On equality checking for general type theories: Implementation in Andromeda 2[*]

Andrej Bauer, Philipp G. Haselwarter, and Anja Petković

University of Ljubljana, Ljubljana, Slovenia

Equality checking algorithms are essential components of proof assistants based on type theories [Coq, Agd, dMKA+15, SBF+19, GCST19, AOV17]. They free the user from the burden of proving equalities, and provide computation-by-normalization engines. The type theories found in the most popular type-theoretic proof assistants are carefully designed to have decidable equality. Some systems [Ded, CA16] also allow user extensions to the built-in equality checkers, possibly sacrificing their completeness.

The situation is worse in a proof assistant that supports arbitrary user-definable theories, such as Andromeda 2 [And, BGH+18], where in general no equality checking algorithm may be available. Short of implementing exhaustive proof search, the construction of equality proofs must be delegated to the user (and still checked by the trusted nucleus). While some may appreciate the opportunity to tinker with equality checking procedures, they are surely outnumbered by those who prefer good support that automates equality checking with minimal effort, at least for well-behaved type theories that one encounters in practice.

We have designed and implemented in Andromeda 2 an extensible equality checking algorithm that supports user-defined computation rules (β-rules) and extensionality rules (interderivable with η-rules). The user needs only to 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 weak head-normal form. In the case of well-behaved type theories such as the simply typed lambda calculus or Martin-Löf type theory with η for dependent products, the algorithm behaves like well-known standard equality checkers. In general, it may be incomplete or non-terminating, but it can never be the source of unsoundness because it resides outside of the trusted nucleus.

Our algorithm is a variant of a type-directed equality checking algorithm [SH06, AS12], described in more detail below. It is implemented in around 1300 lines of OCaml code. The algorithm consults the nucleus to build a trusted certificate of every equality it proves and every term normalization it performs. It is easy to experiment with different sets of equality rules, and dynamically switch between them depending on the situation at hand. Our initial experiments are encouraging, although many opportunities for optimization and improvements await.

Type-directed equality checking. The kind of equality checking algorithm that we employ is comprised of several mutually recursive subroutines:

1. *Weak head-normalize a type A:* the user-provided type computation rules are applied to A to give a sequence of equalities $A \equiv A_1 \equiv \cdots \equiv A_n$, until no more rules apply. Then the heads of A_n are normalized recursively (see below) to obtain $A_n \equiv A'_n$, after which the (certified) equality $A \equiv A'_n$ is output.
2. *Weak head-normalize a term $t : A$:* analogously to normalization of types, the user-provided term computation rules are applied to t until no more rules apply.

[*]This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

3. *Check equality of types $A \equiv B$:* the types A and B are normalized and their normal forms are compared.
4. *Check equality of normalized types $A \equiv B$:* normalized types are compared structurally, i.e., by an application of a suitable congruence rule. Their subexpressions are compared recursively. For example, to prove $\Pi_{(x:C)} D \equiv \Pi_{(x:C')} D'$, we recursively prove $C \equiv C'$ and $x : C \vdash D \equiv D'$.
5. *Check equality of terms $s \equiv t : A$:*
   (a) *type-directed phase:* normalize the type A and based on its normal form apply user-provided extensionality rules, if any, to reduce the equality to subsidiary equalities,
   (b) *normalization phase:* if no extensionality rules apply, normalize s and t and compare their normal forms.
6. *Check equality of normalized terms $s \equiv t : A$:* normalized terms are compared structurally, analogously to comparison of normalized types.

One needs to choose the notions of "computation rule", "extensionality rule" and "normal form" wisely in order to guarantee completeness. In particular, in the type-directed phase the type at which the comparisons are carried out should decrease with respect to a well-founded notion of size, while normalization should be confluent and terminating. These concerns are external to the system, and so the user is allowed to install rules without providing any guarantees of completeness or termination.

Computation and extensionality rules. Term computation rules, type computation rules, and extensionality rules respectively have the forms

$$\frac{P_1 \ \cdots \ P_n}{\vdash u \equiv v : A} \qquad \frac{P_1 \ \cdots \ P_n}{\vdash A \equiv B} \qquad \frac{P_1 \ \cdots \ P_n \quad \vdash x : A \quad \vdash y : A \quad Q_1 \ \cdots \ Q_m}{\vdash x \equiv y : A}$$

In a term computation rule, $P_1, \ldots, P_n$ are *object premises* that introduce term and type meta-variables, while u must be a term symbol applied to subexpressions in which all the meta-variables appear. An extensionality rule, as above, has object premises $P_1, \ldots, P_n$ and subsidiary equality premises $Q_1, \ldots, Q_m$. We require that every meta-variable introduced by the premises appears in A. To tell whether such a rule applies to $s \equiv t : B$, we pattern match B against A, and proceed to work on the instantiated equality subgoals $Q_1, \ldots, Q_m$.

Heads and normal forms. For the algorithm to work correctly, it needs a notion of normal forms that matches the equality rules. We use *weak head-normal forms*: an expression is said to be in normal form if no computation rule applies to it, and its heads are in normal form. Furthermore, when normal forms are compared structurally, their heads are compared structurally in a recursive fashion, while the remaining arguments are compared as ordinary (non-normal) expressions.

The question arises, how to figure out which arguments of a term or a type symbol are the heads. For instance, how can we tell that the third argument of fst above is a head, while pair has no heads? In our implementation the user may specify the heads directly, or let the algorithm read the heads 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 heads of s are those u_i's that are *not* meta-variables, i.e., matching against them does not automatically succeed, and so further normalization is required. By varying the notion of heads we may control how expressions are normalized. For example, strong normal forms are just weak head-normal forms in which all arguments are declared to be heads.

References

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

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

[AOV17] 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.

[AS12] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, Volume 8, Issue 1, 2012.

[BGH$^+$18] 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.

[CA16] 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.

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

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

[dMKA$^+$15] 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.

[GCST19] 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.

[SBF$^+$19] 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.

[SH06] Christopher A. Stone and Robert Harper. Extensional equivalence and singleton types. *ACM Transactions on Computational Logic*, 7(4):676–722, 2006.