Design and Implementation of the Andromeda Proof Assistant

Andrej Bauer[1]
University of Ljubljana, Slovenia

Gaëtan Gilbert
École Normale Supérieure de Lyon, France

Philipp G. Haselwarter[2]
University of Ljubljana, Slovenia

Matija Pretnar
University of Ljubljana, Slovenia

Christopher A. Stone
Harvey Mudd College, Claremont, CA 91711, USA

Abstract

Andromeda is an LCF-style proof assistant where the user builds derivable judgments by writing code in a meta-level programming language AML. The only trusted component of Andromeda is a minimalist nucleus (an implementation of the inference rules of an object-level type theory), which controls construction and decomposition of type-theoretic judgments.

Since the nucleus does not perform complex tasks like equality checking beyond syntactic equality, this responsibility is delegated to the user, who implements one or more equality checking procedures in the meta-language. The AML interpreter requests witnesses of equality from user code using the mechanism of algebraic operations and handlers. Dynamic checks in the nucleus guarantee that no invalid object-level derivations can be constructed.

To demonstrate the flexibility of this system structure, we implemented a nucleus consisting of dependent type theory with equality reflection. Equality reflection provides a very high level of expressiveness, as it allows the user to add new judgmental equalities, but it also destroys desirable meta-theoretic properties of type theory (such as decidability and strong normalization).

The power of effects and handlers in AML is demonstrated by a standard library that provides default algorithms for equality checking, computation of normal forms, and implicit argument filling. Users can extend these new algorithms by providing local "hints" or by completely replacing these algorithms for particular developments. We demonstrate the resulting system by showing how to axiomatize and compute with natural numbers, by axiomatizing the untyped λ-calculus, and by implementing a simple automated system for managing a universe of types.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, proof assistant, equality reflection, computational effects

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.5

[1] This material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.
[2] This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

1 Introduction

A type theory can be interesting and very useful, yet lack metatheoretic properties (e.g., decidability) that permit a straightforward implementation. In fact, the more flexible and expressive the theory, the less likely these properties will hold. Nevertheless, even very expressive type theories deserve automated support in the form of proof assistants. The question is how a useful proof assistant can make minimal demands on the properties of the underlying object language. In this paper, we describe the structure of one such system.

Andromeda is an LCF-style proof assistant [12] in which (derivable) judgments are the fundamental data of the system. These judgments are opaque except within a tiny, trusted nucleus that implements rules of the underlying type theory (to construct new judgments from old) and also implements valid inversion principles (to decompose judgments into sub-judgments). The untrusted remainder of the hard-coded system is a small interpreter for AML, an ML-like meta-language [18] extended with algebraic operations and handlers [21].

The AML interpreter builds and decomposes judgments by making (dynamically checked) requests of the trusted nucleus. When these requests would fail (e.g., because a function is being applied to an argument, and in violation of the appropriate typing rule the domain type of the function is not syntactically identical to the type of the argument), the interpreter triggers a suitable algebraic operation to request additional information (e.g., evidence of equality between the mismatched types) from the user.

User-level AML code directs the construction of judgments, and consists of computations that construct and pattern-match judgment values and user-level handlers that intercept and respond to algebraic operations triggered during judgment construction. The consequence of this design is that most proof-assistant functionality – including equality checking, normalization, unification, and proof tactics – is handled at the user level. Effects and handlers allow default implementations (necessarily incomplete for an undecidable object language) that can be overridden using nested handlers, providing specialized algorithms for specific trouble spots.

The specific expressive object language is largely independent of this system design, but some readers may find our chosen type theory independently interesting. The type theory currently implemented in Andromeda is dependent type theory with *equality reflection*, the principle that propositionally equal terms are judgmentally equal:

$$\frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \;\equiv\; t : A}$$

From a mathematical point of view, equality reflection is appealing and natural, as it makes equality in type theory behave like ordinary equality in mathematics. (In Coq, for example, the types "vector of length $0 + n$" and "vector of length n" are equal because $0 + n$ and n are judgmentally equal, but a "vector of length $n + 0$" requires an explicit coercion to be used as a "vector of length n" because $n + 0$ is only propositionally equal to n.) From the perspective of homotopy type theory, equality reflection is suitable for "set-level" mathematics, i.e., those mathematical structures that do not exhibit any higher homotopical phenomena. Among these are substantial parts of algebra, analysis, and logic, including many aspects of meta-theory of type theory.

Building equality reflection into a proof assistant has practical advantages. First, equality reflection lets users axiomatize type-theoretic constructions such as natural numbers with *judgmental* equalities, meaning that we can implement a smaller trusted core type theory

with a wider variety of possible user extensions. Second, applications of equality reflection are not recorded in the conclusion, and omitting the explicit equality eliminators keeps terms smaller and simpler.

The proof assistant NuPRL [2] validated equality reflection by implementing so-called *computational* type theory, a specific interpretation of type theory akin to realizability models. More recently, however, equality reflection has fallen into disrepute among computer scientists and computationally minded mathematicians. It causes the loss of useful meta-theoretic properties such as strong normalization of terms and decidability of type checking [14], the cornerstones of modern proof assistants like Coq [7], Agda [19] and Lean [10]. Even the property "if an application of a lambda abstraction to an argument is well typed, then its β-reduct is well typed" may not hold if the user assumes nonstandard type equalities.

Nevertheless, the use of effects and handlers allows Andromeda to take advantage of equality reflection and to deal with its negative consequences gracefully.

Contributions. The present paper should be read as a progress report on the development of Andromeda; the system and the underlying type theory may evolve as we gain more experience and consider a wider variety of applications. We focus on the following points of interest:

- the goals of Andromeda and the structure of the system (§2);
- the impact of equality reflection on both the design of the type-theoretic nucleus and the details of its implementation (§3, Appendix A);
- features of the meta-language that allow a variety of proof-development techniques to be implemented at the user level (§4);
- a discussion of the soundness of the system (§5);
- a prototype standard library that provides user-extensible equality checking and implicit-argument filling (§6);
- axiomatization of additional type-theoretic structures (dependent sums, natural numbers, untyped λ-calculus, and universes), *with* the desired judgmental equalities and support for automation (§7).

Andromeda is free software, available at `http://www.andromeda-prover.org/`. Contributions, questions, and requests are most welcome.

2 An overview of Andromeda

Andromeda follows design principles that are similar to those of other proof assistants:

- The system should *work well in the common case*. Equality reflection affords many possibilities for complicating one's life, but we expect most applications to be very reasonable. If the user introduces new computation and extensionality rules that play nicely with the existing ones, the system should work smoothly. Nevertheless, less common scenarios should still work, possibly with more effort on the part of the user.
- The user cannot be expected to write down explicit typing annotations on all terms, or hold in their head various bureaucratic matters, such as the typing contexts. Therefore, the *system should take care of low-level details*.
- There should be a *clearly delineated nucleus* that is the only part of the implementation that the user has to trust[3] in order to believe that Andromeda never produces an invalid

[3] Except for trusting the OCaml compiler, the operating system, the hardware, and the absence of malicious cosmic rays.

judgment. The nucleus should be as small as possible and its functionality should implement type theory in the most straightforward way possible.
- A consequence of this minimalism is that the system should be *user extensible*, so that additional functionality can be introduced without breaking trust.

Andromeda is implemented in the tradition of Robin Milner's Logic for Computable Functions (LCF) [12]. The current implementation, in OCaml [20], consists of around 9500 lines of source code, of which the nucleus comprises 1900 lines. These are very low numbers that clearly classify Andromeda as a prototype. However, we do not expect the nucleus to grow significantly.

The core of Andromeda is the trusted nucleus that directly implements inference rules and inversion rules for dependent type theory with equality reflection. By design, it is the only part of the system that can create and manipulate type-theoretic judgments. The nucleus is small and simple, as it does not perform *any* proof search, unification, equality checking, or normalization. (It cannot, since equality checking is in general undecidable and there is no reasonable notion of normal form [14].) Whenever evidence of equality is needed as a premise to an inference rule, it must be provided to the nucleus explicitly.

The user interacts with the system by writing code in the *Andromeda meta-language (AML)*, a general-purpose programming language in the style of ML. AML exposes the nucleus datatype `judgment` as an abstract datatype of its own. Because judgments may only be constructed by the nucleus, neither the OCaml implementation of AML nor any user code written in AML need be trusted. AML handles only trivial syntactic equality checks. All other evidence of equality is obtained from user-level code, through the mechanism of algebraic operations and handlers [22, 3] (§4.3).

Users are free to organize their AML code in any way they see fit. In most cases they would likely want a good axiomatization of standard type constructors (dependent sums, inductive types, universes, etc.), equality checking algorithms that work well in the common cases, and conveniences such as resolution of implicit arguments. These ought to be provided by a standard library (§6). In principle, there may be several such libraries, or even several equality checking algorithms in a single library. The handlers mechanism allows flexible and local uses of several different equality checking algorithms.

AML is statically typed – and this caught many silly errors while we were coding a standard library – but the AML type system is unrelated to the soundness of the system. Bugs in AML code either prevent code from constructing the desired judgments or construct an unintended judgment, but the abstract type of judgments and run-time checks in the nucleus ensure that only derivable judgments are ever constructed. Any other memory-safe metalanguage (e.g., one modeled on Python or Scheme) would be equally sound, if less robust.

Andromeda in action

Before looking at the three constituent parts of Andromeda in more detail, we provide a small worked example. At this point we cannot explain all the technical details, so we focus on emphasizing the important points and showcasing what Andromeda can do.

We begin by declaring some constants that Andromeda adds to the ambient signature:

```
constant A : Type
constant a : A
constant b : A
constant P : A → Type
constant v : P a
```

Andromeda manipulates *only* judgments. Thus the above declaration binds the AML variable `a` to the nucleus judgment ⊢ `a : A`, not to a bare symbol (and similarly for `b`, `v`, `A` and `P`). Nevertheless, it is often convenient to think of a judgment as "a term with a given type, possibly depending on some hypotheses".

Let us first show that the type of transport is inhabited:

```
Π (x y : A), x ≡ y → P x → P y
```

In intensional type theory we would use a J eliminator, but here we should be able to use the curried term `λ x y ξ u,` `u`. Indeed, `u` may be converted from `P x` to `P y` because these are equal types by an application of the congruence rule for applications and equality reflection of `ξ : x ≡ y`. The Andromeda standard library, which is implemented at the user-level, does all this for us if we tell it to use `ξ` as an *equality hint* while checking that `u` has type `P y`:

```
λ (x y : A) (ξ : x ≡ y) (u : P x), (now hints = add_hint ξ in (u : P y))
```

The above is *not* a proof term but an AML computation that generates a judgment. In particular, AML immediately evaluates the command inside the λ. While doing so it will find it needs a witness for the equality between `P x` (the type of `u`) and `P y` (the type ascribed to `u`); it requests one using the operations and handler mechanism. The standard library handles this, employing an equality checking algorithm that eventually uses the hint `ξ` to equate `x` and `y`, and passing back the requested evidence to AML. Then, AML asks the nuclues to apply equality reflection to obtain the judgmental equality of `P x` and `P y` (at which point the equality witness provided by the standard library is discarded) and to apply conversion. The interaction betwen AML and the nuclues proceeds in this fashion until the judgment witnessing transport is constructed.

If we need to write `add_hint` to guide the type checker, one might ask why this is better than the intentional approach of applying a `J` eliminator with `ξ` to coerce `u` from `P x` to `P y`. A single `add_hint` is like the incorporation of a computation rule that can handle an arbitrarily complex development, whereas `J` is like a single application of a computation rule that has to be repeated at every point where the rule is to be applied.

The other benefit, apparent even here, is that without `J` the proof term is smaller. In the end, the judgment built by the nucleus is the expected one:

```
⊢ λ (x : A) (y : A) (_ : x ≡ y) (u : P x), u
  : Π (x : A) (y : A), x ≡ y → P x → P y
```

Although `ξ` does not appear explicitly in the conclusion, Andromeda is aware it was used. This tracking process becomes apparent if we *temporarily* hypothesize an equality `a ≡ b` and use it as a hint while constructing a judgment that `v` above has type `P b`,

```
assume ζ : a ≡ b in
  now hints = add_hint ζ in (v : P b)
```

This AML expression causes the nucleus to build the *hypothetical* judgment:

```
ζ₀ : a ≡ b ⊢ v : P b
```

The `assume` construct generated a fresh variable ζ_0 of type `a ≡ b` and bound the AML variable ζ to the judgment `ζ₀ : a ≡ b ⊢ ζ₀ : a ≡ b`. Because ζ_0 was used to convert `P a` to `P v`, the nucleus produced a judgment that depends on it.

The AML interpreter communicates with user-level code by invoking operations to be handled, but user-level operations are useful as well. Let us define a simple `auto` tactic for automatically inhabiting simple types. We first need an AML function, called `derive`, which

attempts to inhabit a given type from the currently available hypotheses by performing a recursive search. This takes about 40 lines of uneventful code, shown in Appendix B. Then we declare a new operation that takes no arguments and yields judgments,

```
operation auto : judgment
```

and install a global handler that handles it:

```
handle
| auto : ?Surr ⇒
    match Surr with
    | Some ?T ⇒ derive T
    | None    ⇒ failure
    end
end
```

When the handler intercepts the operation `auto`, the surroundings of the occurrence of `auto` may or may not have indicated an expected result type `T`. If it does, the handler calls `derive T` to inhabit the type, otherwise it fails by triggering the operation `failure` (also defined in the appendix) because it has no information on what type to inhabit.

Now we can use `auto` to inhabit types. For example,

```
λ (X : Type), (auto : X → X)
```

constructs the judgment

```
⊢ λ (X : Type) (x : X), x
  : Π (X : Type), X → X
```

Given types `A`, `B`, and `C`, the computation

```
auto : (A → B → C) → (A → B) → (A → C)
```

results in the judgment

```
⊢ λ (x : A → B → C) (x0 : A → B) (x1 : A), x x1 (x0 x1)
  : (A → B → C) → (A → B) → A → C
```

The Andromeda standard library (§6) takes full advantage of operations and handlers to produce equality proofs and coercions, with default implementations that users can override with local handlers when the standard heuristics fail.

3 The nucleus

The nucleus is the part of the system that implements the object-level type theory. Its functionality includes the following:

- formation and decomposition of term and type judgments,
- construction of equality judgments,
- substitution and syntactic equality checking,
- pretty-printing of judgments and export to JSON.

Before discussing some of these features we take a closer look at the type theory implemented by Andromeda, and engineering issues that it raises.

3.1 Type theory with equality reflection

The Andromeda nucleus implements an extensional Martin-Löf type theory [17, 14] with dependent products $\prod_{(x:A)} B$ and equality types $\mathsf{Eq}_A(s,t)$. Complete rules are provided in Appendix A. Fundamentally, the system is not too far removed from the more common intensional Martin-Löf type theory, but instead of a J eliminator for equality types, we have equality reflection and uniqueness of equality terms:

$$\textsc{eq-reflection}\quad \frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \;\equiv\; t : A} \qquad \textsc{eq-eta}\quad \frac{\Gamma \vdash t : \mathsf{Eq}_A(s,u) \qquad \Gamma \vdash v : \mathsf{Eq}_A(s,u)}{\Gamma \vdash t \;\equiv\; v : \mathsf{Eq}_A(s,u)}$$

The J eliminator can easily be derived from these rules, but direct use of equality reflection is generally simpler. Streicher's K eliminator and uniqueness of identity proofs [25] are also derivable in this setting.

Equality reflection invalidates some common structural rules and inversion principles, so we make further small changes to the type theory to compensate. First, it is usual for products to satisfy an injectivity property, i.e., if $\prod_{(x:A_1)} A_2$ and $\prod_{(x:B_1)} B_2$ are equal then A_1 equals B_1 and A_2 equals B_2. But in our type theory injectivity fails because under the assumption

$$p : \mathsf{Eq}_{\mathsf{Type}}((\mathsf{Nat} \to \mathsf{Nat}), (\mathsf{Nat} \to \mathsf{Bool})) \tag{1}$$

$\mathsf{Nat} \to \mathsf{Nat}$ and $\mathsf{Nat} \to \mathsf{Bool}$ are equal by reflection, *without* equality of Nat and Bool.[4] This may seem a very technical point, but usually one relies on injectivity to prove that β-reductions preserve types. Indeed, under assumption (1) the identity function on Nat also has type $\mathsf{Nat} \to \mathsf{Bool}$, and hence by applying it to 0 and β-reducing, we can show that 0 has type Bool, even though Nat and Bool are not equal.

Andromeda's solution, following [14], is to add explicit typing annotations that can typically be omitted in intentional type theories. A λ-abstraction $\lambda x{:}A.B\,.\,t$ is annotated not only with the domain A of the bound variable but also with the type B of the body t, and an application $s\ @^{x:A.B}\ t$ is similarly annotated with the type of the function being applied. These annotations ensure that terms have unique types up to equality: working again under the assumption (1), we can apply the identity function at type $\mathsf{Nat} \to \mathsf{Nat}$ to get $(\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x)\ @^{x:\mathsf{Nat}.\mathsf{Nat}}\ 0$ of type Nat, or at $\mathsf{Nat} \to \mathsf{Bool}$ to get $(\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x)\ @^{x:\mathsf{Nat}.\mathsf{Bool}}\ 0$ of type Bool. Crucially, the typing annotations now prevent the latter term from β-reducing to 0, as the β-rule requires that the function and the application match:

$$\textsc{prod-beta}\quad \frac{\Gamma,\, x : A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x{:}A.B\,.\,s)\ @^{x:A.B}\ t \;\equiv\; s[t/x] : B[t/x]}$$

Another principle that fails in the presence of equality reflection is *strengthening*, which says that we may safely remove from the context any hypothesis that is not explicitly mentioned in the conclusion of a judgment. Indeed,

$$p : \mathsf{Eq}_{\mathsf{Type}}(\mathsf{Nat} \to \mathsf{Nat}, \mathsf{Nat} \to \mathsf{Bool}) \vdash (\lambda x{:}\mathsf{Nat}.\mathsf{Nat}\,.\,x)\ @^{x:\mathsf{Nat}.\mathsf{Bool}}\ 0 : \mathsf{Bool}$$

[4] The assumption that the Cantor space and the Baire space are equal may seem odd, but it is consistent. For instance, in classical set theory and in the effective topos the two are isomorphic, and with a little work we can arrange them to be equal.

becomes invalid if we remove p, even though there is no explicit use of p in the conclusion. For similar reasons *exchange* is not valid: given types X and Y, the context

$$x : X, p : \mathsf{Eq}_{\mathsf{Type}}(X, Y), q : \mathsf{Eq}_{Y}(x, x)$$

becomes invalid if we exchange the order of p and q, even though their types do not refer to each other. The loss of strengthening and exchange is inconvenient; we discuss an implementation-level solution in §3.2.

Perhaps the biggest difference between Andromeda and standard type theory is that we currently postulate a single universe Type and the rule that makes Type an element of itself:

$$\textsc{ty-type}\ \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}$$

From a logical point of view this is an inconsistent assumption, as Girard's paradox implies that every type is inhabited [11]. From an engineering point of view, however, $\mathsf{Type} : \mathsf{Type}$ is very useful. For Andromeda implementers, it allows a much simpler implementation strategy with fewer different judgment forms. For Andromeda users, it allows postponing the complexities of dealing with type universes and universe levels, and instead focus on other aspects of derivations. (For the same reason, both Coq and Agda allow the assumption $\mathsf{Type} : \mathsf{Type}$ as an option.) Nevertheless, although users are unlikely to stumble into inconsistencies by accident, we ultimately want a sound foundation, and plan to remove TY-TYPE, as discussed in §9.

3.2 Implementation of type theory

The type theory implemented in the nucleus differs from the one presented in several ways. The changes are inessential from a theoretical point of view, but have significant practical impact. We describe them in this section.

Signatures

In Andromeda the user extends the type theory by postulating constants, i.e., they work in type theory over a *signature*. In this respect Andromeda is much like other proof assistants that allow the user to state axioms and postulates. The signature is controlled by the nucleus through an abstract datatype whose interface is very simple: there is an empty signature, and a signature may be extended with a new constant of a given *closed* type (which may refer to the previously declared constants). Because signatures are ever increasing, judgments derived over a signature remain valid when the signature changes.

Inversion principles and natural types

The nucleus implements inversion principles for deconstruction of judgments into subjudgments; these are exposed in AML through pattern matching, cf. §4.4. For example, an application $\Gamma \vdash s\ @^{x:A.B}\ t : C$ can be decomposed into $\Gamma \vdash s : \prod_{(x:A)} B$, $\Gamma \vdash t : A$ and $\Gamma \vdash C\ \mathsf{type}$. The type-theoretic justification for this operation is an *inversion principle*: if $\Gamma \vdash s\,@^{x:A.B}\,t : C$ is derivable then so are $\Gamma \vdash s : \prod_{(x:A)} B$, $\Gamma \vdash t : A$ and $\Gamma \vdash C\ \mathsf{type}$. Proving the principle is not hard but neither is it a complete triviality, because the application could have been formed with the use of type conversions. Similar inversion principles hold for other term and type formers.

If we decompose an application, as above, and put it back together using the application formation rule, we get the judgment $\Gamma \vdash s \mathbin{@^{x:A.B}} t : B[t/x]$. The application has received its "natural type", which may not be the original type C. Nevertheless, the types are equal by *uniqueness of typing*:

If $\Gamma \vdash t : A_1$ *and* $\Gamma \vdash t : A_2$ *then* $\Gamma \vdash A_1 \equiv A_2$.

The nucleus provides evidence of uniqueness of typing by generating, given $\Gamma \vdash t : A$, a witness for equality of A and the *natural type* $\mathcal{N}(t)$ of t, which is read off the typing annotations:

$$\begin{aligned} \mathcal{N}(\mathsf{Type}) &= \mathsf{Type} & \mathcal{N}(\textstyle\prod_{(x:A)} B) &= \mathsf{Type} \\ \mathcal{N}(\mathsf{Eq}_A(s,t)) &= \mathsf{Type} & \mathcal{N}(\lambda x{:}A.B\,.\,t) &= \textstyle\prod_{(x:A)} B \\ \mathcal{N}(s \mathbin{@^{x:A.B}} t) &= B[t/x] & \mathcal{N}(\mathsf{refl}_A\ t) &= \mathsf{Eq}_A(t,t) \end{aligned}$$

The natural types of variables and constants are read off the context and the signature, respectively. Note that the natural type is the one we get if we deconstruct a term judgment and construct it back again. In the standard library the equality of the original type and the natural type is needed in several places during equality checking.

Assumption sets

The nucleus is responsible for decomposing judgments into their component parts, a facility used by pattern matching in AML. For example, we can combine

$$f : \mathsf{Nat} \to \mathsf{Nat} \vdash f : \mathsf{Nat} \to \mathsf{Nat} \qquad \text{and} \qquad x : \mathsf{Nat} \vdash x : \mathsf{Nat}$$

(using weakening and application) to get

$$f : \mathsf{Nat} \to \mathsf{Nat},\ x : \mathsf{Nat} \vdash f \mathbin{@_{-}^{:\mathsf{Nat}.\mathsf{Nat}}} x : \mathsf{Nat},$$

But if we naively pattern-match on this application to get the function part and the argument part (as judgments), we would get the constituents in weakened form

$$f : \mathsf{Nat} \to \mathsf{Nat},\ x : \mathsf{Nat} \vdash f : \mathsf{Nat} \to \mathsf{Nat} \qquad \text{and} \qquad f : \mathsf{Nat} \to \mathsf{Nat},\ x : \mathsf{Nat} \vdash x : \mathsf{Nat}.$$

In a system with strengthening, we could immediately see that x is unnecessary in the first judgment and f in the second. The loss of strengthening is inconvenient enough that we restore it by explicitly keeping track of dependencies on the assumptions in the context.

In the implementation we use *terms with assumptions*, which are ordinary terms that have every subterm annotated with a set of variables, called the *assumptions*, indicating explicitly which part of the context a subterm depends on. Thus $\Gamma \vdash t^{\alpha} : A^{\beta}$ means that we may restrict Γ to variables in α to obtain a smaller context $\Gamma\restriction_{\alpha}$ in which it is still possible to show that t has type A. Similarly, $\Gamma\restriction_{\beta}$ suffices to derive the judgment that A is a type. The types and terms appearing in the context are themselves annotated with assumptions, which endows contexts with the structure of directed acyclic graphs. (In the implementation they are stored as such.)

This means that Andromeda can compose and decompose judgments without information loss. The application above will be recorded internally as:

$$f : (\mathsf{Nat}^{\emptyset} \to \mathsf{Nat}^{\emptyset})^{\emptyset},\ x : \mathsf{Nat}^{\emptyset} \vdash (f^{\{f\}} \mathbin{@_{-}^{:\mathsf{Nat}^{\emptyset}.\mathsf{Nat}^{\emptyset}}} x^{\{x\}})^{\{f,x\}} : \mathsf{Nat}^{\emptyset}.$$

and it is straightforward to recover the two original sub-judgments.

Constants from the signature are not included in assumption sets, since they are omnipresent anyhow.

Context joins

The standard rules of inference require the contexts of the premises to match, for instance the application rule TERM-APP does not allow a change of the context:

$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : A}{\Gamma \vdash s \mathbin{@}^{x:A.B} t : B[t/x]}$$

If we implemented the rule exactly as is, the user would have to plan dependence on hypotheses carefully in advance, which is impractical. Instead, we rely on admissibility of weakening to enlarge contexts as necessary. In every inference rule we accept premises with arbitrary contexts that are then *joined* to form a single extended context, for instance the application rule becomes

$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Delta \vdash t : A}{\Gamma \bowtie \Delta \vdash s \mathbin{@}^{x:A.B} t : B[t/x]}$$

The context join $\Gamma \bowtie \Delta$ is the smallest context that extends both Γ and Δ. In terms of directed acyclic graphs it is just the union of graphs. A context join fails if there is a hypothesis that has different types in Γ and Δ, or if the join would create a cyclic dependency of hypotheses. In practice such failures are infrequent; λ, Π, and `assume` create globally fresh object-level variables, so there is no direct way to create two contexts with the same variable at different types.[5]

Andromeda automatically tracks assumption sets and contexts. Even though the implementation makes an effort to keep them small, they may not be unique or minimal: they merely reflect a history of how judgments were constructed.

4 The Andromeda meta-language

The Andromeda meta-language (AML) is a programming language in the style of ML [18]. We review its structure and capabilities, focusing on the parts that are peculiar to Andromeda. For constructs that are standard in the ML-family of languages, such as type definitions, `let`-bindings, recursive functions, etc., we refer the reader to the Andromeda reference page.[6]

In order to distinguish the expressions of AML from the expressions of the object-level type theory, we refer to the former as *computations* to emphasize that their evaluation may have side effects (such as printing things on the screen), and to the latter as *(type-theoretic) terms*. We refer to the types of AML as *ML-types*.

Keep in mind that the ML-level computations can never enter the object-level terms, as the nucleus knows nothing about AML. What looks like AML code inside an object-level term is *always* just AML code that constructs a judgment. For example, a pattern match inside a λ-abstraction, λ`(x:A), match ... end`, is a computation that evaluates the `match` statement immediately to obtain an object-level term, which is then abstracted. In contrast, the ML-level function `fun x ⇒ match ... end` does suspend the evaluation of its body.

[5] An indirect method to obtain unjoinable contexts is to take a single judgment with the context X:Type, x:X and explicitly substitute in two different ways, replacing X with two distinct types.

[6] `http://www.andromeda-prover.org/meta-language.html`

4.1 ML-types

AML is equipped with static type inference in the style of Hindley-Milner parametric polymorphism [9]. It supports definitions of parametric ML-types, including inductive types. The only non-standard aspect of the ML-type inference arises from the fact that application is overloaded, as it is used both for invoking ML-level application and for building object-level applications. For instance the type of `f`, defined by

```
let f x y = x y
```

could be either `judgment` $\to$ `judgment` $\to$ `judgment` or $(\alpha \to \beta) \to \alpha \to \beta$; in such cases the inferred type constraints are postponed until we are sure that `x` will be a judgment or that at least one of `x` and `y` will not. This strategy works well in practice, with only the occasional application constraint remaining unresolved at the top level.

4.2 Pattern matching

AML pattern matching in `match` statements and `let`-bindings is more flexible than that of Standard ML and related languages. AML patterns need not be linear (i.e., a pattern variable may appear several times in a pattern) and variables may be interpolated into patterns. Pattern variables are prefixed with `?` so that they can be distinguished from interpolated variables. For example,

- the pattern `(?x, ?y)` matches any pair,
- the pattern `(?x, ?x)` matches a pair whose components are equal,
- the pattern `(?x, y)` matches a pair whose second component equals the value of `y`.

Equality in pattern matching always means syntactic identity (α-equivalence in the case of object-level terms), not arbitrary judgmental equalities. The flexibility of pattern matching is handy when we match on values of type `judgment`; see §4.4, where we also discuss patterns for deconstruction of typing judgments.

4.3 Operations and handlers

During evaluation of a computation of ML-type `judgment` the interpreter may need evidence of equality between two types (in order to present it to the nucleus), which it gets by passing control back to user code, together with information on what needs to be done, and how to resume the evaluation once the evidence is obtained. To accomplish this, AML is equipped with algebraic operations and handlers [22] in the style of Eff [3]. We recommend [23, 3] for background reading, and give just a quick overview here. A more detailed discussion on the use of algebraic operations and handlers for the purposes of computing judgments can be found in §4.4.

One way to think of an operation is as a generalized resumable exception: when an operation is invoked it "propagates" outward to the innermost handler that handles it. The handler may then perform an arbitrary computation, and using `yield` c it may resume the execution at the point at which the operation was invoked, yielding the value of c as the result of the operation. Similarly, we can think of a handler as a generalized exception handler, except that it handles one or more operations, as well as values (computations which do not invoke an operation). An example of handlers in action is given in §7.1.

4.4 The datatype judgment

In Andromeda the user *always works with an entire judgment* $\Gamma \vdash t : A$, and never a bare term t. Similarly a type A never stands by itself, but always in a judgment $\Gamma \vdash A : \mathsf{Type}$. The judgments are represented by values of a special primitive type `judgment`.

Judgment forms

The OCaml interface for the nucleus uses distinct abstract datatypes to represent the different judgment forms. These distinctions are not visible to the user, because AML exposes all forms through the single datatype `judgment` whose values are judgments of the form $\Gamma \vdash t : A$. This is possible because $\mathsf{Type} : \mathsf{Type}$ and equality reflection let us express the three forms $\Gamma \vdash A\ \mathsf{type}$, $\Gamma \vdash s \equiv t : A$, and $\Gamma \vdash A \equiv B$ as $\Gamma \vdash A : \mathsf{Type}$, $\Gamma \vdash p : \mathsf{Eq}_A(s,t)$, and $\Gamma \vdash q : \mathsf{Eq}_{\mathsf{Type}}(A,B)$, respectively.

We hope the user finds it simpler to access all object-level entities in a uniform way. On the other hand, having more precise judgment types in AML would help catch potential errors. We discuss this particular design choice in §9.

In AML no direct datatype constructors for `judgment` are available. (Even at the level of OCaml implementation the datatype constructors are invisible outside the nucleus.) Instead, the user may invoke primitive computations of type `judgment` that *look like* term constructors, but really correspond to inference rules of type theory. For instance, an application $c_1\, c_2$, where c_1 and c_2 are computations of type `judgment`, computes an instance of the TERM-APP rule (actually, the version with context joins). The user does not have to provide the explicit typing annotations on the application, as these are derived using a bidirectional typing strategy, as described next.

Inferring and checking modes of evaluation

There are two modes of AML evaluation, *inferring* and *checking*. In inferring mode the type of the result is unconstrained. In checking mode the type is prescribed in advance: there is given a type A (or more precisely, a judgment $\Gamma \vdash A : \mathsf{Type}$) and the computation must evaluate to a judgment of the form $\Delta \vdash t : A$ where Δ extends Γ.

For instance, an application $c_1\, c_2$ is evaluated in inferring mode as follows. First c_1 is evaluated in inferring mode to $\Gamma \vdash s : \prod_{(x:A)} B$ (we discuss what happens if the type of s is not a product below), then c_2 is evaluated in checking mode at type A to $\Delta \vdash t : A$, and the result is $\Gamma \bowtie \Delta \vdash s\ @^{x:A.B}\ t : B[t/x]$.

Judgment computations

The following primitives for computing judgments are provided:

- Primitives for term and type formation:

  `Type` $\quad \Pi(x{:}c_1),\, c_2 \quad c_1\, c_2 \quad \lambda(x{:}c_1),\, c_2 \quad c_1 \equiv c_2 \quad$ `refl` c.

  Note that the notation $c_1 \equiv c_2$ is used for the equality type, rather than for judgmental equality, which the user never writes down explicitly. We emphasize again that these are *not* datatype constructors for forming terms and types of the object-level type theory, but primitive *computations*, with possible side effects, that build *judgments* from sub-judgments by passing through the nucleus.
- Type ascription $c_1 : c_2$, which first evaluates c_2 to $\Gamma \vdash A : \mathsf{Type}$ and then evaluates c_1 in checking mode at type A.

- Top-level `constant` declarations, which introduce new constants.
- The computation `assume` $x : c_1$ `in` c_2, which evaluates c_1 in inferring mode to $\Gamma \vdash A$: Type, and then c_2 with x `let`-bound to $\Gamma, x_i : A \vdash x_i : A$, where x_i is a freshly generated name. This should not be confused with constant declarations: a constant is an omnipresent part of the signature, while an assumption is local to a judgment in which it appears, and is tracked in assumption sets. Furthermore, we may replace an assumption with a term, using the substitution primitive, but not a constant.
- Substitution c_1 `where` $x = c_2$, which replaces x with the value of c_2 in the value of c_1, assuming the types match.
- The computation `occurs` $c_1\, c_2$, which evaluates c_1 to a judgment $\Delta \vdash x : A$ and c_2 to a judgment $\Gamma \vdash t : B$, and checks whether x appears in Γ. It returns `None` if not, and `Some`$(\Xi \vdash C$: Type$)$ if x appears in Γ as a variable of type C.
- Computations that generate witnesses for the β-rule and the congruence rules. There are no primitive computations for extensionality rules EQ-ETA and PROD-ETA because they can be declared with a constant by the user. Indeed, we do so in the standard library.
- The computation `natural` c, which witnesses uniqueness of typing. It evaluates c to a judgment $\Gamma \vdash t : A$ and outputs a witness for equality of A and the natural type $\mathcal{N}(t)$ of t. The witnesses are needed when a tactic deconstructs a term and puts it back together, thus obtaining the original term at its natural type.

Judgment patterns

Apart from computations that form judgments, we also need flexible ways of analyzing and deconstructing them. In AML this is done with the `match` statement and judgment patterns of the form $\vdash p_1 : p_2$, where p_2 may be omitted, and p_1 and p_2 are among the following:

- Anonymous pattern `_`, pattern variables $?x$, and interpolated variables x.
- Patterns for matching terms and types:

  Type $\quad \Pi(?x{:}p_1),\, p_2 \quad p_1\, p_2 \quad \lambda(?x{:}p_1),\, p_2 \quad p_1 \equiv p_2 \quad$ `refl` p.

  Note that the patterns for products and abstractions "open up" the binders so that it is possible to pattern-match under the binders; the judgment matched by p_2 can have the bound variable in its context.
- Patterns for matching free variables `_atom` $?x$ and constants `_constant` $?x$.

More precisely, when $\Gamma \vdash t : A$ is matched with $\vdash p_1 : p_2$, the term t is matched with p_1 and the type A with p_2. Assuming the match succeeds, the pattern variables in p_1 and p_2 are bound to sub-judgments that are obtained through inversion lemmas §3.2. The contexts of the sub-judgments are kept minimal thanks to assumption sets. Examples of pattern matching are shown in Appendix B.

Pattern matching is always executed at the AML level; patterns and the `match` statements exist only as computations, and are not part of the object-level terms. To highlight this point, we show the difference between a `match` inside λ-abstraction and an AML function. Assuming a type `A` with two constants `a, b : A` and an endofunction `f : A → A`, the computation

```
(λ (x : A), match x with
              | ⊢ ?g ?y ⇒ y
              | ⊢ _     ⇒ b
            end             ) (f a)
```

evaluates to the judgment $\vdash$ `(λ (x : A), b) (f a) : A`, while

```
(fun x ⇒ match x with
           | ⊢ ?g ?y ⇒ y
           | ⊢ _     ⇒ b
         end             ) (f a)
```

evaluates to the judgment $\vdash$ `a : A`. In the former case matching occurred inside the abstraction, so `x` evaluated to $\vdash$ `x : A` and the second clause matched; in the latter case matching took place when the function was applied, so `x` was bound to $\vdash$ `f a : A` and the first clause matched.

The AML interpreter matches a judgment against a pattern by first asking the nucleus to invert the judgment. The nucleus returns information about which inversion was used and what constituent parts it produced, from which the interpreter calculates whether the pattern matches and how. If there are sub-patterns, the process continues recursively. Pattern matching uses syntactic equality (up to α-equivalence) and never triggers any operations, although an inexhaustive `match` may fail.

At present there are no judgment patterns for analyzing the context of a judgment. Instead, the primitive computation `context` c evaluates c to a judgment $\Gamma \vdash t : A$ and gives the list of all hypotheses in Γ, sorted so that each hypothesis is preceded by its dependencies.

Equality checks and coercions

AML only verifies syntactic equality automatically. It delegates any other equality $\Gamma \vdash s \equiv t : A$ by triggering the operation `equal` $(\Gamma \vdash s : A)$ $(\Gamma \vdash t : B)$, which passes control back to the user-level AML code. The operation may go unhandled, in which case an error is reported, or it may be intercepted by a handler in the user code. The handler may do whatever it wants, but the intended use is for it to attempt to calculate evidence of the given equality. The handler yields `None` if it fails to compute the evidence (in which case the interpreter reports an error), or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_A(s,t))$ if it finds a witness $\Delta \vdash \xi : \mathsf{Eq}_A(s,t)$. Note that the handler is itself a piece of AML code that may recursively trigger further operations and handling thereof.

Apart from equality checking, there are other situations in which the AML interpreter triggers an operation:

- It may happen that AML needs to know why a given type $\Gamma \vdash A : \mathsf{Type}$ is equal to a product type. Unless A is already syntactically equal to a product type, the interpreter triggers an operation `as_prod` $(\Gamma \vdash A : \mathsf{Type})$. It expects a handler to yield `None` upon failure, or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \prod_{(x:B)} C))$ witnessing that A is equal to a product type.
- Similarly, if AML needs to know why $\Gamma \vdash A : \mathsf{Type}$ is equal to an equality type, it triggers an operation `as_eq` $(\Gamma \vdash A : \mathsf{Type})$. It expects the handler to yield `None` or `Some`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \mathsf{Eq}_B(s,t)))$.
- If an inferring term evaluates to $\Gamma \vdash t : A$ in checking mode at type $\Delta \vdash B : \mathsf{Type}$, the interpreter does *not* ask for evidence that A and B are equal, but instead triggers the operation `coerce` $(\Gamma \vdash t : A)$ $(\Delta \vdash B : \mathsf{Type})$ that gives the user code an opportunity to replace t with a value of type B. The handler must yield:
  - `NotCoercible` to indicate failure to coerce t to B,
  - `Convertible`$(\Xi \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A,B))$ to indicate that A and B are equal, so that AML may apply conversion to t, or
  - `Coercible`$(\Xi \vdash s : B)$ to have t replaced with s.

  This mechanism allows the user to implement various strategies for coercion of values, and control them completely through handlers.

- If the head c_1 of an application $c_1\, c_2$ evaluates to a term $\Gamma \vdash t : A$ where A is not a product type, the interpreter asks the user code to convert t to a function by triggering the operation `coerce_fun` $(\Gamma \vdash t : A)$. The handler should yield `NotCoercible`, `Convertible`$(\Delta \vdash \xi : \mathsf{Eq}_{\mathsf{Type}}(A, \prod_{(x:B)} C))$, or `Coercible`$(\Delta \vdash s : \prod_{(x:B)} C)$, as the case may be.

4.5 References and dynamic variables

As a convenience, AML provides ML-style mutable references. They are used to store the current state of implicit arguments in the standard library (see §6.3).

AML also supports *dynamic variables*. These are globally defined mutable values with dynamic binding discipline. A dynamic variable x is declared and initialized with the top-level command `dynamic` $x = c$. The computation `now` $x = c_1$ `in` c_2 changes x to the value of c_1 locally in the computation of c_2.

AML maintains a dynamic variable `hypotheses`. It is a list of judgments that plays a role in evaluation of computations under binders. To evaluate $\lambda(x{:}A),\, c$, AML generates a fresh variable x_i of type A, binds x to the judgment $x_i : A \vdash x_i : A$, prepends it to `hypotheses`, evaluates c, and abstracts x_i to get the final result. By accessing `hypotheses` the computation c may discover under what binders it is evaluated. For example, in §2 the handler for the `auto` tactic searched `hypotheses` for ways of inhabiting a type.

The standard library uses dynamic variables `betas`, `etas`, `hints`, and `reducing` to store β-hints, η-hints, general hints, and reduction directives. It is important for these variables to follow a dynamic binding discipline so that *local* equality hints work correctly (see §6.2).

5 Soundness of Andromeda

Soundness in Andromeda has both theoretical and engineering aspects.

Theoretical soundness pertains to the differences between the original type theory, (Appendix A) and the type theory implemented in the nucleus (§3.2), which uses assumption sets, context joins, and natural types. In the following we write s^σ for a term s decorated with assumptions σ, i.e., if we remove assumptions sets from the decorated term s^σ we get the ordinary term s. We follow a similar convention for types and context.

▶ **Claim 5.1.** *Given a context Γ, a term s and a type A:*

1. *If $\Gamma \vdash s : A$ is derivable in the original type theory, then $\Delta^\delta \vdash s^\sigma : A^\alpha$ is derivable for some Δ^δ, s^σ, and A^α such that Δ is a subcontext of Γ.*
2. *If $\Gamma^\gamma \vdash s^\sigma : A^\alpha$ is derivable in the implemented type theory, then $\Gamma \vdash s : A$ is derivable in the original type theory.*

We cannot call the statement a theorem because we have not yet proved it in detail. We leave the task as future work for this progress report, and note that we do not anticipate a particularly enlightening or difficult proof, just the usual grinding of cases by structural induction. The most interesting part of the proof will likely by the formalization of the implemented type theory from §3.2, which we have postponed because it has been regularly modified as we gained experience with the implementation.

The second aspect of soundness is an engineering question: how do we know that the implementation of Andromeda works as intended?

▶ **Claim 5.2.** *If Andromeda evaluates a computation to a judgment, then the judgment is derivable from the implemented type theory with respect to the signature containing all the constants declared by the user.*

Let us reiterate the design choices we have made to give credence to the claim. In the OCaml implementation the datatypes representing the judgment forms are all abstract and kept opaque by an interface to a small trusted OCaml module, the *nucleus*. We rely on the soundness of OCaml's type system to ensure that the untrusted remainder of the system cannot forge new values of these abstract types.[7] The nucleus is kept as simple as possible, and it only supports very straightforward type-theoretic constructions which directly correspond to applications of inference rules and admissible rules. Everything else, the AML type inference, the core AML intepreter, the implementation of operations and handlers, and the user code, is on the other side of the barrier and does not influence soundness. In particular, the nucleus does not know anything about AML at all, does not trigger operations, and no AML code can ever enter the object-level terms (so there is no question about having pattern matching, exotic terms involving AML code, or any other part of AML at the object level).

Formally verifying the 1900 lines of the nucleus code is still a tall order to handle, and at present we have no plans to do it. Once we have formulated the implemented type theory, a careful code review of the nucleus will probably unearth some bugs, and hopefully not very many!

There is a third kind of soundness, namely the consistency of the underlying type theory. Obviously, since we included $\mathsf{Type} : \mathsf{Type}$ the theory is at present inconsistent in the sense that all types are inhabited and all judgmental equalities derivable. As soon as we remove $\mathsf{Type} : \mathsf{Type}$ the theory becomes consistent, since what remains are just bare products and equality types with reflection, and these are consistent in virtue of having a model (such as the hereditarily finite sets). We discuss removal of $\mathsf{Type} : \mathsf{Type}$ in §9.

6 The standard library

To test the viability of our design we implemented a small standard library in AML. By design, anything that is implemented in AML is safe: it may not work as expected, or diverge, but it will never produce an invalid judgment, or derive an invalid equality. (Of course, this does not say much until we have dealt with $\mathsf{Type} : \mathsf{Type}$.)

6.1 Equality checking

The most substantial part of the library is a user-extensible equality checking algorithm with rudimentary support for implicit arguments, based on similar ones by Stone and Harper [24] and Coquand [8]. It computes a witness of equality $\Gamma \vdash s \equiv t : A$ in two phases:

- The *type directed* phase computes the weak head-normal form (whnf) of type A to see whether any extensionality rules apply. For instance, if A normalizes to a product $\prod_{(x:B)} C$, the algorithm applies function extensionality PROD-ETA to reduce the equality to $\Gamma, y : B \vdash s \mathbin{@^{x:B.C}} y \equiv s \mathbin{@^{x:B.C}} y : B[y/x]$ at a smaller type. Similarly, if A is an equality type the equality checks succeeds immediately by uniqueness of equality proofs EQ-ETA. Extensionality rules including PROD-ETA and EQ-ETA are user defined (see §6.2).
- Once the type-directed phase simplifies the type so that no further extensionality rules apply, the *normalization phase* computes the weak head-normal forms of s and t and compares them structurally, which generates new equality problems involving subterms.

[7] If we were to reimplement the system in an unsafe language such as C, or if we lacked faith in OCaml, additional mechanisms such as cryptographic signatures could be used.

The equality checking algorithm relies on the computation of weak head-normal forms of terms, which is also implemented by the standard library. Given a term $\Gamma \vdash t : A$, the library computes a witness $\Gamma \vdash \xi : \mathsf{Eq}_A(t, t')$ where t' is in weak head-normal form. It does so by chaining together a sequence of *computation rules* using transitivity of equality. By default the only computation rule is PROD-BETA for reducing β-redices, but the user may install additional rules as explained in §6.2.

6.2 Equality hints

The equality checking algorithm can be extended by the user with new rules, which we call *equality hints*. There are three kinds:

- an *η-hint*, or an extensionality hint, is a term whose type has the form

  $$\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \prod_{(y_1:B)} \prod_{(y_2:B)} \mathsf{Eq}_{C_1}(t_1, s_1) \to \cdots \to \mathsf{Eq}_{C_m}(t_m, s_m) \to \mathsf{Eq}_B(y_1, y_2).$$

  It is a universally quantified equation with equational preconditions, where the left-hand and the right-hand side of the equation are distinct variables. The equality checking algorithm matches such a hint against the goal. If the match succeeds, the goal is reduced to deriving the preconditions.
- a *β-hint*, or a computation hint, is a term whose type is a universally quantified equation

  $$\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \mathsf{Eq}_C(s, t).$$

  The weak head-normal form algorithm matches the left-hand side s of the equation against the term. If the match succeeds, it performs a reduction step from s to t.
- a *general hint* is a term whose type is a universally quantified equation

  $$\prod_{(x_1:A_1)} \cdots \prod_{(x_n:A_m)} \mathsf{Eq}_C(s, t).$$

  The equality checking algorithm matches such a hint against the goal during the type directed phase to see whether it can immediately dispose of the goal.

In addition, the user may give a *reduction strategy* for a given constant by specifying which of its arguments should be reduced eagerly. This is necessary for the equality checking algorithm to work correctly when we introduce new eliminators. For instance, when we axiomatize simple products $A \times B$, the extensionality rule

$$\prod_{(A:\mathsf{Type})} \prod_{(B:\mathsf{Type})} \prod_{(x,y:A\times B)} \mathsf{Eq}_A(\mathtt{fst}\, A\, B\, x, \mathtt{fst}\, A\, B\, y) \to \mathsf{Eq}_A(\mathtt{snd}\, A\, B\, x, \mathtt{snd}\, A\, B\, y) \to \mathsf{Eq}_{A\times B}(x, y)$$

only works correctly if we also specify that the normal form of a projection $\mathtt{fst}\, A\, B\, t$ should have t normalized, and similarly for `snd`. Another example is the recursor for natural numbers, which should eagerly reduce the number at which it is applied.

Examples of equality hints and uses of reduction strategies will be shown in §7. Let us only remark that the hints and reduction strategies may be installed locally, even under a binder using a temporary equality assumption, and that the user is free to install whatever hints they wish, including ones that break completeness of the algorithm. However, as long as hints are confluent and strongly normalizing, the algorithm behaves sensibly.

6.3 Implicit arguments

The standard library provides basic support for implicit arguments. In other systems these are usually implemented with meta-variables, which are not available in AML. In their place, we use ordinary fresh variables generated using the `assume` construct. We refer to these as *implicit variables*. We collect constraints through the operations and handlers mechanism, and resolve them using a simple first-order unification procedure.

More precisely, in the standard library we declare operations

```
operation ? : judgment
operation resolve : judgment → judgment
```

The user may place `?` anywhere where they want the term to be derived automatically, and call `resolve` c to replace the implicit variables with their derived values in the judgment computed by c.

The handler provided by the library keeps a list of implicit variables it has introduced so far, as well as their types and known solutions. The operation `?` may be triggered either in checking or inferring mode. In checking mode at type A and under binders $x_1 : B_1, \ldots, x_n : B_n$, the handler introduces a fresh implicit variable M of type $\prod_{(x_1:B_1)} \cdots \prod_{(x_n:B_n)} A$ and yields $M\, x_1 \ldots x_n$. In inferring mode the type A is not available. For simplicity, at present the library will report an error, although it might be better to create an implicit variable for the $A : \mathsf{Type}$.

During equality checking we may discover that $M\, x_1 \ldots x_n$ should be equal to a term t in which M does not occur. In this case, using `assume` again, the handler generates a term $\xi : \mathsf{Eq}(M, \lambda x_1 \ldots x_m \,.\, t)$, stores it, and also installs it as a β-hint, so that subsequent equality checks take it into account.

The operation `resolve` c is used to replace the implicit variables with their inferred values in the judgment computed by c. Such replacement does not happen automatically because the library cannot guess when is the best moment for doing so. It may be necessary to evaluate several computations before all the implicit variables become known, so we let the user control when resolution should happen.

While we feel quite encouraged by our implementation of equality checking, the implicit arguments feel a bit heavy-handed, and are quite slow. They are a satisfactory proof of concept and a demonstration of the flexibility of operations and handlers, but we need to improve it quite a bit before it becomes useful.

7 Examples

In this section we show Andromeda at work through several examples.

7.1 Proving equality with handlers

As explained in §4.3, when AML is faced with proving a non-trivial equality, it delegates it to user code by triggering the operation `equal`. To see how this works, let us walk through a computation that constructs a term witnessing symmetry of equality (without the standard library installed):

```
λ (A : Type) (x y : A) (p : x ≡ y),
  (handle
    refl x : y ≡ x
   with
   | equal x y ⇒ yield (Some p)
   end)
```

The λ-abstraction introduces a type `A`, elements `x`, `y` of type `A`, and a witness `p` of equality between `x` and `y`. Next, the type ascription is evaluated, with the enveloping handler installed. First `y ≡ x` is evaluated to the equality type $\mathsf{Eq}_{\mathtt{A}}(\mathtt{y},\mathtt{x})$ and then `refl x` is evaluated in checking mode at this type. This triggers a sub-computation of `x` and verification that `x` equals `y` (and a trivial equality check that `x` equals to itself). At this point AML triggers the operation `equal x y`, asking for evidence of equality. The enveloping handler intercepts the operation and yields the evidence `p`.

The result of the computation is displayed by Andromeda without typing annotations and assumption sets as

```
⊢ λ (A : Type) (x : A) (y : A) (_ : x ≡ y), refl x
  : Π (A : Type) (x : A) (y : A), x ≡ y → y ≡ x
```

7.2 Dependent sums

Our second example shows how to axiomatize dependent sums. This time we use the standard library and rely on its equality checking. We start by postulating the type and term constructors:

```
constant Σ : Π (A : Type) (B : A → Type), Type
constant existT : Π (A : Type) (B : A → Type) (a : A), B a → Σ A B
```

Next, we postulate the projections and tell the standard library that that the third argument of a projection should be evaluated eagerly, so that we get a working extensionality rule later on:

```
constant π1 : Π (A : Type) (B : A → Type), Σ A B → A
now reducing = add_reducing π1 [lazy, lazy, eager]

constant π2 : Π (A : Type) (B : A → Type) (p : Σ A B), B (π1 A B p)
now reducing = add_reducing π2 [lazy, lazy, eager]
```

It remains to postulate equalities, and install them as hints. The β-rules are straightforward, except that we must install the β-rule for the first projection before we postulate the second projection, or else Andromeda does not know why the second projection is well typed:

```
constant π1_β :
  Π (A : Type) (B : A → Type) (a : A) (b : B a),
    (π1 A B (existT A B a b) ≡ a)

now betas = add_beta π1_β

constant π2_β :
  Π (A : Type) (B : A → Type) (a : A) (b : B a),
    (π2 A B (existT A B a b) ≡ b)

now betas = add_beta π2_β
```

Similarly, to convince Andromeda that the extensionality rule is well typed, we need to install a *local* hint, as follows (the function `symmetry` is part of the standard library and it computes the symmetric version of an equality):

```
constant Σ_η :
  Π (A : Type) (B : A → Type) (p q : Σ A B)
    (ξ : π1 A B p ≡ π1 A B q),
    now hints = add_hint (symmetry ξ) in
      π2 A B p ≡ π2 A B q → p ≡ q

now etas = add_eta Σ_η
```

7.3 Natural numbers

The standard library provides functions for calculating weak head-normal forms that can be used as a computation device at the level of type-theoretic terms. We show how this is done by axiomatizing natural numbers and computing with them.

We postulate the type of natural numbers and its constructors

```
constant nat : Type
constant O : nat
constant S : nat → nat
```

and the induction principle

```
constant nat_rect : Π (P : nat → Type),
  P O → (Π (n : nat), P n → P (S n)) → Π (m : nat), P m
```

The weak head-normal form of the eliminator should have the fourth argument normalized:

```
now reducing = add_reducing nat_rect [lazy, lazy, lazy, eager]
```

To get computation going, we need the computation rules for the eliminator:

```
constant nat_β_O :
  Π (P : nat → Type) (x : P O) (f : Π (n : nat), P n → P (S n)),
    nat_rect P x f O ≡ x

constant nat_β_S :
  Π (P : nat → Type) (x : P O) (f : Π (n : nat), P n → P (S n))
    (m : nat),
    nat_rect P x f (S m) ≡ f m (nat_rect P x f m)
```

which we install as β-hints:

```
now betas = add_betas [nat_β_O, nat_β_S]
```

At this point, we can compute with the recursor, but there is a better way. In Andromeda there is no built-in notion of "definition" at the level of type theory (one can always use ML-level `let`-bindings, but those are always evaluated which has the undesirable effect of complete unfolding of all definitions). Instead, we break down a definition into a declaration of the constant and its defining equality. If we install the defining equality as a β-hint, a definition behaves like it would in other proof assistants, but that is just one possibility.

For example, we may define addition as follows:

```
constant ( + ) : nat → nat → nat
constant plus_def :
  Π (n m : nat), n + m ≡ nat_rect (λ _, nat) n (λ _ x, S x) m
```

Note that `plus_def` could be written as

```
constant plus_def' :
  ( + ) ≡ (λ (n m : nat), nat_rect (λ _, nat) n (λ _ x, S x) m)
```

The difference between the two is visible when we use them as β-hints: `plus_def` will unfold only after it has been applied to two arguments, whereas `plus_def'` will do so immediately.

We can derive Peano axioms by using `plus_def` as a local β-hint:

```
let plus_O =
  now betas = add_beta plus_def in
    (λ n, refl n) : Π (n : nat), n + O ≡ n

let plus_S =
  now betas = add_beta plus_def in
    (λ n m, refl (n + (S m))) : Π (n m : nat), n + (S m) ≡ S (n + m)
```

We are free to use the Peano axioms for computation rather than `plus_def`, so we install them globally as β-hints:

```
now betas = add_betas [plus_O, plus_S]
```

It should be clear from this that Andromeda is quite flexible, which is good for experimentation and tight control of how things are done, but is also bad because the user has to be more specific in what they want. The overall usability of the system depends on having a good standard library with sensible default settings.

The definition of multiplication and its Peano axioms are derived similarly:

```
constant ( * ) : nat → nat → nat
constant mult_def :
  Π (n m : nat), n * m ≡ nat_rect (λ _, nat) O (λ _ x, x + n) m

let mult_O =
  now betas = add_beta mult_def in
    (λ n, refl O) : Π (n : nat), n * O ≡ O

let mult_S =
  now betas = add_beta mult_def in
    (λ n m, refl (n * (S m))) : Π (n m : nat), n * (S m) ≡ n * m + n

now betas = add_betas [mult_O, mult_S]
```

To compute with numbers, we use the standard library function `whnf` that computes evidence that the given term is equal to its weak head-normal form:

```
do now reducing = add_reducing S [eager] in
   now reducing = add_reducing ( * ) [eager, eager] in
   now reducing = add_reducing ( + ) [eager, eager] in
     whnf ((S (S (S O))) * (S (S (S (S O)))))
```

The `do` command is the top-level command for evaluating a computation. Notice that we locally set the arguments of the successor constructor, addition, and multiplication to be computed eagerly. The effect of this is that the weak head-normal form is not weak or head-normal anymore, but rather a strongly normalizing call-by-value strategy. Thus Andromeda outputs

```
⊢ refl (S (S (S (S (S (S (S (S (S (S (S (S O))))))))))))
  : S (S (S O)) * S (S (S O)) ≡
    S (S (S (S (S (S (S (S (S (S (S (S O)))))))))))
```

It would be easy to obtain just the result, which is the left-hand side of the equality type. Notice that the proof of equality between 3×4 and 12 is a reflexivity term, even though the normalization procedure generated the proof by stringing together a large number of reduction steps. In order to keep equality proofs small, the standard library aggressively replaces equality proofs with reflection terms, using the fact that whenever $p : \mathsf{Eq}_A(s,t)$ then also $\mathsf{refl}_A\ t : \mathsf{Eq}_A(s,t)$.

7.4 Untyped λ-calculus

An example that cannot be done easily in proof assistants based on intensional type theory is in order. Let us axiomatize the untyped λ-calculus as a type that is judgmentally equal to its own function space, and show that it possesses a fixed-point operator.

We first postulate that there is a type equal to its function space:

```
constant D : Type
constant D_reflexive : D ≡ (D → D)
```

We must *not* install `D_reflexive` as a β-hint because it would lead to non-termination. Instead, we install it and its symmetric version as general hints:

```
now hints = add_hints [D_reflexive, symmetry D_reflexive]
```

With these, whenever AML needs to know that `D` and `D → D` are equal, the standard library will provide `D_reflexive`, or its symmetric version, as evidence.

Now, we can simply define the fixed-point operator:

```
let fix =
  (λ f,
    let y = (λ x : D, f ((x : D → D) x)) in
    y y)
  : (D → D) → D
```

The self-application of `x` is well-typed because Andromeda knows that `x` of type `D` also has type `D → D`, thanks to the hints. We did have to explicitly coerce `x` to the function type. (An alternative would be to use the coercion mechanism, which is demonstrated in §7.5.) Once we overcome the problem of typing the fixed-point operator, the usual mechanisms suffice to show that it does in fact compute fixed points:

```
let fix_eq =
  (λ f, refl (fix f)) : Π (f : D → D), fix f ≡ f (fix f)
```

It is a bit trickier to give a type to a term without weak head-normal form, such as $(\lambda x.\ x\,x)(\lambda x.\ x\,x)$. We must block β-reduction of this particular β-redex, without blocking all of them. To achieve this, we first introduce an alias `D'` for the type `D`:

```
constant D' : Type
constant eq_D_D' : D ≡ D'
```

Next, we define the auxiliary term δ and give it the type `D → D`:

```
let δ = (λ x : D, (x : D → D) x)
```

We now form the self-application $\delta\ \delta$ at type `D'`:

```
let Ω =
  now hints = add_hints [eq_D_D', symmetry eq_D_D'] in
  (δ : D' → D') (δ : D) : D
```

We have the desired term in which β-reduction is blocked because the inner λ-abstractions are typed at `D` and the outer application at `D'`. From here, Andromeda happily computes with Ω without ever attempting to reduce it (installing `eq_D_D'` as a global hint would be a mistake).

The preceding example should be taken as a proof of concept only. We have reached the limits of our small standard library. A more serious development of the untyped λ-calculus would use a custom equality-checking algorithm instead of manually juggling hints and type ascriptions.

7.5 Universes

The final example shows how to use coercions and operations to implement a universe à la Tarski. We postulate a universe `U`, whose elements should be thought of as *names* of types, with an operation `El` that converts the names to the corresponding types:

```
constant U : Type
constant El : U → Type
```

Because `El` is an eliminator, its normal form should have the argument in normal form, so we tell the library to normalize it eagerly:

```
now reducing = add_reducing El [eager]
```

Next, we postulate that the universe contains names for products and equality types, and install the relevant equations as β-hints:

```
constant pi : Π (a : U), (El a → U) → U
constant El_pi :
  Π (a : U) (b : El a → U), El (pi a b) ≡ (Π (x : El a), El (b x))
now betas = add_beta El_pi

constant eq : Π (a : U), El a → El a → U
constant El_eq :
  Π (a : U) (x y : El a), El (eq a x y) ≡ (x ≡ y)
now betas = add_beta El_eq
```

For testing purposes we put the name `b` of a basic type `B` into the universe:

```
constant B : Type
constant b : U
constant El_b : El b ≡ B
now betas = add_beta El_b
```

In principle we can work with `U` and `El`, but explicit uses of `El` gets tedious quickly. Ideally we want Andromeda to translate between names and their types automatically, which is achieved with a handler that intercepts coercion requests. It is easy to coerce names to types with `El`, for instance:

```
handle
  (λ x : b, x) : pi b (λ _, b)
with
 | coerce (⊢ ?t : U) (⊢ Type) ⇒ yield (Coercible (El t))
end
```

In the λ-abstraction AML found the name `b` but expected a type, therefore it triggered a coercion operation. The handler intercepted it and yielded `El b`. The process was repeated when AML found `pi b (λ _, b)` instead of a type. The final result printed by Andromeda is

```
⊢ λ (x : El b), x : El (pi b (λ (_ : El b), b))
```

We have to work harder to perform the reverse coercion, when a type is encountered where its code was expected. One first has to implement an AML function `name_of` that takes a type and returns its name, if it can find one. We do not show its implementation here, and ask the interested readers to consult the examples that come with the source code. Using `name_of` we can handle translation between types and names in both directions with the handler

```
let universe_handler =
handler
  | coerce (⊢ ?a : U) (⊢ Type) ⇒ yield (Coercible (El a))
  | coerce (⊢ ?T : Type) (⊢ U) ⇒
    match name_of T with
    | None ⇒ yield NotCoercible
    | Some ?name ⇒ yield (Coercible name)
    end
  end
```

We added a clause that intercepts coercions from `Type` to `U` and uses `name_of`. The handler automatically translates names to types and vice versa. For instance, the computation

```
with universe_handler handle
  (Π (x : b), x ≡ x) : U
```

evaluates to

```
⊢ pi b (λ (y : El b), eq b y y) : U
```

In one direction the handler coerced the name `b` to the type `B`, and in the other the type `Π (x : B), x ≡ x` to its name, as shown above.

8 Related work

Andromeda draws heavily on the experience and ideas from other proof assistants. It is difficult to do justice to all of them. Its overall design follows the tradition of LCF [12] and its descendants [15, 16, 13]. However, LCF and many of its descendants use Church's *simple* type theory [6], whereas Andromeda is based on the *dependent* type theory of Martin-Löf [17]. Consequently, Andromeda cannot advantageously integrate the ML-level and the object-level types. There is by necessity a sharp distinction between the statically typed ML-level and the dynamically evaluated type-theoretic judgments.

It makes sense to compare Andromeda to other proof assistants based on dependent type theory [19, 7, 10]. For instance, the evaluation strategy for judgments in Andromeda is based on bidirectional type-checking found in dependently-typed assistants. Andromeda is primarily a special-purpose programming language, whereas Coq, Agda, and Lean are tools for interactive proof development. The difference in philosophy of design is visible in the level of control given to the user. Andromeda gives the user full control of the system, and expects them to implement their own proof development tools, whereas Coq and Agda provide more of an end-user environment with a rich selection of ready-made tools. It is interesting to note that recently Coq and Agda have both started giving the user more control. New versions of Coq allow the use of tactics inside type-theoretic terms [26, §2.11.2] and allow fine-tuning of Coq's unification algorithm [27]. Agda even lets the user install new normalization rules [1] that might break the system.

We already mentioned that NuPRL [2] validates equality reflection by interpreting types as partial equivalence relations on terms of a computational model, namely an extension of the untyped λ-calculus. We do not wish to make such a commitment in Andromeda, and instead allow interpretations that are inconsistent with computational type theory.

9 Future work

We feel that Andromeda shows a promising way to design a proof assistant based on type theory with equality reflection, but much remains to be done.

Syntactic sugar and end-user support

AML turned out to be a useful tool for the implementers of the standard library. If we imagine that the end-user is a mathematician who just wants to do mathematics, without learning the intricacies of operations and handlers, then we need further support for creating a more user friendly environment. There ought to be ways of introducing new syntactic constructs, and reasonable error reporting by the standard library. We are not quite sure how to provide such functionality. The approach taken by Bowman [4] seems interesting. Another possibility is to allow user-defined notations in the style of Coq, or to completely separate the end-user interface and AML.

Formal verification of the meta-theoretic properties

While we carefully designed the underlying type theory and made sure it is precisely clear what the type-theoretic rules are, we have not formally verified that the system has the desired meta-theoretic properties, such as uniqueness of typing, validity of inversion principles, and well-behaved context joins. We expect no trouble here, but do insist on formal verification. In the early stages of implementation we managed to delude ourselves more than once about the properties of the underlying type theory.

Recording derivations

Like all LCF-style proof assistants, Andromeda does not record derivations, only their conclusions. (In fact, all practical proof assistants do this, though some implement type theories that allow derivations to be reconstructed.) There might be situations in which we wish to record or communicate the derivation. For instance, we might want to send the derivation to another proof assistant for independent verification. This can be accomplished with a minor modification of AML: if we implement *all* calls of AML to the nucleus as operations (whose default handler is the nucleus), then the user can intercept them and do whatever they like: record them, communicate them, or modify them to obtain a proof translation. Alternatively, the `judgment` type in the nucleus could be made into the type of derivations with no breaking changes to the interface.

Removal of Type : Type

A major forthcoming modification of the current system is elimination of Type : Type. The syntax of AML takes advantage of Type : Type to conflate term and type judgments within a single abstract type `judgment`.

But the nucleus does not rely on Type : Type at all and separates the various judgment forms into separate abstract datatypes. There is no technical difficulty in removing Type : Type, but the question is what to replace it with. One possibility is to add a basic type U and a basic type family El indexed by U, and then use these as a Tarski-style universe, with a standard library employing techniques of §7.5 to make the system usable. (This can be extended to multiple universes if desired.) Paolo Capriotti's recently suggested such a setup [5], based on semantic considerations in categories of presheaves. Finally, AML could be modified to support several abstract datatypes, one for each form of object-level judgment.

Once this is done, several interesting possibilities arise. The user could hypothesize any universe structure they like, including putting back Type : Type. We might even be able to remove equality reflection from the nucleus, and make it user-definable. We hope to report on these exciting developments in the near future.

References

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

2 S.F. Allen, M. Bickford, R.L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

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

4 William J. Bowman. Growing a proof assistant. In *Higher-Order Programming with Effects*, 2016.

5 Paolo Capriotti. Notions of type formers. In *23rd International Conference on Types for Proofs and Programs (TYPES)*. Budapest, Hungary, May 29–June 1 2017.

6 Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, 1940. doi:10.2307/2266170.

7 Coq Development Team. The Coq proof assistant. Available at http://coq.inria.fr/, 2016.

8 Thierry Coquand. An Algorithm for Testing Conversion in Type Theory. In Gérard Huet and G. Plotkin, editors, *Logical frameworks*, pages 255–277. Cambridge University Press, 1991.

9 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

10 Leonardo de Moura, Soonho Kong, Floris van Doorn, Jakob von Raumer, and Jeremy Avigad. The Lean theorem prover (system description). In *25th International Conference on Automated Deduction (CADE-25)*, 2015.

11 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

12 Michael J. Gordon, Arthur J. Milnter, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer-Verlag, 1979.

13 John Harrison. The HOL Light theorem prover. Available at https://www.cl.cam.ac.uk/~jrh13/hol-light/.

14 Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997.

15 HOL Interactive Theorem Prover. Available at https://hol-theorem-prover.org.

16 Isabelle proof assistant. Available at https://isabelle.in.tum.de.

17 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Studies in Proof Theory. Bibliopolis, 1984.

18 Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

19 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

20 OCaml programming language. Available at https://ocaml.org.

21 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 80–94, 2009.

22 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.

23 Matija Pretnar. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science*, 319:19–35, 2015. doi:10.1016/j.entcs.2015.12.003.

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

25 Thomas Streicher. Investigations into intensional type theory. Habilitation Thesis, Ludwig-Maximilians Universität, 1993.

26 Coq Development Team. The Coq proof assistant reference manual, version 8.5. Available at https://coq.inria.fr/distrib/8.5/refman/.

27 Beta Ziliani and Matthieu Sozeau. A Unification Algorithm for Coq featuring Universe Polymorphism and Overloading. In *ACM SIGPLAN International Conference on Functional Programming 2015*, 2015.

A The rules of type theory

In this appendix we give the formulation of type theory in a declarative way that minimizes the number of judgments, and so is better suited for a semantic account. We omit formal treatment of bound variables and substitution, which is standard.

A.1 Syntax

Contexts

Γ, Δ	$::=$	$\bullet$	empty context
	$\mid$	$\Gamma,\, x : A$	context Γ extended with $x : A$

Terms and types

s, t, A, B	$::=$	Type	universe
	$\mid$	$\prod_{(x:A)} B$	product
	$\mid$	$\mathsf{Eq}_A(s, t)$	equality type
	$\mid$	x	variable
	$\mid$	$\lambda x{:}A.B\,.\,t$	λ-abstraction
	$\mid$	$s \mathbin{@}^{x:A.B} t$	application
	$\mid$	$\mathsf{refl}_A\ t$	reflexivity

A.2 Judgments

$\Gamma\ \mathsf{ctx}$	Γ is a well formed context
$\Gamma \vdash t : A$	t is a well formed term of type A in context Γ
$\Gamma \vdash s \ \equiv\ t : A$	s and t are equal terms of type A in context Γ

We use the following abbreviations:

$\Gamma \vdash A\ \mathsf{type}$	abbreviates $\Gamma \vdash A : \mathsf{Type}$
$\Gamma \vdash A \equiv B$	abbreviates $\Gamma \vdash A \ \equiv\ B : \mathsf{Type}$

A.3 Contexts

CTX-EMPTY

$$\frac{}{\bullet\ \mathsf{ctx}}$$

CTX-EXTEND

$$\frac{\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash A\ \mathsf{type} \qquad x \notin \mathrm{dom}(\Gamma)}{(\Gamma,\, x : A)\ \mathsf{ctx}}$$

A.4 Terms and types

Conversion

TERM-TY-CONV
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash t : B}$$

Variable

TERM-VAR
$$\frac{(\Gamma,\, x : A)\ \mathsf{ctx}}{\Gamma,\, x : A \vdash x : A}$$

TERM-VAR-SKIP
$$\frac{(\Gamma,\, y : B)\ \mathsf{ctx} \qquad \Gamma \vdash x : A}{\Gamma,\, y : B \vdash x : A}$$

Universe

TY-TYPE
$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}\ \mathsf{type}}$$

Product

TY-PROD
$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma,\, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \prod_{(x:A)} B\ \mathsf{type}}$$

TERM-ABS
$$\frac{\Gamma,\, x : A \vdash t : B}{\Gamma \vdash (\lambda x{:}A.B\,.\,t) : \prod_{(x:A)} B}$$

TERM-APP
$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ @^{x:A.B}\ t : B[t/x]}$$

Equality type

TY-EQ
$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash s : A \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{Eq}_A(s,t)\ \mathsf{type}}$$

TERM-REFL
$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}_A\ t : \mathsf{Eq}_A(t,t)}$$

A.5 Equality

General rules

EQ-REFL
$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \ \equiv\ t : A}$$

EQ-SYM
$$\frac{\Gamma \vdash t \ \equiv\ s : A}{\Gamma \vdash s \ \equiv\ t : A}$$

EQ-TRANS
$$\frac{\Gamma \vdash s \ \equiv\ t : A \qquad \Gamma \vdash t \ \equiv\ u : A}{\Gamma \vdash s \ \equiv\ u : A}$$

Conversion

EQ-TY-CONV
$$\frac{\Gamma \vdash s \ \equiv\ t : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash s \ \equiv\ t : B}$$

Equality reflection

EQ-REFLECTION
$$\frac{\Gamma \vdash u : \mathsf{Eq}_A(s,t)}{\Gamma \vdash s \ \equiv\ t : A}$$

Computation

PROD-BETA

$$\frac{\Gamma,\, x : A \vdash s : B \qquad \Gamma \vdash t : A}{\Gamma \vdash (\lambda x{:}A.B\,.\,s)\;@^{x:A.B}\;t \;\equiv\; s[t/x] : B[t/x]}$$

Extensionality

EQ-ETA

$$\frac{\Gamma \vdash t : \mathsf{Eq}_A(s,u) \qquad \Gamma \vdash v : \mathsf{Eq}_A(s,u)}{\Gamma \vdash t \;\equiv\; v : \mathsf{Eq}_A(s,u)}$$

PROD-ETA

$$\frac{\Gamma \vdash s : \prod_{(x:A)} B \qquad \Gamma \vdash t : \prod_{(x:A)} B \qquad \Gamma,\, x : A \vdash (s\;@^{x:A.B}\;x) \;\equiv\; (t\;@^{x:A.B}\;x) : B}{\Gamma \vdash s \;\equiv\; t : \prod_{(x:A)} B}$$

A.5.1 Congruences

Type formers

CONG-PROD

$$\frac{\Gamma \vdash A \equiv C \qquad \Gamma,\, x : A \vdash B \equiv D[x/y]}{\Gamma \vdash \prod_{(x:A)} B \equiv \prod_{(y:C)} D}$$

CONG-EQ

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash s \;\equiv\; u : A \qquad \Gamma \vdash t \;\equiv\; v : A}{\Gamma \vdash \mathsf{Eq}_A(s,t) \equiv \mathsf{Eq}_B(u,v)}$$

Products

CONG-ABS

$$\frac{\Gamma \vdash A \equiv C \qquad \Gamma,\, x : A \vdash B \equiv D[x/y] \qquad \Gamma,\, x : A \vdash s \;\equiv\; t[x/y] : B}{\Gamma \vdash (\lambda x{:}A.B\,.\,s) \;\equiv\; (\lambda y{:}C.D\,.\,t) : \prod_{(x:A)} B}$$

CONG-APP

$$\frac{\Gamma \vdash A \equiv C \qquad \Gamma,\, x : A \vdash B \equiv D[x/y] \qquad \Gamma \vdash s \;\equiv\; u : \prod_{(x:A)} B \qquad \Gamma \vdash t \;\equiv\; v : A}{\Gamma \vdash (s\;@^{x:A.B}\;t) \;\equiv\; (u\;@^{y:C.D}\;v) : B[t/x]}$$

Equality types

CONG-REFL

$$\frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash s \;\equiv\; t : A}{\Gamma \vdash \mathsf{refl}_A\; s \;\equiv\; \mathsf{refl}_B\; t : \mathsf{Eq}_A(s,s)}$$

B The auto tactic

We include here the complete code for implementing a simple `auto` tactic from § 2.

We first define the `map` function to show how AML syntax works, and the auxiliary `apply` function that folds application of a function over a list of arguments:

```
let rec map f xs =
  match xs with
  | [] ⇒ []
  | ?x :: ?xs ⇒ (f x) :: (map f xs)
  end

let rec apply f xs =
  match xs with
  | [] ⇒ f
  | ?x :: ?xs ⇒ apply (f x) xs
  end
```

Next, we declare the `failure` operation that is triggered when the search fails:

```
operation failure : judgment
```

Next we define the function `subgoals` that takes a goal `A` and a hypothesis `B` and computes a list of subgoals that together with `B` imply `A`. For instance, if `B` is equal to `C → D → A` then the computed subgoals are `[C, D]`:

```
let rec subgoals A B =
  match B with
  | ⊢ A ⇒ []
  | ⊢ ?P → ?Q ⇒ P :: (subgoals A Q)
  | _ ⇒ [failure]
  end
```

The function `derive` takes a goal `A` and attempts to derive it. If the goal is an implication, it introduces the antecedent as a hypothesis and calls itself recursively. Otherwise it tries to prove the goal from the current hypotheses by simple backchaining:

```
let rec derive A =
  match A with
  | ⊢ ?P → ?Q ⇒ λ (x : P), derive Q
  | ⊢ _ ⇒ backchain A hypotheses
  end

and backchain A lst =
  match lst with
  | [] ⇒ failure
  | (⊢ ?f : ?B) :: ?lst ⇒
    handle
      apply f (map derive (subgoals A B))
    with
      failure ⇒ backchain A lst
    end
  end
```

Note how `backchain` uses a handler to intercept `failure`, just like an ordinary exception handler does. Finally, we declare an operation `auto` and define a global handler that handles it. The handler only works when `auto` is used in checking mode:

```
operation auto : judgment

handle
| auto : ?T' ⇒
    match T' with
    | Some ?T ⇒ derive T
    | None ⇒ failure
    end
end
```

Now we can use `auto` to inhabit types. For example,

```
(λ (X : Type), auto : X → X)
```

computes to

```
⊢ λ (X : Type) (x : X), x : Π (X : Type), X → X
```

and given the types

```
constant A : Type
constant B : Type
constant C : Type
```

the computation

```
auto : (A → B → C) → (A → B) → (A → C)
```

results in

```
⊢ λ (x : A → B → C) (x0 : A → B) (x1 : A), x x1 (x0 x1)
  : (A → B → C) → (A → B) → A → C
```