Equality Checking for General Type Theories in Andromeda 2

Andrej Bauer, Philipp G. Haselwarter, and Anja Petković(✉)

University of Ljubljana, Ljubljana, Slovenia
Andrej.Bauer@andrej.com , philipp@haselwarter.org,
Anja.Petkovic@fmf.uni-lj.si

Abstract. We designed a user-extensible judgemental equality checking algorithm for general type theories that supports computation rules and extensionality rules. The user needs only provide the equality rules they wish to use, after which the algorithm devises an appropriate notion of normal form. The algorithm is a generalization of type-directed equality checking for Martin-Löf type theory, and we implemented it in the Andromeda 2 prover.

Keywords: Algorithmic equality checking · Dependent type theory · Proof assistant

1 Introduction

Equality checking algorithms are essential components of proof assistants based on type theories [1,3,7,9,11,13]. They free users from the burden of proving judgemental equalities, and provide computation-by-normalization engines. Indeed, the type theories found in the most popular proof assistants are designed to provide such algorithms. Some systems [6,8] go further by allowing (possibly unsafe) user extensions to the built-in equality checkers.

The situation is less pleasant in a proof assistant that supports arbitrary user-definable theories, such as Andromeda 2 [4,5], where in general no equality checking algorithm may be available. For example, the well-known Martin-Löf "extensional" type theory that includes the equality reflection rule is well-known to have undecidable judgemental equality, and is readily definable in Andromeda 2. 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

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

A. M. Bigatti et al. (Eds.): ICMS 2020, LNCS 12097, pp. 253–259, 2020.
https://doi.org/10.1007/978-3-030-52200-1_25

extensionality rules (inter-derivable 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 normal form. For the usual kinds of type theories (simply typed λ-calculus, Martin-Löf type theory), the algorithm behaves like well-known standard equality checkers.

Our algorithm is a variant of a type-directed equality checking [2,14], as outlined below. It is implemented in about 1300 lines of OCaml code, which resides outside the trusted nucleus. The algorithm calls the nucleus to build a trusted certificate of every equality step, and of every term normalization it performs, so all equalities established by the algorithm, including intermediate steps, are verified. 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.

2 Andromeda 2

Andromeda 2 is an experimental LCF-style proof assistant, i.e., it is a meta-level programming language with an abstract datatype of judgements whose constructors are controlled by a trusted nucleus. We review just enough of it to be able to explain the equality checking algorithm.

In Andromeda 2 the user defines their own type theory by declaring the inference rules for types, terms and equalities. For example, formation of dependent products and the successor for natural numbers,

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x{:}A \vdash B\ \mathsf{type}}{\Gamma \vdash \prod(x{:}A)\,.\,B\ \mathsf{type}} \qquad \frac{\Gamma \vdash x : \mathbb{N}}{\Gamma \vdash s(x) : \mathbb{N}}$$

are written respectively as

```
rule Π (A type) ({x:A} B type) type
rule s (x : ℕ) : ℕ
```

The typing context Γ is left implicit (henceforth we shall elide Γ from all rules), while the context extension $x{:}A$ in the second premise of the product rule is expressed as an *abstraction*. In Andromeda 2 `{x:A} e` is a primitive operation that abstracts the variable `x` in `e`.

The user may also specify equality rules. For instance, the β-rule for functions is written as

```
rule β (A type) ({x:A} B type) ({x:A} t : B{x}) (a : A)
  : app A B (λ A B t) a ≡ t{a} : B{a}
```

where `app` and `λ` are the expected term formers corresponding to application and λ-abstraction, respectively. The notation `t{a}` instantiates the bound variable `x` in `t` with `a`. Note that all terms are fully annotated with types.

The object type theory has no primitive notion of definition (not to be confused with `let`-binding at the meta-language level). Instead, the user may simply declare an equational rule that serves as a definition, e.g.,

```
rule three : ℕ
rule three_def : three ≡ s (s (s z)) : ℕ
```

Structural rules are built into the nucleus. These are reflexivity, symmetry, and transitivity of equality, as well as support for abstraction and substitution. The nucleus automatically generates congruence rules for all term and type formers. For example, the computation

```
congruence (Π A B) (Π C D) α β
```

derives `Π A B ≡ Π C D` by an application of the congruence rule for products. Here `α` and `β` are computations that further consult the nucleus to compute equalities `A ≡ C` and `{x:A} B{x} ≡ D{x}`, respectively.

3 Computation and Extensionality Rules

We describe precisely what form computation and extensionality rules take. For this purpose, define an *object judgement* to be one of the form $A\ \mathsf{type}$ or $t : A$, and an *equation judgement* of the form $A \equiv B$ or $s \equiv t : A$. Accordingly, a premise of an inference rule may be either an object or an equation premise.

Term and type *computation rules* respectively have the forms

$$\frac{P_1 \ \cdots \ P_n}{\vdash u \equiv v : A} \qquad \frac{P_1 \ \cdots \ P_n}{\vdash A \equiv B}$$

where the P_i's are object premises. Furthermore, in a term computation rule the left-hand side u must take the form $\mathsf{s}(e_1, \ldots, e_m)$ where s is a term symbol. In other words, u may not be a variable or a meta-variable. Likewise, in an equation computation rule the left-hand side A must take the form $\mathsf{S}(e_1, \ldots, e_m)$ where S is a type symbol. Additionally, all the meta-variables introduced by the premises must appear in the arguments e_j. These conditions ensure that, given a term t, performing simple pattern matching of t against u tells us whether the rule applies to t and how. An example of a computation rule is the usual β-rule for simple products:

$$\frac{\vdash A\ \mathsf{type} \quad \vdash B\ \mathsf{type} \quad \vdash p : A \quad \vdash r : B}{\vdash \mathsf{fst}(A, B, \mathsf{pair}(A, B, p, r)) \equiv p : A}$$

Observe that the left-hand side of the equation mentions all four meta-variables A, B, p, r. In Andromeda 2 the above rule is postulated as

```
rule fst_β (A type) (B type) (p : A) (r : B) :
  fst A B (pair A B p r) ≡ p : A
```

and installed into the equality checker with `eq.add_rule fst_β`. The equality checker automatically determines that `fst_β` is a computation rule.

An *extensionality rule* says, broadly speaking, that two types or terms are equal when their eliminations are equal. Such a rule has the form

$$\frac{P_1 \ \cdots \ P_n \quad \vdash x : A \quad \vdash y : A \quad Q_1 \ \cdots \ Q_m}{\vdash x \equiv y : A},$$

where $P_1, \ldots, P_n$ are object premises and $Q_1, \ldots, Q_m$ are equality premises. We require that every meta-variable introduced by the premises appear in A. To tell whether such a rule applies to $s \equiv t : B$, we pattern match B against A, and recursively check suitably instantiated subsidiary equalities $Q_1, \ldots, Q_m$. Note that both sides of the conclusion of an extensionality rule must be meta-variables, so that the rule applies as soon as the type matches.

As an example we give the extensionality rule for simple products:

$$\frac{\begin{array}{c}\vdash A\ \mathsf{type} \quad \vdash B\ \mathsf{type} \quad \vdash p : A \times B \quad \vdash q : A \times B \\ \vdash \mathsf{fst}(A, B, p) \equiv \mathsf{fst}(A, B, q) : A \quad \vdash \mathsf{snd}(A, B, p) \equiv \mathsf{snd}(A, B, q) : B\end{array}}{\vdash p \equiv q : A \times B}$$

In Andromeda 2 it is postulated as

```
rule prod_ext (A type) (B type) (p : A × B) (q : A × B)
  (fst A B p ≡ fst A B q : A)
  (snd A B p ≡ snd A B q : B) :
  p ≡ q : A × B
```

Again, the rule is installed with the command `eq.add_rule prod_ext`.

A second example is the extensionality rule for dependent functions (not to be confused with function extensionality):

$$\frac{\begin{array}{c}\vdash A\ \mathsf{type} \quad x{:}A \vdash B\ \mathsf{type} \quad \vdash f : \prod(x{:}A)\,.\,B \quad \vdash g : \prod(x{:}A)\,.\,B \\ x{:}A \vdash \mathsf{app}(A, B, f, x) \equiv \mathsf{app}(A, B, g, x) : B(x)\end{array}}{\vdash f \equiv g : \prod(x{:}A)\,.\,B}$$

which in Andromeda is written as

```
rule Π_ext (A type) ({x:A} B type)
  (f : Π A B) (g : Π A B)
  ({x:A} app A B f x ≡ app A B g x : B{x}) :
  f ≡ g : Π A B
```

It is easy to see that the `Π_ext` rule is inter-derivable with the η-rule for functions.

4 Normalizing Arguments and Normal Forms

The equality checking algorithm from Sect. 5 requires a notion of normal forms. We define an expression to be *normal* if no computation rule applies to it, and its *normalizing arguments* are in normal form. Thus, our notion of normal form depends on the computation and extensionality rules, as well as on which arguments of term and type symbols are normalizing.

In Andromeda 2 the user may specify the normalizing arguments directly, or let the algorithm determine the normalizing arguments from the computation rules automatically as follows: if $\mathtt{s}(u_1, \ldots, u_n)$ appears as a left-hand side of a computation rule, then the normalizing arguments of `s` are those u_i's that are *not* meta-variables, i.e., matching against them does not automatically succeed, and so they have to be normalized before they are matched.

By varying the notion of normalizing arguments we can control how expressions are normalized. The automatic procedure results in weak head-normal forms, while strong normal forms are obtained if all the arguments are declared to be normalizing.

The normal form of a term t of type A is computed by a call to the command `eq.normalize` t, which outputs a certified equation $t \equiv t' : A$ where t' is the normal form of t. Similarly the command `eq.compute` t provides the strong normal form of t. Normalization of types works analogously.

The user may also verify an equation, say equality of types A and B, by running the command

```
eq.prove (A ≡ B by ??)
```

The equality checker outputs a certified judgement $A \equiv B$, or reports failure. In the above command, $A \equiv B$ `by ??` is a *boundary*, which is a primitive notion in Andromeda 2 that expresses a goal. Each judgement form has a corresponding boundary: "`??`$:A$" is the goal asking that the type A be inhabited, "`?? type`" that a type be constructed, and "$t \equiv s : A$ `by ??`" that equality of terms s and t be proved.

5 An Overview of the Equality-Checking Algorithm

The equality-checking algorithm has several mutually recursive sub-algorithms:

1. *Normalize a type A:* the user-provided type computation rules are applied to A to give a sequence of (nucleus verified) equalities $A \equiv A_1 \equiv \cdots \equiv A_n$, until no more rules apply. Then the normalizing arguments of A_n are normalized recursively to obtain $A_n \equiv A'_n$, after which the equality $A \equiv A'_n$ is output.
2. *Normalize a term t of type A:* analogously to normalization of types, the user-provided term computation rules are applied to t until no more rules apply, after which the normalizing arguments are normalized.
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 normal types* $A \equiv B$*:* normal types are compared structurally, i.e., by an application of a suitable congruence rule. The arguments are compared recursively: the normalizing ones by applications of congruence rules, and the non-normalizing ones by applications of the algorithm.
5. *Check equality of terms* s *and* t *of type* 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 normal terms* s *and* t *of type* A*:* normal terms are compared structurally, analogously to comparison of normal types.

One needs to choose the notions of "computation rule", "extensionality rule" and "normalizing argument" 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.

6 Related and Future Work

Dedukti [8] is a proof assistant based on $\lambda\Pi$ modulo user-definable equational theories. Its pattern matching and rewriting capabilities are more advanced than ours. It does not have a type-directed phase through which user-defined extensionality rules could be applied, although of course one can reformulate those as η-rules.

Similar in spirit to Andromeda 2 is the equality checking algorithm used in the reconstruction phase of MMT [10,12], a meta-meta-language for description of formal theories. While in Andromeda 2 the user specifies the rules in declarative style that cannot break the trust in the nucleus, in MMT inference rules are implemented directly as executable code. This makes MMT more flexible at the price of importing arbitrary user-code into the trusted part of the system.

Our equality checker is general enough to support a wide range of equality checking algorithms that are based on a type-directed phase followed by normalization. It is easy to use because it automatically classifies equality rules as either computation or extensionality rules, and determines which arguments are normalizing. There are several possible future directions of research, of which we mention three.

First, we have already experimented with *local* equality rules that are installed temporarily. This is sometimes necessary to establish that an object appearing in a rule is well-formed due to an equational premise. More work is needed to design a usable interface for such local rules.

Second, there is no support for checking termination or confluence of the given rules. Consequently, the user may inadvertently install rules that cause

the normalization phase to diverge, or experience unpredictable behaviour when the rules are not confluent. It would be worthwhile helping the user in this respect.

Third, combining our equality checker with other kinds of equality-checking algorithms would further facilitate proof development. Even naive proof search could be useful in certain situations. In principle, the user may direct Andromeda 2 to use a specific equality checker in a given situation, but it would be friendlier if the system behaved in an intelligent way with minimal direction from the user.

References

1. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. In: Proceedings of the ACM on Programming Languages, vol. 2, no. POPL, December 2017
2. Abel, A., Scherer, G.: On irrelevance and algorithmic equality in predicative type theory. Log. Methods Comput. Sci. **8**(1) (2012). https://lmcs.episciences.org/1045
3. The Agda proof assistant. https://wiki.portal.chalmers.se/agda/
4. The Andromeda proof assistant. http://www.andromeda-prover.org/
5. Bauer, A., Gilbert, G., Haselwarter, P.G., Pretnar, M., Stone, C.A.: Design and implementation of the Andromeda proof assistant. In: 22nd International Conference on Types for Proofs and Programs (TYPES 2016). LIPIcs, vol. 97, pp. 5:1–5:31 (2018)
6. Cockx, J., Abel, A.: 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)
7. The Coq proof assistant. https://coq.inria.fr/
8. The Dedukti logical framework. https://deducteam.github.io
9. Gilbert, G., Cockx, J., Sozeau, M., Tabareau, N.: Definitional proof-irrelevance without K. In: Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, January 2019
10. The MMT language and system. https://uniformal.github.io//
11. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
12. Rabe, F.: A modular type reconstruction algorithm. ACM Trans. Comput. Log. **19**(4), 24:1–24:43 (2018)
13. Sozeau, M., Boulier, S., Forster, Y., Tabareau, N., Winterhalter, T.: Coq Coq correct! Verification of type checking and erasure for Coq, in Coq. In: Proceedings of the ACM on Programming Languages, vol. 4, no. POPL, December 2019
14. Stone, C.A., Harper, R.: Extensional equivalence and singleton types. ACM Trans. Comput. Log. **7**(4), 676–722 (2006)