivity, certain congruences are invertible. This is a key aspect of intentional type theories, whose conversion is sufficiently restricted – injectivity of type constructors fails in extensional type theories [52].

What makes injectivity attractive is that despite its importance it does not imply logical consistency – although no-confusion implies *equational consistency*: not all equations hold. Consequently, and in contrast with normalisation and its Gödelian limitations, it can be proven in a meta-theory weaker than the object theory [49, 45, 17].

Untyped conversion. To better understand the relation between meta-theory and checker, we actually explore two conversion checking algorithms. The first is a type-directed one, similar to that already certified by Adjedj *et al.* [7], and Abel *et al.* [3] before them. Our main addition here is to more finely decompose this certification. However, this algorithm is not really faithful to the kernels of proof assistants: in ROCQ and LEAN, conversion does not maintain any type information, and AGDA's conversion, while primarily type-directed, similarly uses term-directed η-expansion of functions. We thus attack the novel certification of an untyped conversion checker, with η-laws.

Importantly, the specification remains typed: *specifying* η-laws in an untyped way is perilous [33], although their *implementation* is reasonable [16], as we verify. An important takeaway of our work is that, in a sense, talking about untyped conversion is misleading:

[1] METACOQ has already exceeded the 200kLoC, and its meta-theory is far from exhaustive.
[2] This is a typed judgment, although we sometimes omit the context and/or type in informal explanations.

	Positive soundness	Negative soundness (typed conversion)	Negative soundness (untyped conversion)	Termination
Injectivity of type constructors	×	×	×	×
Term-level injectivities		×	×	
Normalisation				×

Figure 1 Sufficient meta-theoretic properties for certification: to show a property of the algorithms (column) it suffices to have the properties of the type systems ticked in the corresponding lines. For instance, for positive soundness, injectivity of type constructors is sufficient.

we should rather be talking about *term-directed typed conversion*. We are not the first to try and relate typed and untyped conversion [44, 2], although we hope to demonstrate that concentrating on algorithms makes the proof of equivalence pleasantly straightforward compared to those previous works.

Certified decision procedures. We have introduced the main meta-theoretic properties we focus on: normalisation and injectivity. On the other side, what do we want to certify of the type/conversion checker? Abstractly, these take some input data, and hopefully return either a positive answer – and possibly some data associated to it, *e.g.* an inferred type – or a negative answer. We say "hopefully" because conversion checking is non-structurally recursive, and thus a priori partial. Given such a potentially partial procedure p, supposed to correspond to a proposition P, we can decompose its correctness into

- *positive soundness*: if p answers positively, then P holds;
- *negative soundness*: if p answers negatively, then $\neg P$ holds;
- *termination*: p always answers in one of these two ways.

Together, they imply that p is a decision procedure for P, in particular we have *completeness*: if P holds then p answers positively. We show that injectivity of type constructors is enough to prove positive soundness; with extra injectivity properties for terms, which differ slightly between the two conversion-checking algorithms, we establish negative soundness; instead, adding normalisation, we obtain termination.

Summary of results. Building on a previous formalisation [7] in the Rocq proof assistant [50] (previously known as Coq), we certify the properties gathered in Fig. 1, for two typing algorithms, using respectively type-directed and untyped conversion checking, for a version of Martin-Löf type theory (MLTT).[3] The rest of this paper is organised as follows:

- Sec. 2 summarises the declarative specification and its properties;
- Sec. 3 presents the algorithms we certify;
- Sec. 4 explores their certification and the relation of typed and untyped conversion;
- Sec. 5 concludes with related and future work.

[3] Featuring dependent function (Π) and pair (Σ) types with their η-laws, a universe, an empty type, natural numbers, and an identity type, all with large elimination.

$$\begin{array}{lcll}
\boxed{\mathrm{can}\, f} & := & \mathcal{U} \mid \Pi\, x{:}\, t.t \mid \lambda\, x{:}\, t.t \mid \mathbb{N} \mid 0 \mid \mathrm{S}(t) & \text{weak-head canonical forms} \\
\boxed{\mathrm{ne}\, n} & := & x \mid n\ t \mid \mathrm{rec}_{\mathbb{N}}(n, x.t, t, x.y.t) & \text{weak-head neutrals} \\
\boxed{\mathrm{nf}\, f} & := & \mathrm{can}\, f \vee \mathrm{ne}\, f & \text{weak-head normal forms} \\
\boxed{\mathrm{isTy}\, t} & := & \mathcal{U} \mid \Pi\, x{:}\, t.t \mid \mathbb{N} \mid n & \text{canonical types}
\end{array}$$

Figure 2 Canonical, neutral, and normal form predicates (excerpt, see [34]). Way to read the BNF-style grammars: isTy is the predicate on terms given by the grammar $\mathcal{U} \mid \Pi\, x{:}\, t.t \mid \mathbb{N} \mid n$ (n stands for a neutral and t for an arbitrary term).

Our formalisation adds roughly 10kLoC to the original code base. Links to it are indicated by this symbol: . Full versions of inference rules and other definitions are given either in the appendix or in the extended version [34].

2 The declarative system and its properties

2.1 Definitions

The type system. Our declarative type system and specification is a version of Martin-Löf's type theory [40] with its five mutually defined judgments: typing of types, terms, and contexts and conversion (definitional equality) of types and terms, written $\cong$. They are defined as mutual inductive predicates on "raw" objects. We use $=$ to denote meta-level equality – which coincides α-equality, as our representation uses pure de Bruijn indices.

The language supports dependent function (Π) and pair (Σ) types, which we call "negative", with their β and η-rules. We also include inductive types, with dependent large eliminators: an empty type $\bot$, natural number type $\mathbb{N}$, and an identity type $\mathbb{I}\mathrm{d}$. Finally, we have a universe $\mathcal{U}$, with codes for all type formers except itself. We refer to these types as "positive". Although our work does not directly involve focusing or logical polarity, where the positive/negative distinction stems from, it will still be an important distinction in the way our conversion algorithms work. We focus on $\mathcal{U}$, Π and $\mathbb{N}$ as representative examples on paper and direct interested readers to the code for others.

The development relies on parallel substitution, written $t[\sigma]$. If u is a term, we also denote as u the substitution mapping the last variable in context to u and leaving all others untouched, as used in *e.g.* β-reduction: $(\lambda\, x{:}\, A.t)\ u \rightsquigarrow^{\star} t[u]$. In Rocq, we use AutoSubst 2 [46, 19] to generate boilerplate for the substitution calculus and automate tedious proofs with a decision procedure for meta-level equality of expressions in this calculus.

Normal forms and reduction. Amongst terms, we particularly focus on (weak-head) normal forms, as this is the class with good injectivity properties, defined in Fig. 2. *Canonical forms* are terms starting with a constructor: S, λ, etc. *Neutral forms* are "stuck" terms, consisting of a spine of eliminators on top of a variable. *Normal forms* are either. We also define subclasses for each type: isFun t holds if t is either a neutral or a λ-abstraction, and similarly for isTy and isNat. The isTy_{+} predicate isolates positive types: $\mathbb{N}$, $\mathcal{U}$ and neutrals.

Finally, reduction $t \rightsquigarrow^{\star} t'$ is the reflexive, transitive closure of one-step reduction, defined in Fig. 3. We use only a deterministic *weak-head* reduction. Reduction is *not* part of the specification of typing, we present it at this stage only to state normalisation in Sec. 2.2.

$\boxed{t \rightsquigarrow t'}$ Term t weak-head reduces in one step to term t'

$$\beta\textsc{Fun}\ \frac{}{(\lambda\, x \colon A.t)\ u \rightsquigarrow t[u]} \qquad \beta\textsc{Zero}\ \frac{}{\mathrm{rec}_{\mathbb{N}}(0, x.P, b_0, x.y.b_{\mathrm{S}}) \rightsquigarrow b_0} \qquad \textsc{AppRed}\ \frac{t \rightsquigarrow t'}{t\ u \rightsquigarrow t'\ u} \qquad \ldots$$

Figure 3 Weak-head reduction (excerpt, see [34]).

2.2 Meta-theory

Let us now *state* the main meta-theoretic properties we consider – we comment about how to *prove* them at the end of this section and in Sec. 5. In Rocq, these are represented as type-classes: most theorems take instances of the relevant type-classes as parameters. This enables fine-grained bookkeeping of the theorems' dependency, while keeping it lightweight as instances are automatically filled in. This is inspired by the treatment of function extensionality and univalence in the HoTT library [11].

Type-level injectivity.

▶ **Property 1** (Injectivity and no-confusion of type constructors)**.** *Assume T and T' are convertible types in weak-head normal form, i.e. we have that* $\mathrm{isTy}\, T$, $\mathrm{isTy}\, T'$, *and* $\Gamma \vdash T \cong T'$. *Then one of the following holds:*

- $T = \mathbb{N} = T'$ *or* $T = \mathcal{U} = T'$;
- $T = \Pi\, x \colon A.B$, $T' = \Pi\, x \colon A'.B'$, *with* $\Gamma \vdash A' \cong A$ *and* $\Gamma, x \colon A' \vdash B \cong B'$;
- T, T' *are both neutral, and* $\Gamma \vdash T \cong T' : \mathcal{U}$.

Any other case is thus impossible (no-confusion)*, e.g. we cannot have* $\Gamma \vdash \Pi\, x \colon A.B \cong \mathbb{N}$.

This cannot be proven by mere induction, and indeed it implies a form of non-degeneracy, namely *equational consistency*: not all types are convertible. Consequences are numerous.

▶ **Corollary 2** (Preservation)**.** *Assuming Property 1, if* $\Gamma \vdash A$ *and* $A \rightsquigarrow^{\star} A'$ *then* $\Gamma \vdash A \cong A'$. *Similarly, if* $\Gamma \vdash t : A$ *and* $t \rightsquigarrow^{\star} t'$ *then* $\Gamma \vdash t \cong t' : A$.

▶ **Corollary 3** (Types classify normal forms)**.** *Assuming Property 1, consider t a normal form. If* $\Gamma \vdash t : \Pi\, x \colon A.B$, *then* $\mathrm{isFun}\, t$. *If* $\Gamma \vdash t : \mathbb{N}$, *then* $\mathrm{isNat}\, t$. *If* $\Gamma \vdash t : \mathcal{U}$, *then* $\mathrm{isTy}\, t$. *If* $\Gamma \vdash t : T$ *and T is neutral, then t is neutral.*

Preservation relies on the injectivity part, while classification uses no-confusion. Corollary 3 is the key lemma towards progress – the fact that every well-typed term is either a normal form or reduces –, although we do not show this in the formalisation. Thus, Property 1 suffices to establish safety in the progress-and-preservation approach [54].

Term-level injectivities. Statements of injectivity (and no-confusion) for term constructors are relatively close to those for types.

▶ **Property 4** (Injectivity and no-confusion at $\mathbb{N}$)**.** *Assume n and n' are such that* $\mathrm{isNat}\, n$, $\mathrm{isNat}\, n'$, *and* $\Gamma \vdash n \cong n' : \mathbb{N}$. *Then one of the following must hold:*

- $n = 0 = n'$;
- $n = \mathrm{S}(t)$, $n' = \mathrm{S}(t')$, *with* $\Gamma \vdash t \cong t' : \mathbb{N}$;
- n, n' *are both neutral.*

Maybe surprisingly, the eliminator makes these statements directly provable. Indeed, assuming $\mathrm{S}(t) \cong \mathrm{S}(t')$, by the computation and congruence rule for the eliminator, we get that $t \cong \mathrm{rec}_{\mathbb{N}}(x.\, \mathbb{N}, \mathrm{S}(t), 0, x.y.x) \cong \mathrm{rec}_{\mathbb{N}}(x.\, \mathbb{N}, \mathrm{S}(t'), 0, x.y.x) \cong t' : \mathbb{N}$. We can similarly reduce no-confusion to that for types. A similar phenomenon happens at Π types, where injectivity of λ can also be directly derived.

Injectivity for the universe () follows a similar pattern, but is not so easily proven for lack of an eliminator. For neutral types, which have no constructors, Corollary 3 is all we need: two (convertible) normal forms at a neutral type must be neutrals.

Neutral injectivities. To complete the injectivity picture, we turn to neutrals, for which injectivity/no-confusion can be expressed in two ways. The first follows a familiar pattern.

▶ **Property 5** (Injectivity of neutral eliminators)**.** *Assume n and n' are neutrals such that $\Gamma \vdash n \cong n' : T$. Then one of the following must hold:*

- *$n = x = n'$, with $(x : T') \in \Gamma$ and $\Gamma \vdash T' \cong T$;*
- *$n = m\ u$, $n' = m'\ u'$ with $\Gamma \vdash m \cong m' : \Pi\, x : A.B$, $\Gamma \vdash u \cong u' : A$ and $\Gamma \vdash B[u] \cong T$;*
- *$n = \mathrm{rec}_{\mathbb{N}}(m, x.P, t_0, x.y.t_{\mathrm{S}})$, $n' = \mathrm{rec}_{\mathbb{N}}(m', x.P', t'_0, x.y.t'_{\mathrm{S}})$, with convertible subterms.*

However, in some cases this is too strong: at negative types our type-directed algorithm eagerly η-expands, so the above are only really needed at positive types. To state the “minimal” property we require, we take another way, starting with the following definition.

▶ **Definition 6** (Neutral comparison)**.** Neutral comparison *(see Sec. A.2), written $\Gamma \vdash n \sim n' : T$, is the least relation that is:*

- *reflexive on variables, i.e. $\Gamma \vdash x \sim x : T$ if $(x : T) \in \Gamma$;*
- *closed under the congruence rules for eliminators;*
- *stable under conversion: if $\Gamma \vdash n \sim n' : T$ and $\Gamma \vdash T \cong T'$ then $\Gamma \vdash n \sim n' : T'$.*

We can then rephrase injectivity of neutral constructors as follows.

▶ **Property 7** (Completeness of neutral comparison)**.** *Given any two neutrals n and n' such that $\Gamma \vdash n \cong n' : T$, we have $\Gamma \vdash n \sim n' : T$.*

▶ **Proposition 8** ()**.** *Property 5 and Property 7 are equivalent.*

But now, we can weaken it to only consider *positive types.*

▶ **Property 9** (Completeness of neutral comparison at positive types)**.** *Given any two neutrals n and n' such that $\Gamma \vdash n \cong n' : T$, if moreover $\mathrm{isTy}_{+}\, T$, then we have $\Gamma \vdash n \sim n' : T$.*

Normalisation. The last property is normalisation. Its main use is to argue for termination of the algorithm, so it has to go beyond weak-head reduction and take into account η-expansion, and subterms of (weak-head) normal forms.

▶ **Definition 10** (Deeply normalising)**.** *A term t is* deeply normalising *at type A in context Γ if they reduce respectively to weak-head normal forms t' and A', and moreover:*

- *A' is a negative type, and the η-expansion of t' is itself normalising;*
- *or A' is a positive type, and the subterms of t' are normalising at the relevant types.*

A type A is deeply normalising *in context Γ if it reduces to a weak-head normal form, whose subterms are themselves normalising. See [34] for the inference rules.*

While not apparent in this short version of the definition, the context actually plays a role in order to express the type at which a neutral should be normalising.

▶ **Property 11** (Deep normalisation). *Every well-typed term is deeply normalising (at its type). Every well-formed type is deeply normalising.*

Meta-theoretic consequences include canonicity () – any closed term of type $\mathbb{N}$ is convertible to one of the form $\mathrm{S}^n\,0$ – and, most importantly, logical consistency. This means that to prove Property 11, the meta-theory must be logically stronger than the object theory.

▶ **Corollary 12** (Logical consistency). *Assuming Property 11, there are no closed terms of type $\bot$.*

Proving the properties. In our formalisation, most properties are proven via a logical relation, in the style pioneered by Abel *et al.* [3], and later adapted in Rocq [7], to which we refer for details. Multiple universes in the meta-theory provide the logical strength we need to prove consistency. This approach is however by far not the only possibility, see the discussion in Sec. 5. Note the following two points.

First, the proof of deep normalisation is indirect. Indeed, the logical relation is parameterised by an abstract partial equivalence relation (PER) relating neutrals, which we instantiate with algorithmic neutral comparison to obtain normalisation for subterms of neutrals. To avoid this detour through algorithmic aspects, we envisioned strengthening the logical relation to directly encompass neutrals, but did not find a satisfying approach.

Second, as logical relations at negative types are naturally defined by observations (*e.g.* a function is reducible when its application to reducible inputs are reducible), these types "know nothing" about neutrals, and completeness of neutral comparison at negative types cannot be derived easily. We expect other proofs of "semantic" flavour to have similar issues with full completeness of neutrals, making the restriction to positive types (Property 9 vs. Property 7) an important distinction. And indeed, while completeness at positive types is a direct consequence of our logical relation (), the approach we found to completeness at all types takes a long detour via the two algorithmic presentations () and strengthening.

3 Algorithmic typing and conversion

Let us now turn to our algorithms for conversion and typing. Only the untyped conversion algorithm is new, the typed conversion and bidirectional typing ones come directly from MLTT *à la* Coq [7]. We give the algorithms informally as rules, which are translated in the formalisation in two ways. First () as inductively defined relations, like declarative typing. Second, we implement them as functions (). As we wish to separate their definition from their termination argument, we must embrace partiality. To that end, we use McBride's free recursion monad [41], via Winterhalter's PartialFun [53]. This lets us define the functions prior to any proofs; immediately execute them with bounded recursion depth – McBride's "petrol semantics"; extract a typing (dis)proof from a terminating execution once we prove either soundness; and get proper decidability once we show normalisation. Cherry on the cake: as the recursion monad exposes the programs' recursive structure, we can generically derive "functional" induction principles, following said recursive structures.

Bidirectional typing. Our typing algorithm, as most for dependent types, is bidirectional: it separates *inference*, written $\Gamma \vdash t \rhd T$, where the type is an output to be found, from type *checking*, written $\Gamma \vdash t \lhd T$, where the type is a constraint given as input. We refer to others [42, 20, 31] for exposition – this last reference is the closest to us. Bidirectional typing generally has two main interests, although only the first is relevant to us here.

$\boxed{\Gamma \vdash t \cong_{\mathrm{h}} t' \lhd A}$ Reduced terms t and t' are convertible at type A

$$\textsc{TFun}\ \frac{\Gamma \vdash A \cong A' \lhd \mathcal{U} \qquad \Gamma, x\colon A' \vdash B \cong B' \lhd \mathcal{U}}{\Gamma \vdash \Pi\, x\colon A.B \cong_{\mathrm{h}} \Pi\, x\colon A'.B' \lhd \mathcal{U}} \qquad \textsc{FunExp}\ \frac{\Gamma, x\colon A \vdash f\ x \cong f'\ x \lhd B}{\Gamma \vdash f \cong_{\mathrm{h}} f' \lhd \Pi\, x\colon A.B}$$

$$\textsc{TSucc}\ \frac{\Gamma \vdash t \cong t' \lhd \mathbb{N}}{\Gamma \vdash \mathrm{S}(t) \cong_{\mathrm{h}} \mathrm{S}(t') \lhd \mathbb{N}} \qquad \textsc{NePos}\ \frac{\mathrm{isTy}_{+}\, T \qquad \Gamma \vdash n \sim n' \rhd S}{\Gamma \vdash n \cong_{\mathrm{h}} n' \lhd T} \qquad \ldots$$

$\boxed{\Gamma \vdash t \cong t' \lhd A}$ Terms t and t' are convertible at type T

$\boxed{\Gamma \vdash t \sim_{\mathrm{h}} t' \rhd T}$ Neutrals t and t' are comparable, inferring the reduced type T

$$\textsc{TTmRed}\ \frac{\begin{array}{c} T \leadsto^{\star} U \qquad t \leadsto^{\star} u \qquad t' \leadsto^{\star} u' \\ \Gamma \vdash u \cong_{\mathrm{h}} u' \lhd U \end{array}}{\Gamma \vdash t \cong t' \lhd T} \qquad \textsc{NRed}\ \frac{\Gamma \vdash n \sim n' \rhd T \qquad T \leadsto^{\star} S}{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd S}$$

$\boxed{\Gamma \vdash t \sim t' \rhd T}$ Neutrals t and t' are comparable, inferring the type T

$$\textsc{TVar}\ \frac{(x\colon T) \in \Gamma}{\Gamma \vdash x \sim x \rhd T} \qquad \textsc{TApp}\ \frac{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \Pi\, x\colon A.B \qquad \Gamma \vdash u \cong u' \lhd A}{\Gamma \vdash n\ u \sim n'\ u' \rhd B[u]} \qquad \ldots$$

Figure 4 Typed conversion algorithm (excerpt, see Sec. B.1).

First, bidirectional typing gives better control over conversion. By replacing the conversion rule with syntax-directed rules, we get a term-directed system, amenable to implementation. This makes the bidirectional system *a priori* more restrictive than the declarative one, since we have strictly constrained the way conversion can be used. The main point of negative soundness is to show that this restriction does not, in fact, lose anything.

Second, propagating types "up" in derivations lightens the need for type annotations and gives better error messages. This is important for surface languages, less for kernels, where terms are typically not human-written. On the contrary, extra information can be useful in various ways, and we would rather dispense of the complication of segregating terms containing too little information to infer. Hence, as Rocq and Lean, we adopt a syntax where all terms infer, although completeness for the other approach is certainly interesting and feasible [35].

Typed conversion is bidirectional. A point already implicitly present in Abel *et al.* [3], but made explicit in Adjedj *et al.* [7], is that the type-directed conversion algorithm is bidirectional, too. It is decomposed in two main functions, one to compare general terms and the other dedicated to neutrals. The former takes a type as input – it is *checking* –, while the latter reconstructs types as it traverses the neutrals – it is *inferring*.

More precisely, conversion (see Fig. 4), written $\Gamma \vdash t \cong u \lhd A$, goes as follows:

1. reduce t, u and A to weak-head normal forms;
2. if A is a negative type (Π, Σ), recursively compare the η-expansions of t and u;
3. if A is a positive type $(\mathbb{N}, \mathcal{U}, \ldots)$ and the terms are canonical, use the relevant congruence;
4. if A is a positive type and the terms are neutrals, call neutral comparison.

$\boxed{t \cong t'}$ Terms t and t' are convertible

$$\textsc{UTmRed}\ \frac{t \leadsto^\star u \qquad t' \leadsto^\star u' \qquad u \cong_{\mathrm{h}} u'}{t \cong t'}$$

$\boxed{t \cong_{\mathrm{h}} t'}$ Reduced terms t and t' are convertible

$$\textsc{UUni}\ \frac{}{\mathcal{U} \cong_{\mathrm{h}} \mathcal{U}} \qquad \textsc{UFun}\ \frac{A \cong A' \qquad B \cong B'}{\Pi\, x\colon A.B \cong_{\mathrm{h}} \Pi\, x\colon A'.B'} \qquad \textsc{ULamLam}\ \frac{t \cong t'}{\lambda\, x\colon A.t \cong_{\mathrm{h}} \lambda\, x\colon A'.t'}$$

$$\textsc{ULamNe}\ \frac{\mathrm{ne}\, n' \qquad t \cong n'\ x}{\lambda\, x\colon A.t \cong_{\mathrm{h}} n'} \qquad \textsc{USucc}\ \frac{t \cong t'}{\mathrm{S}(t) \cong_{\mathrm{h}} \mathrm{S}(t')} \qquad \textsc{UNeNe}\ \frac{n \sim n'}{n \cong_{\mathrm{h}} n'} \qquad \ldots$$

$\boxed{n \sim n'}$ Neutrals n and n' are comparable

$$\textsc{UVar}\ \frac{}{x \sim x} \qquad \textsc{UApp}\ \frac{n \sim n' \qquad u \cong u'}{n\ u \sim n'\ u'} \qquad \ldots$$

Figure 5 Untyped algorithmic conversion (excerpt, see Sec. B.2).

Steps 2-4 are delegated to an auxiliary function, denoted $\cong_{\mathrm{h}}$ (the "h" is for "head"). Neutral comparison, written $\Gamma \vdash n \sim n' \rhd S$ follows their structure, applying congruences as it goes.

Rule TApp demonstrates this bidirectional yoga. As conversion is type-directed, we need a type A to compare the arguments, that is obtained through the recursive comparison of the functions, which must thus be inferring. We would not be able to construct $\Pi\, x\colon A.B$ anyway, even given $B[u]$: inverting substitutions is unfeasible. Thus, even with a syntax with complete inference, the distinction between checking constructors and inferring eliminators, a landmark of the "Nordic" approach to bidirectionalism [43, 42], appears naturally.

Untyped conversion. Untyped conversion (Fig. 5) looks much simpler than typed conversion: there are no types, no separation between type and term conversion, and the separation of neutral comparison is kept mostly for readability.

The main point is that we replace the all-encompassing FunExp by term-directed rules, an approach pioneered by Coquand, first for functions [16], then pairs [2]. When comparing λ-abstractions, we apply the congruence ULamLam – which would also happen in FunExp after contracting the redexes. If one side is a λ and the other a neutral (ULamNe), we again simulate FunExp, but this time we see the application to a fresh variable is visible on the neutral's side. And this is all! That is, *there is no η-rule when the two sides are neutrals.* How could there be? In the extreme case of two variables, the only information we could use is... type information, which we precisely decided not to rely on. This discrepancy on neutrals at negative types is the key difference between the two algorithms.

Comparison. As a concrete example, we can look at the "execution trace" in the following problem: $x\colon \mathbb{N} \to \mathbb{N} \vdash x \cong x : \mathbb{N} \to \mathbb{N}$. The untyped algorithm immediately uses UNeNe to go in "neutral mode", and finishes with UVar. The typed algorithm, on the other hand, first η-expands, introducing a fresh variable y, and recursively compares $x\ y$ to $x\ y$. Only then, by reaching a positive type, does it change to "neutral mode". Neutral comparison then peels off the application node, compares x to x again but this time as neutrals, and finally

Raw judgment		Precondition		Postcondition
$\Gamma \vdash t \rhd T$	$\wedge$	$\vdash \Gamma$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash t : T \ \wedge\ T$ unique
$\Gamma \vdash t \lhd T$	$\wedge$	$\Gamma \vdash T$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash t : T$
$\Gamma \vdash T \cong T' \lhd$	$\wedge$	$\Gamma \vdash T \ \wedge\ \Gamma \vdash T'$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash T \cong T'$
$\Gamma \vdash t \cong t' \lhd T$	$\wedge$	$\Gamma \vdash t : T \ \wedge\ \Gamma \vdash t' : T$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash t \cong t : T$
$\Gamma \vdash t \sim t' \rhd T$	$\wedge$	$\mathrm{ne}\, t \ \wedge\ \exists A.\ \Gamma \vdash t : A \wedge$ $\mathrm{ne}\, t' \ \wedge\ \exists A'.\ \Gamma \vdash t' : A'$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash n \cong n' : T \wedge$ T unique
$T \cong T'$	$\wedge$	$\exists \Gamma.\ \Gamma \vdash T \ \wedge\ \Gamma \vdash T'$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash T \cong T'$
$t \cong t'$	$\wedge$	$\exists \Gamma\ T.\ \Gamma \vdash t : T \ \wedge\ \Gamma \vdash t' : T$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash t \cong t : T$
$t \sim t'$	$\wedge$	$\mathrm{ne}\, t \ \wedge\ \mathrm{ne}\, t' \wedge$ $\exists \Gamma\ A\ A'.\ \Gamma \vdash t : A \ \wedge\ \Gamma \vdash t' : A'$	$\overset{?}{\Rightarrow}$	$\Gamma \vdash n \cong n' : T \wedge$ T unique

Figure 6 Pre- and post-conditions of the algorithmic judgments.

applies variable reflexivity. It also needs to compare y to y first as normal forms at $\mathbb{N}$, then as neutrals, and finally terminates. We can already see the work spared by the term-directed algorithm – exacerbated for Σ types, where η introduces duplication.

4 Properties of the algorithms

An important piece is missing from the algorithmic judgments as defined so far: invariants. Indeed, there is a number of things that are not directly checked. For instance, TVar does not enforce context well-formation, and NePos does not check that the inferred type coincides with the input one. These are not unfortunate oversight; instead, we avoid re-doing work that can be obtained for free through invariants, which is crucial in implementations – constantly re-checking context well-formation, for instance, is terrible for performance.

Hence, each function has preconditions, which should be true before we even dare ask the question that the function ought to answer; and a postcondition, which is what it is in charge of establishing. These are summarised in Fig. 6. By "T unique" we mean that any other type also satisfying the same property must be convertible to T. "Reduced" judgments, *e.g.* $\Gamma \vdash T \cong_{\mathrm{h}} T' \lhd$, have the same postconditions as their unreduced counterparts, and the extra precondition that inputs should be weak-head normal forms.

As we can see, although no type is visible in the definition of untyped conversion, the name "untyped" is misleading. We still aim to decide the same typed relation, but have merely remarked that we can arrange the algorithm in a way that makes type information superfluous. Yet types are still present in invariants, silently keeping the algorithm on rails. Untyped conversion may thus be more aptly characterised as *term-directed typed conversion*.

4.1 Properties of typed conversion

Most results in this section are not *per se* new, although they were significantly reworked, in particular to precisely track their dependencies. In particular, termination relies explicitly on normalisation, rather than on the reflexivity of the logical relation. The work around negative soundness is entirely new.

Positive soundness. We split the proof in two, using the inductive judgment as intermediate. The first half is almost tautological, and does not depend on any meta-theory.

▶ **Proposition 13** (Positive soundness of the typed conversion function)**.** *If a typed conversion function returns a positive answer, then the corresponding judgment of Fig. 4 holds.*

▶ **Proposition 14** (Positive soundness of the typed algorithmic conversion judgment)**.** *Assuming injectivity of type constructors (Property 1), the algorithmic typed conversion judgments imply their postconditions whenever their preconditions hold.*

This second half is the interesting one. The crucial part of this proof, and the backbone of others in this section, is to show that each inductive rule correctly preserves the invariants. This is a very regular property, so much that we meta-programmed[4] the generation of "local preservation lemmas" () for each rule, expressing this idea. We assemble these lemmas into a "bundled induction principle", again programmatically generated, which threads the invariants through, giving access to them as extra hypotheses in each induction step. Soundness follows by instantiating this induction principle with the constant `True` predicates.

Negative soundness. For negative soundness, we cannot go through the intermediate algorithmic judgments, as they only encode successful runs of the functions. Thus, we bite the bullet and directly relate the output of the functions to the declarative judgments.

▶ **Proposition 15** (Negative soundness of typed algorithmic conversion)**.** *Assuming injectivity of type and term constructors (Properties 1, 4, etc.), and completeness of neutral conversion* at positive types *(Property 9), if a typed conversion function returns a negative answer and its precondition holds, then its postcondition cannot hold.*

Again, we rely on invariant preservation, the proof is otherwise straightforward: we have extracted exactly the injectivity properties needed to justify each step. In essence, we show that at any stage the current "remaining goals" are equivalent to the initial query, relying on injectivity to know that we only use invertible congruences, which are safe to apply.

Termination. As expected, this is where normalisation becomes necessary. As before, we need invariants throughout, and thus injectivity of type constructors. The most subtle part of the proof concerns the reduction machine.

▶ **Lemma 16** (Completeness of reduction)**.** *Assuming Property 1, if t is well-typed and reduces to the weak-head normal form t', then the reduction function applied to t returns t'.*

The main task is to define a well-founded order on the configurations of the abstract machine implementing reduction, a lexicographic product of subterm ordering and reduction. The assumption that t reduces to a weak-head normal form ensures that this order is well-founded. Winterhalter introduced a similar order for MetaCoq [52, chap. 23].

▶ **Proposition 17** (Termination of typed algorithmic conversion)**.** *Assuming injectivity of type constructors (Property 1) and deep normalisation (Property 11), typed conversion functions terminate whenever their preconditions are satisfied.*

The proof is by induction on deep normalisation, again using local preservation lemmas. The inductive structure of deep normalisation readily encodes the combination of subterm, reduction and η-expansion that one needs to be well-founded. In contrast, in MetaCoq, Winterhalter introduced a complex dependent lexicographic order modulo [52, chap 24], and admits that his setup would not easily incorporate η-expansion.

[4] We hardcoded these in the current code to remove a dependency which caused proof engineering issues.

Non-triviality. Soundness, both positive and negative, has a default: the nowhere-defined function is sound! Although we repeat that these properties should not be read to merely say “there exists a function such that...” but to characterise a particular checker – that we can run on examples () to check for non-triviality –, it is good to formally ensure that our functions are non-trivial. Again, this is split in two.

▶ **Proposition 18** (Completeness of the implementation)**.** *Assuming injectivity of type constructors (Property 1), if an algorithmic conversion judgment and its precondition hold, then the corresponding function answers positively.*

Next, we show that algorithmic conversion satisfies many closure properties, which Abel *et al.* call “generic conversion”. These are rather rich: all the proofs on the logical relation side can be done with respect to *any* generic conversion.

▶ **Proposition 19** ()**.** *Algorithmic conversion is a generic conversion, thus for instance symmetric and transitive, stable by weakening and expansion, congruent for constructors.*

Compared to declarative typing, the only missing part is congruences for eliminators, which only hold for neutral conversion, and reflexivity – which would be direct by induction if we had all congruence rules. The point is that eliminator congruences are the dangerous ones that can trigger reductions: applying a normal form to a normal form does not yield a normal form. And indeed a term is deeply normalising exactly when it is reflexive for (typed) algorithmic conversion, explaining why reflexivity cannot be achieved easily.

Typing. All of these propositions extend to bidirectional typing. More precisely, assuming *any* conversion relation/function between types that satisfies the relevant property, we can derive a similar one for typing. This means that the proofs are modular, in that we can swap out a different implementation of conversion, as long as it satisfies the right properties, which is exactly what we are about to do.

4.2 Properties of untyped conversion

The high-level ideas are very similar to typed conversion. For positive soundness there is really no difference: the equivalents of Propositions 13 () and 14 () hold. Even better, we can reuse most of the local preservation lemmas directly.

Negative soundness is where the subtle difference between the two algorithms shows. Echoing the discussion in Secs. 2.2 and 3, the key point is neutrals at negative types. And indeed, while in Proposition 15 we could make do with the weaker completeness (Property 9), this time we really need the full power of completeness at all types.

▶ **Proposition 20** (Negative soundness of untyped algorithmic conversion)**.** *Assuming injectivity of type and term constructors (Properties 1, 4, etc.), and completeness of neutral conversion* at all types *(Property 7), if the untyped conversion function returns a negative answer and its preconditions holds, then its postcondition cannot hold.*

As for termination, there is a twist again, of course at negative types. The issue is that deep normalisation systematically η-expands terms, but untyped conversion does not always do this. Worse, it is non-deterministic, in the sense that whether a given term is η-expanded depends on the term it is compared to. Thus, our termination measure is not exactly fit, and circumventing the issue involves proving a form of strengthening of the algorithm: it gives the same output when irrelevant variables are removed from the context.

▶ **Proposition 21** (Termination of untyped algorithmic conversion)**.** *Assuming Property 1 and Property 11, untyped conversion functions terminate whenever their preconditions hold.*

4.3 Equivalence between the algorithms

To wrap up, we can directly compare the two algorithms. Given how close they are, this should be rather easy, or at any rate easier than comparing declarative systems – where the seemingly innocuous transitivity is actually a source of headaches [44], because it means "real" untyped conversion can relate well-typed terms through ill-typed ones, which cannot be easily emulated by its typed counterpart. In fact, streamlining the relation between typed and untyped conversion was one of our original motivation [31, chap. 6] [32].

▶ **Proposition 22** (Typed to untyped conversion)**.** *Assuming Property 1, the typed algorithmic judgments imply their untyped counterparts whenever their preconditions hold.*

Proof. The proof is by bundled induction on the typed algorithmic judgment. We simultaneously prove that if $\Gamma \vdash n \cong n' \lhd T$ and moreover n, n' are neutrals, then $n \sim n'$.

The main interesting case is of course that of FunExp. In this case, the typing invariants imply, by Corollary 3, that both sides are either abstractions or neutrals. In the first three cases, we can directly conclude by induction hypothesis, up to β-reduction of the η-expanded abstraction. Remains the case of two neutrals n and n'. The extra induction hypothesis for neutrals tells us that $n\ y \sim n'\ y$, or more precisely in de Bruijn syntax, that $n[\uparrow]\ y \sim n'[\uparrow]\ y$, where $\uparrow$ is the substitution shifting all indices by 1. We can easily invert this to $n[\uparrow] \sim n'[\uparrow]$, and conclude by an auxiliary lemma that untyped conversion admits strengthening. ◀

Note the final use of strengthening: while this is readily proven by induction for algorithmic judgments, it is generally difficult to prove for declarative systems! Admitting easy proofs of strengthening is a major advantage of algorithmic presentations [30].

For the converse, there is a catch: normalisation is required. Indeed, imagine a type A, well-formed in Γ, such that there is an infinite reduction sequence $A \rightsquigarrow A_1 \rightsquigarrow \dots$ Trying to deduce $\Gamma, x \colon A \vdash x \cong x : A$, the untyped algorithm would terminate immediately, while the typed one would diverge on the evaluation of A. More insidiously, if a type B keeps on exposing Π constructors – say a solution to $B \cong \mathbb{N} \to B$ – comparing $x \colon B$ to itself would terminate immediately on one side, and spin in never ending η-expansions on the other. We can, however, prove the following crucial lemma.

▶ **Lemma 23** (Completeness of typed neutral conversion at all types)**.** *Assuming Property 11, given a context Γ and A, m, n respectively a well-formed type and well-typed terms in Γ, such that $\Gamma \vdash m \sim n \rhd A'$ and $\Gamma \vdash A' \cong A$, we have $\Gamma \vdash m \cong n \lhd A$.*

Proof sketch[5]**.** By induction on the deep normalisation of A, using which, after a series of η-expansions of m and n, we end up at a positive type. At this point we go into neutral mode, peel off all introduced η-expansions, and use our neutral comparison hypothesis, adequately weakened to account for the extra introduced variables, to conclude. ◀

▶ **Proposition 24** (Untyped to typed conversion)**.** *Assuming injectivity of typed constructors and deep normalisation, the untyped algorithmic judgments imply their typed counterparts whenever their preconditions hold.*

After Lemma 23, remains a straightforward induction – as usual, using preservation lemmas as necessary. However, let us emphasise that while the proof of equivalence is relatively direct, it nonetheless relies on difficult properties: strengthening, and normalisation.

[5] In the formalisation, we shortcut this proof using the equivalence of declarative and algorithmic typing.

4.4 Extensions and limits

While our language is simplified compared to real proof assistants, we believe that the features we include are enough to discover important subtleties, and that our general methodology would scale to the type theories of Rocq, Lean or Agda.[6] The main aspect that we did not cover is definitional uniqueness, *i.e.* types with one inhabitant up to conversion. Still, in the light of what precedes, we can comment on it.

Definitional unit. The η rule for the unit type says that any two inhabitant are convertible:

$$\textsc{Unit}\ \frac{\Gamma \vdash t : \top \qquad \Gamma \vdash t' : \top}{\Gamma \vdash t \cong t' : \top}$$

In a sense, this is the ultimate type-directed rule. Accordingly, it completely wrecks completeness of neutral comparison, since *any* two neutrals are convertible. This also "leaks" to other negative types via their own η-laws. For instance, $\mathbb{N} \to (\Sigma\, x \colon \top . \top)$ is also "unit-like", all its elements being convertible, and neutral comparison is likewise incomplete.

Different strategies are adopted to circumvent this issue in major proof assistants, where $\top$ typically appears as a record type with no fields. Agda is the least affected, as its conversion is type-directed, although the subtle behaviour of unit-like types is a regular source of bugs [8, 14, 15, 25]. Rocq instead commits to untyped conversion, but must therefore enforce all record types to have at least one relevant field, forbidding the definitional unit type. Lean takes an intermediate solution, re-inferring the type of neutrals on the fly if their untyped comparison fails, and implements a dedicated criterion to detect which types are "unit-like",[7] a strategy also explored by Kovács in normalisation by evaluation [28].

Strict propositions. The universe of strict propositions [21] – called `Prop` in Agda and Lean, and `SProp` in Rocq, which we write $\mathcal{P}$ –, reflects the idea of propositions being proof irrelevant: any two proofs are definitionally equal. Concretely, the rule is as follows:

$$\textsc{SProp}\ \frac{\Gamma \vdash t : P \qquad \Gamma \vdash t' : P \qquad \Gamma \vdash P : \mathcal{P}}{\Gamma \vdash t \cong t' : P}$$

This is very similar to definitional unit, and of course comes with similar threat to completeness of neutral comparison. Agda's conversion, being type-directed – and already maintaining universe information –, was easily extended. In Lean, the same strategy as for unit – re-inferring types in case of failure – is used. In combination with heavy sharing and little type-level computation, this seems enough to keep decent performances.

Rocq, however, does something special to remain purely term-directed [22, 36]. The core remark is that there is an important difference between SProp and Unit: computing types is in general costly – as one needs to compute substitutions, evaluate terms, etc. – but merely maintaining information about the *universe* is much cheaper. Moreover, since strict propositions form a universe of definitionally irrelevant types, by construction closed under as many operations as possible, the proliferation is contained to $\mathcal{P}$ only. For instance any function type with codomain in $\mathcal{P}$ is again in $\mathcal{P}$. Thus, universe information can be used to implement SProp in a cheap yet complete way, and this is the strategy used by Rocq.

[6] Barring complex extensions such as Cubical Agda, which would warrant further investigations.
[7] Although this criterion is currently incomplete, causing issues in the type-checker [6].

η and strengthening. In the example of Sec. 3, the type-directed algorithm introduces a fresh variable to the context while the untyped one does not. This discrepancy between the algorithms explains the need for strengthening in Sec. 2.2 and Proposition 22.

While we do not have an example of a type theory that would stress the equivalence by breaking strengthening, remark that in extensional type theory, where any two terms are convertible in an inconsistent context and strengthening fails, to deduce $x, y : \bot \to \mathbb{N} \vdash x \cong y : \bot \to \mathbb{N}$ it is necessary to go through η-expansion to introduce a variable $z : \bot$ in the context. Extensional type theory is undecidable, thus less interesting to us, but a similar phenomenon happens in cubical type theory. In Cubical Agda, elements of the type `Partial i0 nat` behave as functions `isOne i0 -> nat`, but are compared only on the geometric condition encoded by `isOne i0`. Since this condition is degenerate, any two variables of type `Partial i0 -> nat` are convertible, but only through a form of η-expansion. In other words, strengthening fails with respect to a variable of type `isOne i0`.

For this and similar reasons, as far as we can tell it is widely believed that cubical systems cannot be implemented using a purely term-directed conversion checker, and given this hard failure of strengthening and neutral completeness, we are inclined to follow this belief.

5 Conclusion, related and future work

Similar formalisations. We are neither the only, nor the first, to work on formalising the meta-theory of a dependent type system. These formalisations roughly fall into two groups. The first [3, 51, 23, 7, 24, 38] tackles systems roughly the same complexity as ours, and focus on meta-theory via varying forms of logical relations. They all prove everything all they need in one go via the logical relation, and thus do not attempt a detailed analysis of dependency between properties.

A second group focuses on describing and specifying realistic type checkers. Lean4Lean's [13] checker is on par with Lean's kernel, but only describes its intended type system without relating it to the checker, and develops only minimal meta-theory. Agda-Core [37] defines a complex system modelled after Agda, and provide a checker returning typing derivations, thus ensuring positive soundness by construction. They also do not consider meta-theory or properties beyond positive soundness. Only MetaCoq [45] develops a comprehensive meta-theory, for a type theory close to Rocq's, backing a certified sound, complete and terminating checker – up to normalisation, which is axiomatised. We hope that the present work clarifies the role of meta-theory in these projects: we encourage them to adopt the "meta-theory as black box" motto, and focus on kernel certification against a handful of axiomatised properties expected to hold – similar to MetaCoq's approach to normalisation.

Avoiding normalisation. A wealth of approaches exist to establish injectivity properties independently of normalisation. Confluence is powerful and scalable, as demonstrated by MetaCoq [45]. Parallel reduction has been adapted to typed conversion [5, 44] and η for functions and pairs [48], although neither work tackle the combination of the two. A different approach is taken by Coquand and Huber [17], who use a semantic in domains to obtain rich injectivity properties for a type theory very close to ours. The key point is that they carry out their construction in a weak ambient theory, although they rely on the stratification of universes, which shows that the logical power of injectivity properties is low.

Another approach to avoid normalisation is that of Abel and Altenkirch [1], who show a coinductive form of completeness which makes no promise about termination. They also put forward confluence, injectivity of type constructors, and *standardisation*, which gives a form of completeness of weak-head reduction: if a term is convertible to a weak-head normal form, it (weak-head) reduces to a weak-head normal form with the same shape. This also appeared naturally in our formalisation, although less prominently than injectivity.

Intrinsic typing. An alternative approach to meta-theory [47, 12] is based on "intrinsic" typing, where well-typed terms are defined directly as an initial algebra or quotient inductive-inductive type (QIIT) [26], letting one use powerful semantic tools to prove syntactic properties. We would like to yield this powerful hammer, yet important hurdles remain.

In the intrinsic approach, syntax is by construction quotiented by conversion, which makes it difficult to draw finer-grained distinctions. Thus, while expressing injectivity of type constructors is straightforward, neutral comparison, and thus Properties 7 and 9, is more problematic. Accordingly, these approaches rephrase decidability in a "reduction-free" manner. It seems difficult to argue about termination of our function based only on such conversion-invariant properties, we likely need something more intensional. A way out might be to reason about conversion proofs as elements of the meta-level identity type of object terms, although this is impossible with current QIITs, which are Set-truncated. A final issue is that we ultimately wish to certify an implementation acting on raw terms, rather than intrinsic syntax. Relating the two amounts to the initiality theorem [39], which is no small feat!

Future work. Given the centrality of injectivity properties, a natural future direction is to develop new techniques focused on them, especially ones that would scale to the complexity of real kernels. Even if a normalisation proof for Rocq, Lean or Agda is a major endeavour not likely to manifest soon – and will always have to circumvent Gödelian limitations –, this seems more reachable, although we are not here yet. Whether confluence-like techniques will suffice, or whether we will need more semantic approaches, remains to be explored.

On the other side of implementation, we have started a minimal exploration of the world of conversions, by showing we can tackle untyped conversion even against a typed specification. Our implementation is rather naïve, and PartialFun is currently not geared towards good extraction. Exploring the certification of efficient, optimised conversion checkers, inspired by those explored – and certified! – by Courant [18], or the work of Kovács on normalisation by evaluation [27] is also an interesting aspiration.

References

1 Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type:type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). doi:10.1016/j.entcs.2011.02.013.

2 Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue. URL: http://fi.mimuw.edu.pl/abs77.html#15.

3 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158111.

4 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A Verified Implementation of HOL Light. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving (ITP 2022)*, volume 237 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.ITP.2022.3.

5 Robin Adams. Pure type systems with judgemental equality. *J. Funct. Program.*, 16(2):219–246, 2006. doi:10.1017/S0956796805005770.

6 Arthur Adjedj. Issue #2258: Defeq transitivity failures for eta. GitHub issue. URL: https://github.com/leanprover/lean4/issues/2258.

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

8 Antoine Allioux. Issue #5174: Rewrite rules and eta-expansion of the unit type. GitHub issue. URL: https://github.com/agda/agda/issues/5174.

9 Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 1149–1238. North-Holland, Amsterdam, 2001. doi:10.1016/B978-044450813-3/50020-5.

10 Bruno Barras and Benjamin Werner. Coq in coq. Unpublished manuscript, 1997. URL: http://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf.

11 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The hott library: a formalization of homotopy type theory in coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172. Association for Computing Machinery, 2017. doi:10.1145/3018610.3018615.

12 Rafaël Bocquet, Ambrus Kaposi, and Christian Sattler. For the metatheory of type theory, internal sconing is enough. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023*, volume 260 of *LIPIcs*, pages 18:1–18:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.FSCD.2023.18.

13 Mario Carneiro. Lean4lean: Towards a formalized metatheory for the lean theorem prover, March 2024. doi:10.48550/arXiv.2403.14064.

14 Jesper Cockx. Issue #3785: Comparison of blocked terms doesn't respect eta. GitHub issue. URL: https://github.com/agda/agda/issues/3785.

15 Jesper Cockx. Issue #5837: Occurs check does not properly handle singleton type. GitHub issue. URL: https://github.com/agda/agda/issues/5837.

16 Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1), 1996. doi:10.1016/0167-6423(95)00021-6.

17 Thierry Coquand and Simon Huber. An adequacy theorem for dependent type theory. *Theory of Computing Systems*, 63(4):647–665, July 2018. doi:10.1007/s00224-018-9879-9.

18 Nathanaëlle Courant. *Towards an efficient and formally verified convertibility checker*. PhD thesis, Université Paris Cité, 2024.

19 Adrian Dapprich. Generating Infrastructural Code for Terms with Binders using MetaCoq and OCaml. Bachelor thesis, Saarland University, 2021.

20 Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5), May 2021. doi:10.1145/3450952.

21 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, January 2019. doi:10.1145/329031610.1145/3290316.

22 Gaëtan Gilbert. *A type theory with definitional proof-irrelevance*. PhD thesis, Université de Nantes, 2020.

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

24 Junyoung Jang, Jason Z. S. Hu, Antoine Gaulin, and Brigitte Pientka. Mctt: Building a correct-by-construction proof checkerfor martin-löf type theory. In *4th Workshop on the Implementation of Type Systems*, 2025.

25 Ambrus Kaposi. Issue #6417: Forward declaration breaks eta for unit type. GitHub issue. URL: `https://github.com/agda/agda/issues/6417`.

26 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proceeding of the ACM on Programming Languages*, 3(POPL), January 2019. `doi:10.1145/3290315`.

27 András Kovács. Efficient evaluation with controlled definition unfolding. In *3rd Workshop on the Implementation of Type Systems*, 2024.

28 András Kovács. Eta conversion for the unit type (is still not that simple). In *4th Workshop on the Implementation of Type Systems*, 2025.

29 Meven Lennon-Bertrand. LogRel-Coq – FSCD 2025. Software, version fscd25., swhId: `swh:1:dir:175ef91ad4724a9348081efb37ae93dd8c9082ba` (visited on 2025-06-23). URL: `https://github.com/CoqHott/logrel-coq/tree/fscd25`, `doi:10.4230/artifacts.23665`.

30 Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ITP.2021.24`.

31 Meven Lennon-Bertrand. *Bidirectional Typing for the Calculus of Inductive Constructions*. phdthesis, Nantes Université, 2022.

32 Meven Lennon-Bertrand. Equivalence between typed and untyped algorithmic conversions. In *28th International Conference on Types for Proofs and Programs*, June 2022. URL: `https://types22.inria.fr/programme/`.

33 Meven Lennon-Bertrand. À bas l'η – Coq's troublesome η-conversion. In *1st Workshop on the Implementation of Type Systems*, 2022.

34 Meven Lennon-Bertrand. What does it take to certify conversion?, 2025. `doi:10.48550/arXiv.2502.15500`.

35 Meven Lennon-Bertrand and Neel Krishnaswami. Decidable type-checking for bidirectional martin-löf type theory. In *29th International Conference on Types for Proofs and Programs*, 2023. URL: `https://types2023.webs.upv.es/`.

36 Yann Leray. Formalisation et implémentation des propositions strictes dans MetaCoq. Technical report, Inria Rennes - Bretagne Atlantique ; LS2N-Nantes Université, August 2022. URL: `https://inria.hal.science/hal-04433492`.

37 Bohdan Liesnikov and Jesper Cockx. Building a correct-by-construction type checker for a dependently typed core language. In Oleg Kiselyov, editor, *Programming Languages and Systems*, pages 63–83, Singapore, 2025. Springer Nature Singapore.

38 Yiyun Liu, Jonathan Chan, and Stephanie Weirich. Consistency of a dependent calculus of indistinguishability. *Proceeding of the ACM of Programming Languages*, 9(POPL), January 2025. `doi:10.1145/3704843`.

39 Peter LeFanu Lumsdaine, Guillaume Brunerie, Menno de Boer, and Anders Mörtberg. Initiality for martin-löf type theory. Talk at the HoTTest seminar, 2020.

40 Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Number 1 in Studies in Proof Theory. Napoli: Bibliopolis, 1984.

41 Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction*, pages 257–275, Cham, 2015. Springer International Publishing. `doi:10.1007/978-3-319-19797-5_13`.

42 Conor McBride. Basics of bidirectionalism. Blog post, August 2018. URL: `https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/`.

43 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, September 2007.

44 Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *J. Funct. Program.*, 22(2):153–180, 2012. `doi:10.1017/S0956796812000044`.

45 Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and complete type checking and certified erasure for coq, in coq. *Journal of the ACM*, November 2024. doi:10.1145/3706056.

46 Kathrin Stark. *Mechanising syntax with binders in Coq*. PhD thesis, Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/28822.

47 Jonathan Sterling. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory*. PhD thesis, Carnegie Mellon University, November 2021. Doctoral thesis of Jonathan Sterling, Carnegie Mellon University. doi:10.5281/zenodo.6990769.

48 Kristian Stoevring. Extending the extensional lambda calculus with surjective pairing is conservative. *Logical Methods in Computer Science*, Volume 2, Issue 2, 2006. doi:10.2168/LMCS-2(2:1)2006.

49 M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995. doi:10.1006/inco.1995.1057.

50 The Coq Development Team. The coq proof assistant, June 2024. doi:10.5281/zenodo.14542673.

51 Pawel Wieczorek and Dariusz Biernacki. 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*, CPP 2018, pages 266–279, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167091.

52 Théo Winterhalter. *Formalisation and meta-theory of type theory*. PhD thesis, Université de Nantes, 2020.

53 Théo Winterhalter. Composable partial functions in Coq, totally for free. In *29th International Conference on Types for Proofs and Programs*, 2023.

54 Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

A Declarative Martin-Löf Type Theory

On top of $t[u]$ for the substitution of the last variable of t by u, we more generally use $t[u, v, w]$ for the parallel substitution of the last variables of t with u, v and w.

A.1 Declarative conversion

$\boxed{\Gamma \vdash T \cong T'}$ Types T and T' are convertible in context Γ

$$\textsc{ReflTy}\ \frac{\Gamma \vdash A}{\Gamma \vdash A \cong A} \qquad \textsc{SymTy}\ \frac{\Gamma \vdash A \cong B}{\Gamma \vdash B \cong A} \qquad \textsc{TransTy}\ \frac{\Gamma \vdash A \cong B \qquad \Gamma \vdash B \cong C}{\Gamma \vdash A \cong C}$$

$$\textsc{ElC}\ \frac{\Gamma \vdash A \cong A' : \mathcal{U}}{\Gamma \vdash A \cong A'} \qquad \textsc{FunTyC}\ \frac{\Gamma \vdash A \cong A' \qquad \Gamma, x\colon A \vdash B \cong B'}{\Gamma \vdash \Pi\, x\colon A.B \cong \Pi\, x\colon A'.B'}$$

$$\textsc{SigTyC}\ \frac{\Gamma \vdash A \cong A' \qquad \Gamma, x\colon A \vdash B \cong B'}{\Gamma \vdash \Sigma\, x\colon A.B \cong \Sigma\, x\colon A'.B'}$$

$$\textsc{IdTyC}\ \frac{\Gamma \vdash A \cong A' \qquad \Gamma \vdash t \cong t' : A \qquad \Gamma \vdash u \cong u' : A}{\Gamma \vdash \mathbb{Id}(A, t, u) \cong \mathbb{Id}(A', t', u')}$$

$\boxed{\Gamma \vdash t \cong t' : T}$ Terms t and t' are convertible at type T in context Γ

$$\textsc{Refl}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash t \cong t : A} \qquad \textsc{Sym}\ \frac{\Gamma \vdash t \cong u : A}{\Gamma \vdash u \cong t : A} \qquad \textsc{Trans}\ \frac{\Gamma \vdash t \cong u : A \qquad \Gamma \vdash u \cong v : A}{\Gamma \vdash t \cong v : A}$$

$$\textsc{Conv}\ \frac{\Gamma \vdash t \cong t' : A \qquad \Gamma \vdash A \cong B}{\Gamma \vdash t \cong t' : B} \qquad \textsc{FunCong}\ \frac{\Gamma \vdash A \cong A' : \mathcal{U} \qquad \Gamma, x{:}\, A \vdash B \cong B' : \mathcal{U}}{\Gamma \vdash \Pi\, x{:}\, A.B \cong \Pi\, x{:}\, A'.B' : \mathcal{U}}$$

other congruences omitted

$$\beta\textsc{Fun}\ \frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma, x{:}\, A \vdash B \\ \Gamma, x{:}\, A \vdash t : B \qquad \Gamma \vdash u : A\end{array}}{\Gamma \vdash (\lambda\, x{:}\, A.t)\ u \cong t[u] : B[u]} \qquad \eta\textsc{Fun}\ \frac{\Gamma \vdash f : \Pi\, x{:}\, A.B}{\Gamma \vdash f \cong \lambda\, x{:}\, A.f\ x : \Pi\, x{:}\, A.B}$$

$$\beta\textsc{Sig}_1\ \frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma, x{:}\, A \vdash B \\ \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t]\end{array}}{\Gamma \vdash \pi_1\ (t, u)_{A.B} \cong t : A} \qquad \beta\textsc{Sig}_2\ \frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma, x{:}\, A \vdash B \\ \Gamma \vdash t : A \qquad \Gamma \vdash u : B[t]\end{array}}{\Gamma \vdash \pi_2\ (t, u)_{A.B} \cong u : B[t]}$$

$$\eta\textsc{Sig}\ \frac{\Gamma \vdash A \qquad \Gamma, x{:}\, A \vdash B \qquad \Gamma \vdash p : \Sigma\, x{:}\, A.B}{\Gamma \vdash p \cong (\pi_1\ p, \pi_2\ p)_{A.B} : \Sigma\, x{:}\, A.B}$$

$$\beta\textsc{Zero}\ \frac{\Gamma, x{:}\, \mathbb{N} \vdash P \qquad \Gamma \vdash b_0 : P[0] \qquad \Gamma, x{:}\, \mathbb{N}, y{:}\, P[x] \vdash b_{\mathrm{S}} : P[\mathrm{S}(x)]}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(0, x.P, b_0, x.y.b_{\mathrm{S}}) \cong b_0 : P[0]}$$

$$\beta\textsc{Succ}\ \frac{\Gamma \vdash n : \mathbb{N} \qquad \Gamma, x{:}\, \mathbb{N} \vdash P \qquad \Gamma \vdash b_0 : P[0] \qquad \Gamma, x{:}\, \mathbb{N}, y{:}\, P[x] \vdash b_{\mathrm{S}} : P[\mathrm{S}(x)]}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(\mathrm{S}(n), x.P, b_0, x.y.b_{\mathrm{S}}) \cong b_{\mathrm{S}}[n, \mathrm{rec}_{\mathbb{N}}(n, x.P, b_0, x.y.b_{\mathrm{S}})] : P[\mathrm{S}(n)]}$$

$$\beta\textsc{Refl}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash a : A \qquad \Gamma, x{:}\, A, y{:}\, \mathbb{Id}(A, a, x) \vdash P \qquad \Gamma \vdash b : P[a, \mathrm{refl}_{A,a}]}{\Gamma \vdash \mathrm{rec}_{\mathbb{Id}(A,a,\cdot)}(\mathrm{refl}_{A,a}, x.y.P, b) \cong b : P[a, \mathrm{refl}_{A,a}]}$$

A.2 Neutral comparison

$\boxed{\Gamma \vdash n \sim n' : T}$ Neutrals n and n' are convertible at type T

$$\textsc{NConv}\ \frac{\Gamma \vdash n \sim n' : T \qquad \Gamma \vdash T \cong S}{\Gamma \vdash n \sim n' : S} \qquad \textsc{NVar}\ \frac{(x{:}\, T) \in \Gamma}{\Gamma \vdash x \sim x : T}$$

$$\textsc{NApp}\ \frac{\Gamma \vdash n \sim n' : \Pi\, x{:}\, A.B \qquad \Gamma \vdash u \cong u' : A}{\Gamma \vdash n\ u \sim n'\ u' : B[u]} \qquad \textsc{NSig}_1\ \frac{\Gamma \vdash n \sim n' : \Sigma\, x{:}\, A.B}{\Gamma \vdash \pi_1\ n \sim \pi_1\ n' : A}$$

$$\textsc{NSig}_2\ \frac{\Gamma \vdash n \sim n' : \Sigma\, x{:}\, A.B}{\Gamma \vdash \pi_2\ n \sim \pi_2\ n' : B[\pi_1\ n]}$$

$$\textsc{NNatElim}\ \frac{\begin{array}{c}\Gamma \vdash n \sim n' : \mathbb{N} \\ \Gamma, x{:}\, \mathbb{N} \vdash P \cong P' \qquad \Gamma \vdash b_0 \cong b_0' : P[0] \qquad \Gamma, x{:}\, \mathbb{N}, y{:}\, P[n] \vdash b_{\mathrm{S}} \cong b_{\mathrm{S}}' : P[\mathrm{S}(n)]\end{array}}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(n, x.P, b_0, x.y.b_{\mathrm{S}}) \sim \mathrm{rec}_{\mathbb{N}}(n', x.P', b_0', x.y.b_{\mathrm{S}}') : P[n]}$$

$$\textsc{NEmptyElim}\ \frac{\Gamma \vdash n \sim n' : \bot \qquad \Gamma, x\colon \bot \vdash P \cong P'}{\Gamma \vdash \mathrm{rec}_{\bot}(n, x.P) \sim \mathrm{rec}_{\bot}(n', x.P') : P[s]}$$

$$\textsc{NIdInd}\ \frac{\begin{array}{c}\Gamma \vdash n \sim n' : \mathbb{I}\mathrm{d}(A'', a'', b'') \\ \Gamma, x\colon A, y\colon \mathbb{I}\mathrm{d}(A, a, x) \vdash P \cong P' \qquad \Gamma \vdash h \cong h' : P[a, \mathrm{refl}_{A,a}]\end{array}}{\Gamma \vdash \mathrm{rec}_{\mathbb{I}\mathrm{d}(A,a,\cdot)}(n, x.y.P, h) \sim \mathrm{rec}_{\mathbb{I}\mathrm{d}(A',a',\cdot)}(n', x.y.P', h') : P[b'']}$$

B Algorithmic Martin-Löf Type Theory

B.1 Typed algorithmic conversion

$\boxed{\Gamma \vdash T \cong T' \lhd}$ Types T and T' are convertible

$$\textsc{TyRed}\ \frac{T \rightsquigarrow^{\star} U \qquad T' \rightsquigarrow^{\star} U' \qquad \Gamma \vdash U \cong_{\mathrm{h}} U' \lhd}{\Gamma \vdash T \cong T' \lhd}$$

$\boxed{\Gamma \vdash t \cong t' \lhd A}$ Terms t and t' are convertible at type A

$$\textsc{TmRed}\ \frac{T \rightsquigarrow^{\star} U \qquad t \rightsquigarrow^{\star} u \qquad t' \rightsquigarrow^{\star} u' \qquad \Gamma \vdash u \cong_{\mathrm{h}} u' \lhd U}{\Gamma \vdash t \cong t' \lhd T}$$

$\boxed{\Gamma \vdash T \cong_{\mathrm{h}} T' \lhd}$ Reduced types T and T' are convertible

$$\textsc{CProdTy}\ \frac{\Gamma \vdash A \cong A' \lhd \qquad \Gamma, x\colon A' \vdash B \cong B' \lhd}{\Gamma \vdash \Pi\, x\colon A.B \cong_{\mathrm{h}} \Pi\, x\colon A'.B' \lhd}$$

$$\textsc{CSigTy}\ \frac{\Gamma \vdash A \cong A' \lhd \qquad \Gamma, x\colon A \vdash B \cong B' \lhd}{\Gamma \vdash \Sigma\, x\colon A.B \cong_{\mathrm{h}} \Sigma\, x\colon A'.B' \lhd} \qquad \textsc{CNatTy}\ \frac{}{\Gamma \vdash \mathbb{N} \cong_{\mathrm{h}} \mathbb{N} \lhd}$$

$$\textsc{CReflTy}\ \frac{}{\Gamma \vdash \bot \cong_{\mathrm{h}} \bot \lhd} \qquad \textsc{CIdTy}\ \frac{\Gamma \vdash A \cong A' \lhd \qquad \Gamma \vdash t \cong t' \lhd A \qquad \Gamma \vdash u \cong u' \lhd A}{\Gamma \vdash \mathbb{I}\mathrm{d}(A, t, u) \cong_{\mathrm{h}} \mathbb{I}\mathrm{d}(A', t', u') \lhd}$$

$$\textsc{CUniTy}\ \frac{}{\Gamma \vdash \mathcal{U} \cong_{\mathrm{h}} \mathcal{U} \lhd} \qquad \textsc{NeuTy}\ \frac{\Gamma \vdash n \sim n' \rhd T}{\Gamma \vdash n \cong_{\mathrm{h}} n' \lhd}$$

$\boxed{\Gamma \vdash t \cong_{\mathrm{h}} t' \lhd A}$ Reduced terms t and t' are convertible at type A

$$\textsc{CFun}\ \frac{\Gamma \vdash A \cong A' \lhd \mathcal{U} \qquad \Gamma, x\colon A' \vdash B \cong B' \lhd \mathcal{U}}{\Gamma \vdash \Pi\, x\colon A.B \cong_{\mathrm{h}} \Pi\, x\colon A'.B' \lhd \mathcal{U}} \qquad \textsc{CFunEta}\ \frac{\Gamma, x\colon A \vdash f\ x \cong f'\ x \lhd B}{\Gamma \vdash f \cong_{\mathrm{h}} f' \lhd \Pi\, x\colon A.B}$$

$$\textsc{CSig}\ \frac{\Gamma \vdash A \cong A' \lhd \mathcal{U} \qquad \Gamma, x\colon A' \vdash B \cong B' \lhd \mathcal{U}}{\Gamma \vdash \Sigma\, x\colon A.B \cong_{\mathrm{h}} \Sigma\, x\colon A'.B' \lhd \mathcal{U}} \qquad \textsc{CSigEta}\ \frac{\begin{array}{c}\Gamma \vdash \pi_1\ p \cong \pi_1\ p' \lhd A \\ \Gamma \vdash \pi_2\ p \cong \pi_2\ p' \lhd B[\pi_1\ p]\end{array}}{\Gamma \vdash p \cong_{\mathrm{h}} p' \lhd \Sigma\, x\colon A.B}$$

$$\textsc{CNat}\ \frac{}{\Gamma \vdash \mathbb{N} \cong_{\mathrm{h}} \mathbb{N} \lhd \mathcal{U}} \qquad \textsc{CZero}\ \frac{}{\Gamma \vdash 0 \cong_{\mathrm{h}} 0 \lhd \mathbb{N}} \qquad \textsc{CSucc}\ \frac{\Gamma \vdash t \cong t' \lhd \mathbb{N}}{\Gamma \vdash \mathrm{S}(t) \cong_{\mathrm{h}} \mathrm{S}(t') \lhd \mathbb{N}}$$

$$\textsc{CEmpty}\ \frac{}{\Gamma \vdash \bot \cong_{\mathrm{h}} \bot \lhd \mathcal{U}} \qquad \textsc{CId}\ \frac{\Gamma \vdash A \cong A' \lhd \mathcal{U} \qquad \Gamma \vdash t \cong t' \lhd A \qquad \Gamma \vdash u \cong u' \lhd A}{\Gamma \vdash \mathbb{Id}(A, t, u) \cong_{\mathrm{h}} \mathbb{Id}(A', t', u') \lhd \mathcal{U}}$$

$$\textsc{ReflRefl}\ \frac{}{\Gamma \vdash \mathrm{refl}_{A,a} \cong \mathrm{refl}_{A',a'} \lhd \mathbb{Id}(A'', t, u)} \qquad \textsc{NeuPos}\ \frac{\Gamma \vdash n \sim n' \rhd S \qquad \mathrm{isTy}_{+}\, T}{\Gamma \vdash n \cong_{\mathrm{h}} n' \lhd T}$$

$\boxed{\Gamma \vdash t \sim_{\mathrm{h}} t' \rhd T}$ Neutrals t and t' are comparable, inferring the reduced type T

$$\textsc{NRed}\ \frac{\Gamma \vdash n \sim n' \rhd T \qquad T \rightsquigarrow^{\star} S}{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd S}$$

$\boxed{\Gamma \vdash t \sim t' \rhd T}$ Neutrals t and t' are comparable, inferring the type T

$$\textsc{NVar}\ \frac{(x{:}\,T) \in \Gamma}{\Gamma \vdash x \sim x \rhd T} \qquad \textsc{NApp}\ \frac{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \Pi\, x{:}\, A.B \qquad \Gamma \vdash u \cong u' \lhd A}{\Gamma \vdash n\ u \sim n'\ u' \rhd B[u]}$$

$$\textsc{NSig}_1\ \frac{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \Sigma\, x{:}\, A.B}{\Gamma \vdash \pi_1\ n \sim \pi_1\ n' \rhd A} \qquad \textsc{NSig}_2\ \frac{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \Sigma\, x{:}\, A.B}{\Gamma \vdash \pi_2\ n \sim \pi_2\ n' \rhd B[\pi_1\ n]}$$

$$\textsc{NNatElim}\ \frac{\begin{array}{c}\Gamma \vdash n \sim n' \rhd \mathbb{N} \qquad \Gamma, x{:}\, \mathbb{N} \vdash P \cong P' \lhd \\ \Gamma \vdash b_0 \cong b_0' \lhd P[0] \qquad \Gamma, x{:}\, \mathbb{N}, y{:}\, P[n] \vdash b_{\mathrm{S}} \cong b_{\mathrm{S}}' \lhd P[\mathrm{S}(n)]\end{array}}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(n, x.P, b_0, x.y.b_{\mathrm{S}}) \sim \mathrm{rec}_{\mathbb{N}}(n', x.P', b_0', x.y.b_{\mathrm{S}}') \rhd P[n]}$$

$$\textsc{NEmptyElim}\ \frac{\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \bot \qquad \Gamma, x{:}\, \bot \vdash P \cong P' \lhd}{\Gamma \vdash \mathrm{rec}_{\bot}(n, x.P) \sim \mathrm{rec}_{\bot}(n', x.P') \rhd P[s]}$$

$$\textsc{NIdInd}\ \frac{\begin{array}{c}\Gamma \vdash n \sim_{\mathrm{h}} n' \rhd \mathbb{Id}(A'', a'', b'') \\ \Gamma, x{:}\, A, y{:}\, \mathbb{Id}(A, a, x) \vdash P \cong P' \lhd \qquad \Gamma \vdash h \cong h' \lhd P[a, \mathrm{refl}_{A,a}]\end{array}}{\Gamma \vdash \mathrm{rec}_{\mathbb{Id}(A,a,\cdot)}(n, x.y.P, h) \sim \mathrm{rec}_{\mathbb{Id}(A',a',\cdot)}(h', x.y.z.P', x.h') \rhd P[b'']}$$

B.2 Untyped algorithmic conversion

$\boxed{t \cong t'}$ Terms t and t' are convertible

$$\textsc{TmRed}\ \frac{t \rightsquigarrow^{\star} u \qquad t' \rightsquigarrow^{\star} u' \qquad u \cong_{\mathrm{h}} u'}{t \cong t'}$$

$\boxed{t \cong_{\mathrm{h}} t'}$ Reduced terms t and t' are convertible

$$\textsc{CUni}\ \frac{}{\mathcal{U} \cong_{\mathrm{h}} \mathcal{U}} \qquad \textsc{CFun}\ \frac{A \cong A' \qquad B \cong B'}{\Pi\, x{:}\, A.B \cong_{\mathrm{h}} \Pi\, x{:}\, A'.B'} \qquad \textsc{CLam}\ \frac{t \cong t'}{\lambda\, x{:}\, A.t \cong_{\mathrm{h}} \lambda\, x{:}\, A'.t'}$$

$$\textsc{CLamNe}\ \frac{\mathrm{ne}\, n' \qquad t \cong n'\ x}{\lambda\, x\colon A.t \cong_{\mathrm{h}} n'} \qquad\qquad \textsc{CNeLam}\ \frac{\mathrm{ne}\, n \qquad n\ x \cong n'}{n \cong_{\mathrm{h}} \lambda\, x\colon A'.t'}$$

$$\textsc{CSig}\ \frac{A \cong A' \qquad B \cong B'}{\Sigma\, x\colon A.B \cong_{\mathrm{h}} \Sigma\, x\colon A'.B'} \qquad\qquad \textsc{CPair}\ \frac{p \cong p' \qquad q \cong q'}{(p, q) \cong_{\mathrm{h}} (p', q')}$$

$$\textsc{CPairNe}\ \frac{p \cong \pi_1\ n' \qquad q \cong \pi_2\ n'}{(p, q) \cong_{\mathrm{h}} n'} \qquad\qquad \textsc{CNePair}\ \frac{\pi_1\ n \cong p' \qquad \pi_2\ n \cong q'}{n \cong_{\mathrm{h}} (p', q')}$$

$$\textsc{CNat}\ \frac{}{\mathbb{N} \cong_{\mathrm{h}} \mathbb{N}} \qquad \textsc{CZero}\ \frac{}{0 \cong_{\mathrm{h}} 0} \qquad \textsc{CSucc}\ \frac{t \cong t'}{\mathrm{S}(t) \cong_{\mathrm{h}} \mathrm{S}(t')} \qquad \textsc{CEmpty}\ \frac{}{\bot \cong_{\mathrm{h}} \bot}$$

$$\textsc{CId}\ \frac{A \cong A' \qquad t \cong t' \qquad u \cong u'}{\mathbb{I}\mathrm{d}(A, t, u) \cong_{\mathrm{h}} \mathbb{I}\mathrm{d}(A', t', u')} \qquad \textsc{ReflRefl}\ \frac{}{\mathrm{refl}_{A,a} \cong \mathrm{refl}_{A',a'}} \qquad \textsc{NeuNeu}\ \frac{n \sim n'}{n \cong_{\mathrm{h}} n'}$$

$\boxed{n \sim n'}$ Neutrals n and n' are comparable

$$\textsc{NVar}\ \frac{}{x \sim x} \qquad\qquad \textsc{NApp}\ \frac{n \sim n' \qquad u \cong u'}{n\ u \sim n'\ u'} \qquad\qquad \textsc{NSig}_1\ \frac{n \sim n'}{\pi_1\ n \sim \pi_1\ n'}$$

$$\textsc{NSig}_2\ \frac{n \sim n'}{\pi_2\ n \sim \pi_2\ n'} \qquad \textsc{NNatElim}\ \frac{n \sim n' \qquad P \cong P' \qquad b_0 \cong b_0' \qquad b_{\mathrm{S}} \cong b_{\mathrm{S}}'}{\mathrm{rec}_{\mathbb{N}}(n, x.P, b_0, x.y.b_{\mathrm{S}}) \sim \mathrm{rec}_{\mathbb{N}}(n', x.P', b_0', x.y.b_{\mathrm{S}}')}$$

$$\textsc{NEmptyElim}\ \frac{n \sim n' \qquad P \cong P'}{\mathrm{rec}_{\bot}(n, x.P) \sim \mathrm{rec}_{\bot}(n', x.P')}$$

$$\textsc{NIdInd}\ \frac{n \sim_{\mathrm{h}} n' \qquad P \cong P' \qquad h \cong h'}{\mathrm{rec}_{\mathbb{I}\mathrm{d}(A,a,\cdot)}(n, x.y.P, h) \sim \mathrm{rec}_{\mathbb{I}\mathrm{d}(A',a',\cdot)}(h', x.y.z.P', x.h')}$$