# Equality checking for dependent type theories*

Andrej Bauer and Anja Petković

University of Ljubljana, Ljubljana, Slovenia

Equality checking algorithms are essential components of proof assistants based on type theories [9, 3, 10, 13, 12, 1]. They free the user from the burden of proving equalities, and provide computation-by-normalization engines. Some systems [11, 8, 7] also allow user extensions to the built-in equality checkers, possibly sacrificing completeness and sometimes even soundness. The situation is even more challenging in a proof assistant that supports arbitrary user-definable type theories, such as Andromeda 2 [4, 5], where in general no equality checking algorithm may be available. Still, the proof assistant should provide convenient support for equality checking that works well in the common, well-behaved cases.

We developed an extensible equality checking algorithm and proved it to be sound [6] for a large class of dependent type theories. The algorithm is parameterized by computation rules ($\beta$-rules), extensionality rules (inter-derivable with $\eta$-rules), and a notion of normal form. It combines and extends algorithms based on type-directed equality checking [14, 2] that intertwine two phases: the type-directed phase applies extensionality rules to reduce the problem to simpler types, while the normalization phase applies computation rules to compute normal forms.

We define precisely what it means for an equality rule to be a computation or an extensionality rule. For this purpose we identify the notion of an object-invertible rule, which guarantees soundness of normalization steps and of type-directed reductions of subsidiary equations. We give simple syntactic criteria for recognizing computation and extensionality rules.

We implemented the algorithm in the Andromeda 2 proof assistant in around 1400 lines of OCaml code. The user needs only provide the equality rules they wish to use, after which the algorithm automatically classifies them either as computation or extensionality rules (and rejects those that are of neither kind), and devises an appropriate notion of normal form. The implementation consults the nucleus to build a trusted certificate of every equality it proves and every term it normalizes. It is easy to experiment with different sets of equality rules and dynamically switch between them. In the case of well-behaved type theories, such as the simply typed lambda calculus or Martin-Löf type theory, the algorithm behaves like well-known standard equality checkers. We do not address completeness and termination, as these depend heavily on the choice of computation and extensionality rules.

**Object-invertible, computation and extensionality rules.** In an inference rule

$$\frac{P_1 \quad \cdots \quad P_n}{C}$$

the *object premises* are those $P_i$ which are type or term judgements, and *equational premises* those that are type or term equations. We say that such a rule is *object-invertible* when the following holds for every instance of it: if the conclusion is derivable (possibly by application of a different rule) then the object premises are derivable.

Object-invertible rules may be used to invert derivable judgements up to equational premises. That is, if a derivable judgement $J$ coincides with some instance of the conclusion $C$ of an object-invertible rule, then we are guaranteed that the corresponding instances of the object premises are also derivable, so only the equational premises must be checked.

*This material is based upon work supported by the U.S. Air Force Office of Scientific Research under award number FA9550-17-1-0326, grant number 12595060, and award number FA9550-21-1-0024. We thank Philipp G. Haselwarter, who was initially contributing to the project, for his support and discussions.

A *type computation rule* is a derivable type equality rule, shown below on the left,

$$\frac{P_1 \quad \cdots \quad P_n}{\vdash A \equiv B} \qquad\qquad \frac{P_1 \quad \cdots \quad P_n}{\vdash A\ \mathsf{type}}$$

such that its left-hand side presupposition, shown above on the right, is object-invertible and deterministic (its conclusion can be instantiated to match a given judgement in at most one way). *Term computation rules* are defined similarly.

An *extensionality rule* is a derivable rule, shown below on the left,

$$\frac{P_1 \quad \cdots \quad P_n \quad \vdash x : C \quad \vdash y : C \quad Q_1 \quad \cdots \quad Q_m}{\vdash x \equiv y : C} \qquad\qquad \frac{P_1 \quad \cdots \quad P_n}{\vdash C\ \mathsf{type}}$$

such that $Q_1, \ldots, Q_m$ are equational premises, and its type presupposition, shown above on the right, is object-invertible and derivable.

**Principal arguments and normal forms.** A third component of the algorithm is a suitable notion of normal form, which guarantees correct execution of normalization and coherent interaction of both phases of the algorithm. In our setting, normal forms are determined by a selection of *principal arguments*. By varying these, we obtain known notions, such as weak head-normal and strong normal forms (all arguments are declared principal). An expression is said to be in normal form if no computation rule applies to it, and its principal arguments are in normal form.

In the implementation the user may specify the principal arguments directly, or let the algorithm read the principal arguments off the computation rules automatically, as follows: if $\mathsf{s}(u_1, \ldots, u_n)$ appears as a left-hand side of a computation rule, then the principal arguments of $\mathsf{s}$ are those $u_i$'s that are *not* metavariables, i.e., matching against them does not automatically succeed, and so they should first be normalized.

**Overview of the type-directed equality checking.** The equality checking algorithm is parameterized by the underlying type theory, computation rules, extensionality rules, and principal arguments. It has the following mutually recursive parts:

1. *Normalize a type or a term:* normalize the principal arguments, apply a computation rule and recursively check subsidiary equations as they arise; repeat until no computation rule applies.
2. *Check $A \equiv B$:* normalize $A$ and $B$ and structurally compare their normal forms.
3. *Structurally check $A \equiv B$:* compare $A$ and $B$ by an application of a congruence rule, where the principal arguments are recursively compared structurally and the others by the general equality checks.
4. *Check $s \equiv t : A$:*
   (a) *type-directed phase:* normalize $A$ and apply extensionality rules, if any, to reduce the equality to subsidiary equalities,
   (b) *normalization phase:* if no extensionality rules apply, normalize $s$ and $t$ and structurally compare their normal forms.
5. *Structurally check $s \equiv t : A$:* compare $s$ and $t$ by an application of a congruence rule, where the principal arguments are recursively compared structurally and the others by the general equality checks.

# References


[1] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages*, 2(POPL), December 2017.

[2] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, Volume 8, Issue 1, 2012.

[3] The Agda proof assistant. `https://wiki.portal.chalmers.se/agda/`.

[4] The Andromeda proof assistant. `http://www.andromeda-prover.org/`.

[5] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and implementation of the Andromeda proof assistant. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *LIPIcs*, pages 5:1–5:31. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[6] Andrej Bauer and Anja Petković. An extensible equality checking algorithm for dependent type theories, 2021.

[7] Jesper Cockx. Type theory unchained: Extending Agda with user-defined rewrite rules. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[8] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In *22nd International Conference on Types for Proofs and Programs TYPES 2016*, University of Novi Sad, 2016.

[9] The Coq proof assistant. `https://coq.inria.fr/`.

[10] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *25th International Conference on Automated Deduction (CADE 25)*, August 2015.

[11] The Dedukti logical framework. `https://deducteam.github.io`.

[12] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof-Irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019.

[13] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), December 2019.

[14] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, 2006.