Design and Implementation of the Andromeda proof assistant

Andrej Bauer[1], Gaëtan Gilbert[2], Philipp Haselwarter[1], Matija Pretnar[1], and Christopher A. Stone[3]

[1] University of Ljubljana, Slovenia
[2] École Normale Supérieure Lyon, France
[3] Harvey Mudd College, USA

Andromeda [1] is a proof assistant for dependent type theory with equality reflection following the tradition of Edinburgh LCF [3]: (1) there is an abstract datatype of type-theoretic judgments whose values can only be constructed by a small *nucleus*, and (2) the user interacts with the nucleus by writing programs in a high-level, statically typed *Andromeda meta-language (AML)*. The only part of the system that needs to be trusted is the nucleus, which at present counts around 1800 lines of OCaml code.

The underlying type theory of Andromeda has dependent products and equality types (LCF and its descendants implement simple type theory). The rules for products are standard and include function extensionality. The terms are explicitly tagged with typing annotations, which is necessary because we want to avoid various anomalies caused by the *equality reflection* rule [4]

$$\frac{\Gamma \vdash e : \mathsf{Eq}_T(e_1, e_2)}{\Gamma \vdash e_1 \equiv e_2 : T}$$

The rule has the additional disadvantage of making judgmental equality undecidable. Nevertheless, this is the type theory we want to implement because it has a great deal of expressive power. The user may essentially adjoin new judgmental equalities by hypothesizing inhabitants of the corresponding equality types, and thus axiomatize many type-theoretic constructions (sums, propositional truncation, (co)inductive types, inductive-inductive and inductive-recursive types, etc.). In contrast to the J-rule of intensional type theory, equality reflection erases uses of equality proofs, which ought to prove useful in certain kinds of formalization. By combining equality reflection with handlers, described below, the user also controls opacity of definitions and application of (user-provided) normalization strategies.

The *AML evaluator* performs bidirectional type checking of terms, invoking operations (questions) that can be handled (answered) by user-provided AML code in the style of Eff [2]. For instance, to construct a well-typed application $e_1\, e_2$, the first step is to synthesize the type T_1 of e_1 and express it as a product. Since type theory with equality reflection does not enjoy strong normalization, the evaluator simply triggers an operation $\texttt{as\_prod}(\Gamma \vdash T_1 : \texttt{Type})$ and expects a handler to yield back an inhabitation judgment $\Gamma \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(T_1, \prod_{(x:A)} B)$ for some A, B, and ξ. Once it has the appropriate premises, the evaluator invokes the nucleus to construct the resulting judgment $\Gamma \vdash e_1\, e_2 : B[e_2/x]$ (with some typing annotations elided). There is also an operation `as_eq` for asking how to convert a type to an equality type.

Similarly, whenever the evaluator encounters a non-trivial equality $\Gamma \vdash e_1 \equiv e_2 : T$ it triggers the operation $\texttt{equal}(\Gamma \vdash e_1 : T)(\Gamma \vdash e_2 : T)$, and the equation is verified if and when the handler yields a judgment $\Gamma \vdash e : \mathsf{Eq}_T(e_1, e_2)$ back to the evaluator. The handler can employ an arbitrary equality checking algorithm, using support from the nucleus to produce witnesses for the β-rule, function extensionality, η-rules for records, uniqueness of equality proofs, and congruence rules.

```
λ (θ : a ≡ b) (ξ : b ≡ c),
  handle θ : a ≡ c with
  | equal ((⊢ a ≡ b) as ?X) ((⊢ a ≡ c) as ?Y) =>
      handle yield (congruence X Y) with
      | equal (⊢ b) (⊢ c) => yield (Some ξ)
      end
  end
```

Listing 1: Transitivity of equality

In practice most equalities can be verified by a standard type-directed equality checking algorithm. We have implemented such an algorithm in AML and extended it with *equality hints*. These allow the user to dynamically add extensionality rules, β-rules, and instructions on how to immediately resolve equalities that are not amenable to rewriting (such as commutativity of addition). The fact that the algorithm is implemented in AML gives it a strong correctness guarantee.

For example, given a type T with elements $a, b, c : T$, the AML program[1] in Listing 1 computes a witness of $\mathsf{Eq}_T(a,b) \to \mathsf{Eq}_T(b,c) \to \mathsf{Eq}_T(a,c)$, namely the term $\lambda\theta\,\xi\,.\,\theta$ (where type annotations have been elided). While verifying that θ does have type $\mathsf{Eq}_T(a,c)$, the nucleus encounters a non-trivial equality of types $\mathsf{Eq}_T(a,b)$ and $\mathsf{Eq}_T(a,c)$. The outer handler handles this by an application of `congruence`, which attempts to generate a witness of equality by applying congruence rules. Structural comparison of $\mathsf{Eq}_T(a,b)$ and $\mathsf{Eq}_T(a,c)$ generates three further equality checks, of which $T \equiv_{\mathsf{Type}} T$ and $a \equiv_T a$ are trivial, and $b \equiv_T c$ is handled by the hypothesis ξ. In general we do not expect users to write such low-level handlers but rather rely on a sophisticated standard library provided by the developers.

Much work remains to be done. We plan to introduce mechanisms in the AML evaluator that will allow users to implement implicit coercions, type classes, and universes. For a more substantial example we plan to implement Voevodsky's Homotopy Type System [5] as a way of introducing intensional identity types in Andromeda.

References

[1] The Andromeda theorem prover. https://github.com/Andromedans/andromeda/tree/TYPES2016.

[2] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[3] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[4] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984.

[5] Vladimir Voevodsky. A simple type system with two identity types. https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS.pdf, 2013.

[1] In AML, $e_1 \equiv e_2$ can be read as $\mathsf{Eq}_T(e_1, e_2)$, and handlers return their answers using `yield`; see [1] for details.