The Andromeda proof assistant

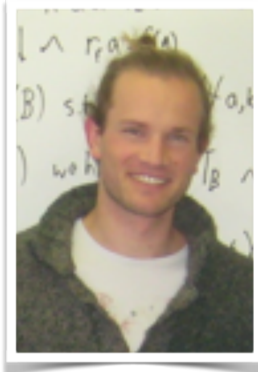
Andrej Bauer
University of Ljubljana

Workshop on Categorical Logic and Univalent Foundations
University of Leeds, July 2016

Thank you for the invitation. Today I would like to talk about a tool we have been developing. Nowadays we call it Andromeda.

Team

Gaëtan
Gilbert



Philipp
Haselwarter

Matija
Pretnar

Chris
Stone

Our team consists of several people: Gaëtan Gilbert from ENS Lyon, Chris Stone from Harvey Mudd College, USA, and Matija Pretnar, Philipp Haselwarter and myself from University of Ljubljana. If you'd like to participate you can find us on GitHub.

1. Goals
2. Architecture
3. Examples

The talk is divided into three parts.
First I'd like to explain what our goals are.
Second, I'd like to go into a bit of detail on how Andromeda works.
Third, I will show some examples of what Andromeda can do.

1. **Goals**
2. Architecture
3. Examples

So what are we trying to do here?

Voevodsky's Homotopy Type System

Originally Andromeda (then called Brazil) was implementation of (a variant of) Voevodsky's system HTS: a type theory with two kinds of equality, a strict and the non-strict one coexisting.

But then we started stripping features from it because we saw they were user-definable. Eventually nothing was left of the "H" part and we were left with a bare-bones type theory. Nonetheless, it allows us to express a great variety of type-theoretic constructs.

But Andromeda should not be thought of as a type theory. It is primarily a *programming language* for formalization of mathematics in type theory, and especially for *experimenting* with formalization techniques. It is *not* trying to be an interactive user-friendly system such as Coq, Agda or Lean (although of course we would be delighted to eventually have an interactive layer on top of it, and in principle nothing prevents us from having one).

Type theory with $\prod$ and equality reflection

Originally Andromeda (then called Brazil) was implementation of (a variant of) Voevodsky's system HTS: a type theory with two kinds of equality, a strict and the non-strict one coexisting.

But then we started stripping features from it because we saw they were user-definable. Eventually nothing was left of the "H" part and we were left with a bare-bones type theory. Nonetheless, it allows us to express a great variety of type-theoretic constructs.

But Andromeda should not be thought of as a type theory. It is primarily a *programming language* for formalization of mathematics in type theory, and especially for *experimenting* with formalization techniques. It is *not* trying to be an interactive user-friendly system such as Coq, Agda or Lean (although of course we would be delighted to eventually have an interactive layer on top of it, and in principle nothing prevents us from having one).

Type theory with $\prod$ and equality reflection

Flexible programming language for formalization

Originally Andromeda (then called Brazil) was implementation of (a variant of) Voevodsky's system HTS: a type theory with two kinds of equality, a strict and the non-strict one coexisting.

But then we started stripping features from it because we saw they were user-definable. Eventually nothing was left of the "H" part and we were left with a bare-bones type theory. Nonetheless, it allows us to express a great variety of type-theoretic constructs.

But Andromeda should not be thought of as a type theory. It is primarily a *programming language* for formalization of mathematics in type theory, and especially for *experimenting* with formalization techniques. It is *not* trying to be an interactive user-friendly system such as Coq, Agda or Lean (although of course we would be delighted to eventually have an interactive layer on top of it, and in principle nothing prevents us from having one).

Not built in

- equality checking
- normal forms
- no notion of computation
- implicit arguments, coercions, type classes
- inductive types, records, universes

Let me emphasize that we're not making yet another interactive proof assistant. All of the features listed here are commonly expected by the user of a proof assistant, but they are *not* built into Andromeda. They are however user-definable.

For instance, as long as we stay within a reasonable fragment of type theory, we can implement equality checking – and we did.

~~Not built in~~

User definable

- equality checking
- normal forms
- no notion of computation
- implicit arguments, coercions, type classes
- inductive types, records, universes

Let me emphasize that we're not making yet another interactive proof assistant. All of the features listed here are commonly expected by the user of a proof assistant, but they are *not* built into Andromeda. They are however user-definable.

For instance, as long as we stay within a reasonable fragment of type theory, we can implement equality checking – and we did.

1. Goals
2. **Architecture**
3. Examples

Let us move onto describing the design of Andromeda. There are too many details to cover in a single talk, but I hope to convey some design choices which are interesting also from a foundational point of view.

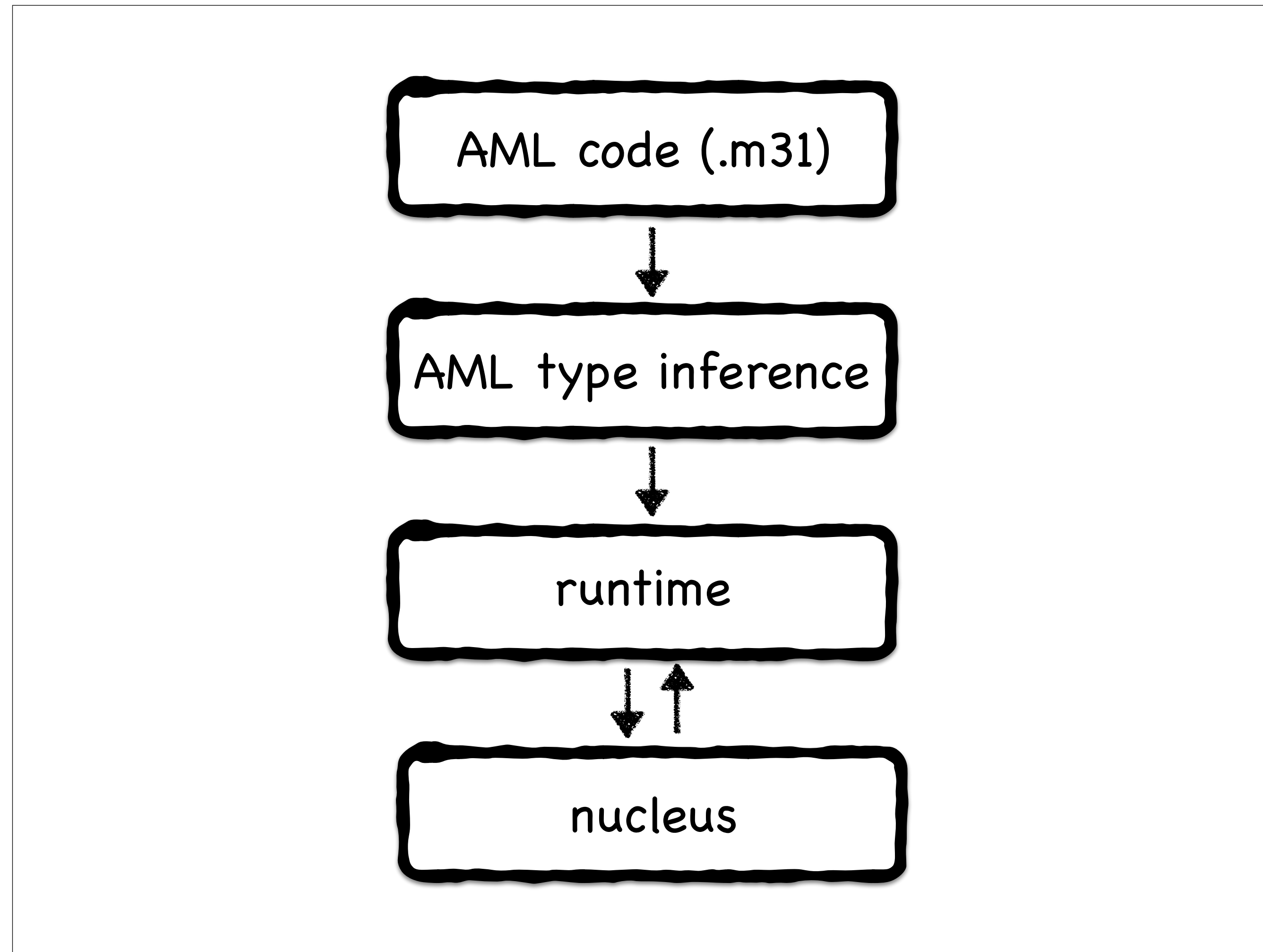
Andromeda meta-language (AML)

- general-purpose programming language
- statically typed
- abstract type `judgment`

If you know about the LCF and HOL family of theorem provers then you can think of Andromeda as an LCF-style theorem prover, but for *dependent* type theory. In contrast, HOL implements Church's simple type theory.

Theorems are proved by writing programs in the Andromeda meta-language (AML), which is an ML-like language with Ocaml-style syntax.

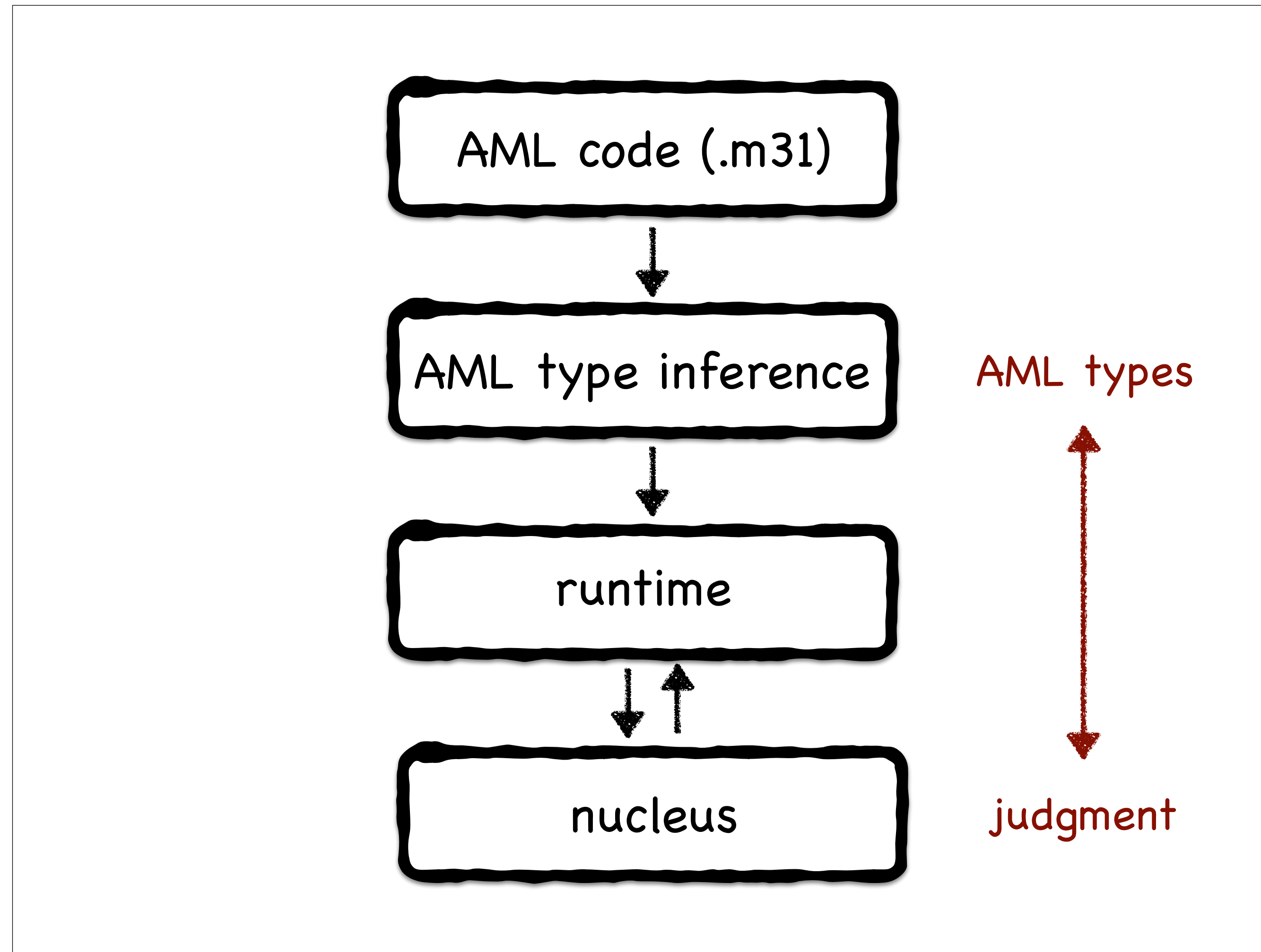
It has an abstract datatype "judgement" of type-theoretic judgements. The only way to construct values of this type is to pass through a small trusted nucleus (galaxies do not have "kernels", they have "nuclei"), which guarantees correctness. This is a familiar setup.



The following diagram shows how programs are executed.

1. The user writes some AML code.
2. The *meta-level* types are inferred. There are no dependent types at this level, just ML types.
3. An evaluator executes the program.
4. Any computations involving judgments pass through the nucleus.

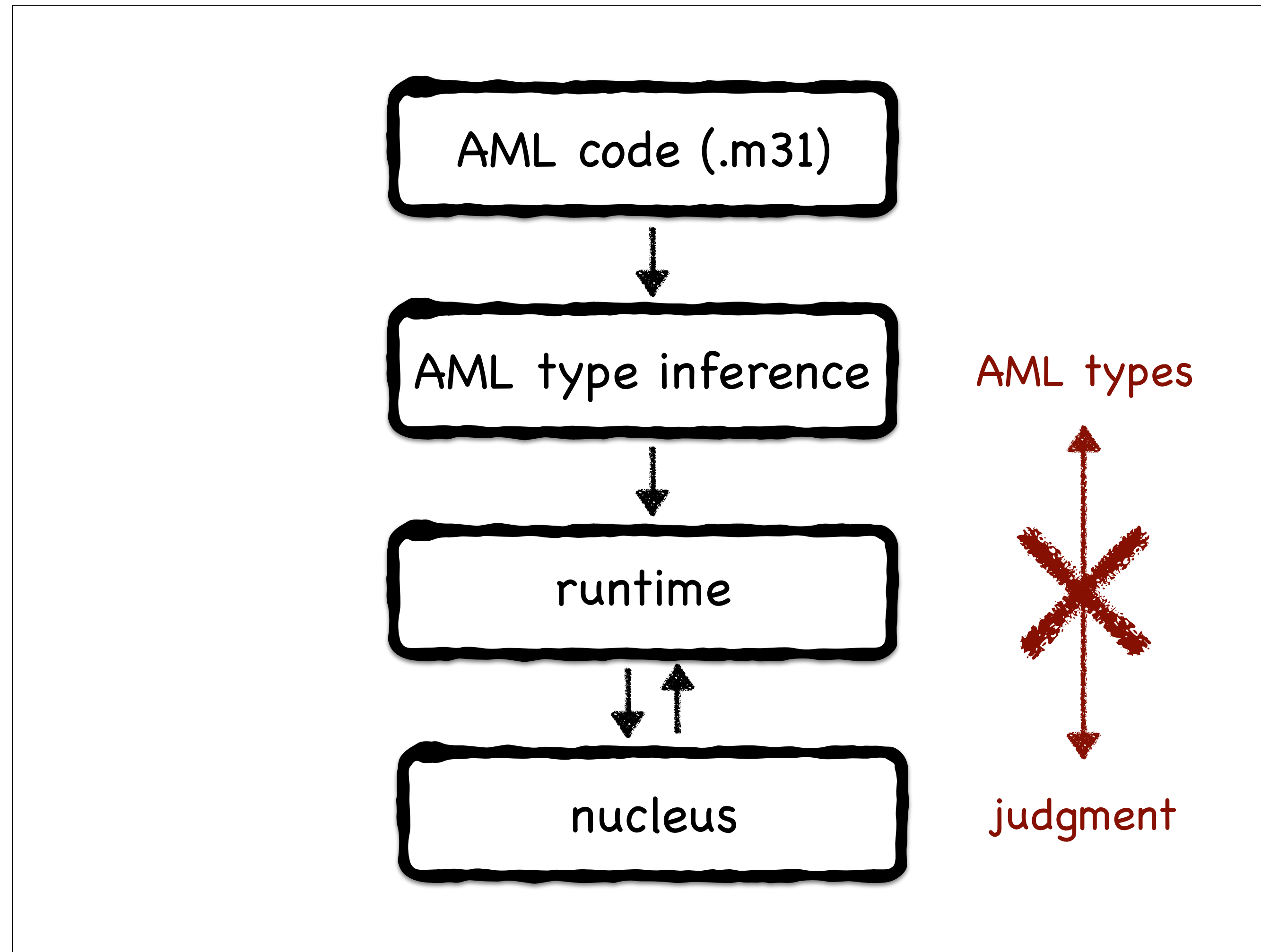
There is an important difference between HOL and Andromeda regarding the interaction between the meta-level type inference and the object-level types. In HOL the object-level types are simple and so the meta-level type inference can deduce useful information about the object-level (for example, that the identity function has the polymorphic type $\alpha \to \alpha$). No such interaction exists in Andromeda. The judgment type is completely opaque to AML. This means that during runtime there can be an object-level type error.



The following diagram shows how programs are executed.

1. The user writes some AML code.
2. The *meta-level* types are inferred. There are no dependent types at this level, just ML types.
3. An evaluator executes the program.
4. Any computations involving judgments pass through the nucleus.

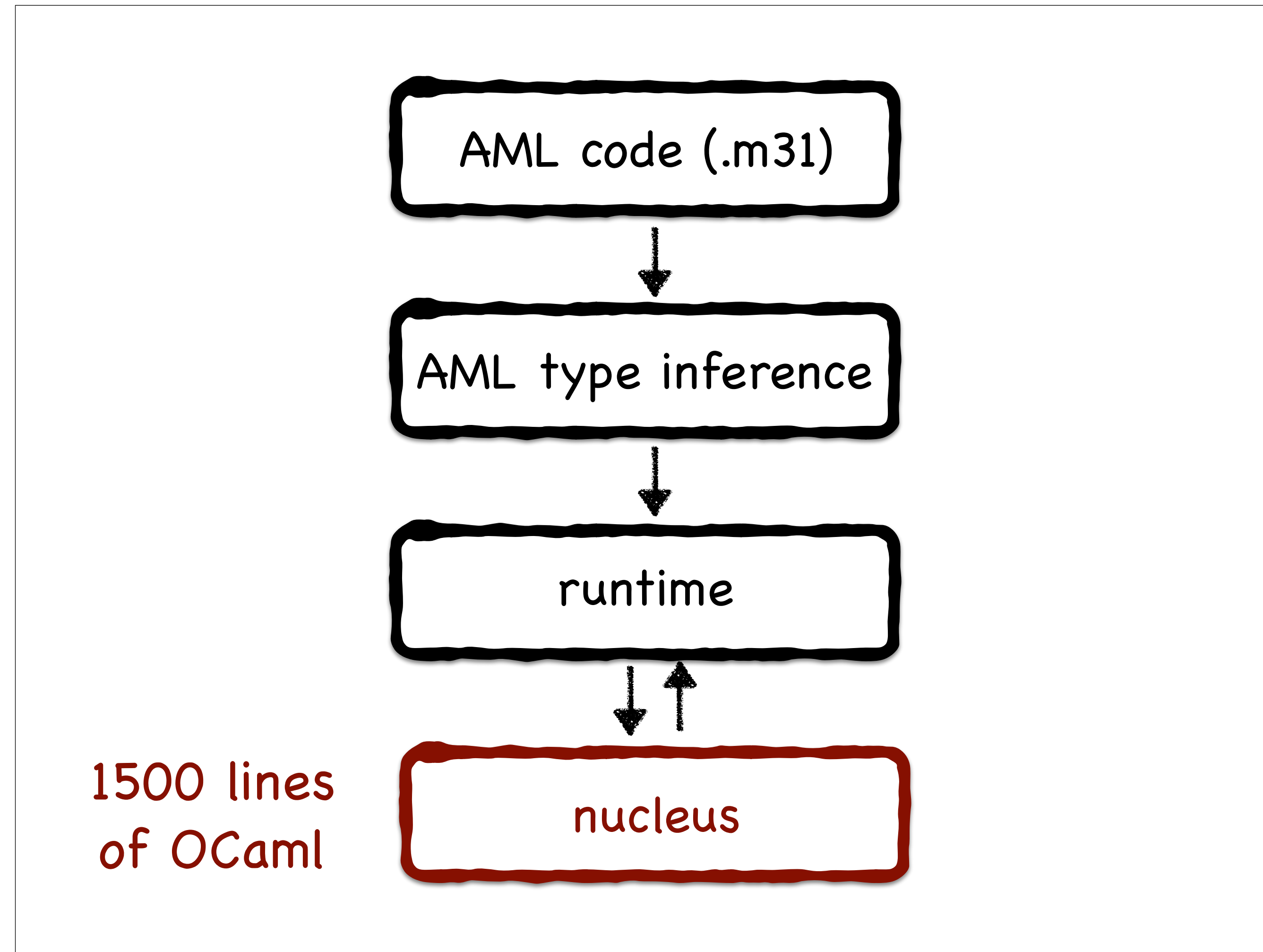
There is an important difference between HOL and Andromeda regarding the interaction between the meta-level type inference and the object-level types. In HOL the object-level types are simple and so the meta-level type inference can deduce useful information about the object-level (for example, that the identity function has the polymorphic type $\alpha \to \alpha$). No such interaction exists in Andromeda. The judgment type is completely opaque to AML. This means that during runtime there can be an object-level type error.



The following diagram shows how programs are executed.

1. The user writes some AML code.
2. The *meta-level* types are inferred. There are no dependent types at this level, just ML types.
3. An evaluator executes the program.
4. Any computations involving judgments pass through the nucleus.

There is an important difference between HOL and Andromeda regarding the interaction between the meta-level type inference and the object-level types. In HOL the object-level types are simple and so the meta-level type inference can deduce useful information about the object-level (for example, that the identity function has the polymorphic type $\alpha \to \alpha$). No such interaction exists in Andromeda. The judgment type is completely opaque to AML. This means that during runtime there can be an object-level type error.



Andromeda is implemented in OCaml in around 9000 lines of code. The nucleus, which is the trusted part, has about 1500 lines of code. We do not anticipate it growing is size much.

(In reality you also have to trust the parser, the pretty printer, Intel CPUs and cosmic rays.)

Type theory

- products `∏(x:A),B`
- equality types $\mathtt{Eq}_{A}(u,v)$
- `Type:Type`

Let us turn attention to the object-level dependent type theory. On the Andromeda web site http://andromedans.github.io/andromeda/ you can find a complete description of the type theory.

In essence, we have only dependent products and equality types.

The dependent products are standard, with the expected rules, including β-equality.
We do *not* assume an η-rule or function extensionality because those are user-definable through equality reflection.

The equality type has an introduction rule for reflexivity terms. Instead of the elimination rule we have equality reflection, see next slide. We do *not* assume any strictness (such as uniqueness of identity proofs) because that is again a user-definable feature.

We have “type in type”, a simplifying hack that allows easy experiments. We definitely want to remove it in the future.

Type theory

User-definable:

- products $\prod(x:A),B$
- equality types $Eq_A(u,v)$
- Type:Type

Let us turn attention to the object-level dependent type theory. On the Andromeda web site http://andromedans.github.io/andromeda/ you can find a complete description of the type theory.

In essence, we have only dependent products and equality types.

The dependent products are standard, with the expected rules, including β-equality.
We do *not* assume an η-rule or function extensionality because those are user-definable through equality reflection.

The equality type has an introduction rule for reflexivity terms. Instead of the elimination rule we have equality reflection, see next slide. We do *not* assume any strictness (such as uniqueness of identity proofs) because that is again a user-definable feature.

We have “type in type”, a simplifying hack that allows easy experiments. We definitely want to remove it in the future.

Type theory

User-definable:

function extensionality

- products $\prod(x:A),B$
- equality types $Eq_A(u,v)$
- Type:Type

Let us turn attention to the object-level dependent type theory. On the Andromeda web site http://andromedans.github.io/andromeda/ you can find a complete description of the type theory.

In essence, we have only dependent products and equality types.

The dependent products are standard, with the expected rules, including β-equality.
We do *not* assume an η-rule or function extensionality because those are user-definable through equality reflection.

The equality type has an introduction rule for reflexivity terms. Instead of the elimination rule we have equality reflection, see next slide. We do *not* assume any strictness (such as uniqueness of identity proofs) because that is again a user-definable feature.

We have “type in type”, a simplifying hack that allows easy experiments. We definitely want to remove it in the future.

Type theory

User-definable:

- products $\prod(x:A),B$ — function extensionality
- equality types $Eq_A(u,v)$ — UIP
- Type:Type

Let us turn attention to the object-level dependent type theory. On the Andromeda web site http://andromedans.github.io/andromeda/ you can find a complete description of the type theory.

In essence, we have only dependent products and equality types.

The dependent products are standard, with the expected rules, including β-equality.
We do *not* assume an η-rule or function extensionality because those are user-definable through equality reflection.

The equality type has an introduction rule for reflexivity terms. Instead of the elimination rule we have equality reflection, see next slide. We do *not* assume any strictness (such as uniqueness of identity proofs) because that is again a user-definable feature.

We have "type in type", a simplifying hack that allows easy experiments. We definitely want to remove it in the future.

Type theory

User-definable:

- products $\prod(x:A),B$ — function extensionality
- equality types $Eq_A(u,v)$ — UIP
- Type:Type ← temporary hack

Let us turn attention to the object-level dependent type theory. On the Andromeda web site http://andromedans.github.io/andromeda/ you can find a complete description of the type theory.

In essence, we have only dependent products and equality types.

The dependent products are standard, with the expected rules, including β-equality.
We do *not* assume an η-rule or function extensionality because those are user-definable through equality reflection.

The equality type has an introduction rule for reflexivity terms. Instead of the elimination rule we have equality reflection, see next slide. We do *not* assume any strictness (such as uniqueness of identity proofs) because that is again a user-definable feature.

We have "type in type", a simplifying hack that allows easy experiments. We definitely want to remove it in the future.

Equality reflection

$$\frac{\Gamma \vdash p : \mathrm{Eq}_A(e_1, e_2)}{\Gamma \vdash e_1 \equiv_A e_2}$$

A distinguishing feature of the underlying type theory is *equality reflection*.

It gives the type theory expressive power because it allows us to essentially hypothesize new *judgmental equalities* by postulating elements of equality types.

It also makes the type theory tricky. If we figure out how to remove equality reflection without sacrificing expressivity and convenience, we will do so, but it does not seem easy.

`x`	variable
`Type`	universe
`∏(x:A),B`	product
`λ(x:A),(e )`	abstraction
e_1 e_2	application
`Eq` (e_1, e_2)	equality type
`refl (e)`	reflexivity

The syntax of types and terms is therefore quite simple. However, the full syntax is not just what you see here. It is augmented with extra information that is necessary if we are to make sense of things in the presence of equality reflection. I do not have time to explain why the extra information is necessary, but I am happy to do so offline. You can also ask Philipp who is sitting in the audience.

x	variable
Type	universe
$\prod$(x:A),B	product
λ(x:A),(e:B)	abstraction
$e_1{}^{@(x:A,B)}e_2$	application
$Eq_A(e_1,e_2)$	equality type
$refl_A(e)$	reflexivity

First, we need to add explicit typing annotations on the codomain of an abstraction, applications, equality types and reflexivity terms.

$$
\begin{array}{c}
\texttt{A:Type,} \\
\texttt{B:Type,} \\
\xi\texttt{:(Eq}_{\texttt{Type}}\texttt{(A,B)),} \\
\texttt{f:(}\prod\texttt{(_:A),A),} \\
\texttt{x:B} \\
\vdash \\
\texttt{(f}\ ^{@(_:A,A)}\ \texttt{x) : A}
\end{array}
$$

The second modification of standard theory involves judgments. Let me explain this with an example.

Consider the following judgement, where the context is written vertically. We have two types A and B, an assumption ξ that they are equal, a function f from A to A, and x of type B. It makes sense to apply f to x because by ξ and equality reflection A and B are *judgmentally* equal. But notice that ξ is not mentioned anywhere. This breaks *strengthening*: just because a variable is not mentioned, that does not mean we can delete it.

Not having strengthening is a practical annoyance (I am not explaining why). To recover it, we annotate every subterm with the part of the context that is needed for that subterm to be well-formed, i.e., we make dependencies on assumptions explicit.

Once this information is available, another modification happens: a context is not a list anymore, but rather *acyclic directed graph* indicating dependencies between variables. In the above example, we would have edges from ξ to A and to B, from f to A, and from x to B.

$$
\begin{array}{c}
A : \mathsf{Type}, \\
B : \mathsf{Type}, \\
\xi : (\mathsf{Eq}_{\mathsf{Type}}(A^{A}, B^{B}))^{A,B}, \\
f : (\textstyle\prod(_ : A^{A}), A^{A})^{A}, \\
x : B^{B} \\
\vdash \\
(f^{f,A} \; @(_ : A^{A}, A^{A}) \; x^{x,B})^{x,B,\xi} : A^{A}
\end{array}
$$

The second modification of standard theory involves judgments. Let me explain this with an example.

Consider the following judgement, where the context is written vertically. We have two types A and B, an assumption ξ that they are equal, a function f from A to A, and x of type B. It makes sense to apply f to x because by ξ and equality reflection A and B are *judgmentally* equal. But notice that ξ is not mentioned anywhere. This breaks *strengthening*: just because a variable is not mentioned, that does not mean we can delete it.

Not having strengthening is a practical annoyance (I am not explaining why). To recover it, we annotate every subterm with the part of the context that is needed for that subterm to be well-formed, i.e., we make dependencies on assumptions explicit.

Once this information is available, another modification happens: a context is not a list anymore, but rather *acyclic directed graph* indicating dependencies between variables. In the above example, we would have edges from ξ to A and to B, from f to A, and from x to B.

$$\frac{\Gamma \vdash e_1 : \prod(x{:}A), B \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash e_1{}^{@(x:A,B)} e_2 : B[e_1/x]}$$

The third modification involves treatment of judgments, again let me show this with an example.
Consider the application rule (subterm dependencies are not shown).

$$\frac{\Gamma_1 \vdash e_1 : \prod(x{:}A),B \qquad \Gamma_2 \vdash e_2 : B}{? \vdash e_1 \ {}^{@(x:A,B)}e_2 : B[e_1/x]}$$

In practice, the user might, and does compute the two premises with *different contexts.*
What should we do with the context in the conclusion?

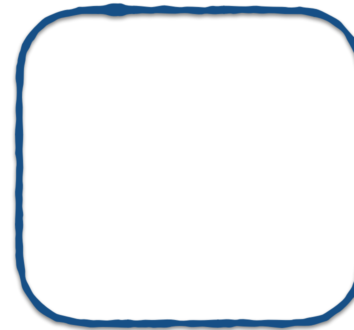
$$\frac{\Gamma_1 \vdash e_1 : \prod(x{:}A),B \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1 \bowtie \Gamma_2 \vdash e_1 {}^{@(x:A,B)} e_2 : B[e_1/x]}$$

We *join* the two contexts into a large one by forming the *join* of the contexts, which is just the union of the directed graphs.

The contexts Γ_1 and Γ_2 are incompatible if they disagree about the type of a variable, in which case the join cannot be performed. In practice this does not seem to be a problem.
This joins of contexts make type theory a bit more complicated but work really well in practice.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

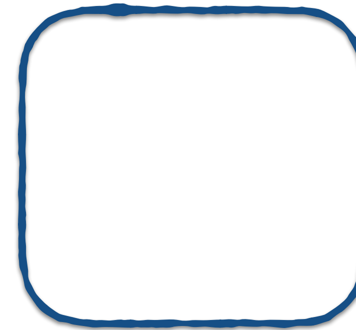


We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* "coerce x A". The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely "you do not have to coerce from A to B, because ξ shows they are equal". Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

runtime



We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* “coerce x A”. The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely “you do not have to coerce from A to B, because ξ shows they are equal”. Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

runtime

check x : A ?

We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* “coerce x A”. The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely “you do not have to coerce from A to B, because ξ shows they are equal”. Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

runtime

check x : A ?

coerce x A

We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* “coerce x A”. The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely “you do not have to coerce from A to B, because ξ shows they are equal”. Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

runtime

check x : A ?

coerce x A

We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* “coerce x A”. The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely “you do not have to coerce from A to B, because ξ shows they are equal”. Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

```
constant A, B : Type
constant ξ : B ≡ A

do handle
  λ (f:A→A) (x:B), f x
with
| coerce (⊢ _:B) (⊢ A) ⇒
    yield (Convertible ξ)
end
```

runtime

check x : A ?

coerce x A

convert x ξ

We said that the runtime performs no equality checks at all. Similarly, the nucleus just implements the rules of inference (and several admissible rules) and has no notion of computation. So how do things get done? Let us look at the following example. This is actual Andromeda code.
First we postulate existence of types A and B and an equality between them.
We then try to construct an abstraction. The application f x results in checking that x has type A.
Since the type of x is B, the runtime gives up and triggers an *operation* "coerce x A". The control is now handed back to the user who provided a *handler* for this operation. The handler yields back an answer, namely "you do not have to coerce from A to B, because ξ shows they are equal". Now the runtime uses ξ to generate the judgment x : A using conversion (done by the nucleus). From this point on, the runtime uses the nucleus to compute the abstraction.

Operations

`equal` e_1 e_2	Give me evidence of $e_1 \equiv e_2$
`coerce (e:A) B`	Give me evidence of $A \equiv B$ or a B
`coerce_fun e`	Give me a function
`as_prod A`	Give me evidence of $A \equiv \prod(x:B),C$
`as_eq A`	$A \equiv (e_1 \equiv e_2)$

The runtime consults the user through several other operations, shown here.
The user can then use the handler mechanism to provide the evidence that the runtime asked for.
The evidence is again computed by the runtime, so we get an entangled interaction between the user code and the runtime.
The nucleus is not involved in the interaction. The runtime first gathers all the evidence it needs and then passes it to the nucleus to produce a judgment.

1. Goals
2. Architecture
3. **Examples**

Let us look at some examples.

Standard library

- 1200 lines of AML code, safe by desin
- Type-directed equality checking (Harper & Stone)
- User-extensible extensionality rules
- User extensible with β-rules
- Implicit arguments and simple unification

It takes about 1200 lines of AML code to implement a basic equality checking algorithm.
The code cannot generate an invalid judgment, but it can loop or abort.
The standard library allows the user to put in new extensionality rules and new β-rules.
In addition, it support computation of implicit arguments.

It is the user's responsibility to feed sensible rules to the system. Bad rules can make the system run forever or fail to prove an equality, but they will never produce an invalid judgment.

A recent version of Agda introduced similar user-defined β-rules (but not user-defined extensionality rules). One difference is that in Andromeda a rule can be used locally, for instance inside an abstraction that provides an equality hypothesis.

- Natural numbers: `examples/nat.m31`
- Universes: `examples/universe.m31`

I am going to take you through two examples, interactively in Emacs.

The natural numbers example shows how we axiomatize a structure and the associated computation rules, how we introduce new β-rules, and how we can perform computation (normalization of terms).

The universe example shows how the operations and handlers mechanism helps us with automatic translation from names to types and vice versa. The example could and should be extended to one with a universe of fibrant types.

The AML code is the proof certificate!

- LCF and HOL do not have proof certificates.
- If you have doubts, just run the code again.
- Send the code to your friends, they can run it. And this is in fact going to be the most efficient and useful way of communicating proofs to them.
- Efficiency matters, decidability and termination do not. People can directly control efficiency of AML code but not of some derived certificates.

In every talk about Andromeda somebody realizes that Andromeda does *not* compute proof certificates that could be rechecked, and asks “why don’t you have proof certificates”.

My view is that the AML code *is* the evidence of a judgment!
When you run it, it will implicitly produces a derivation of the judgment.
It can be trusted by design.

Your friends want the source code, not gigabytes of unreadable and unmodifiable proof certificates. Do you know anyone who prefers to download compiled Coq files instead of the source code?

Efficiency matters a great deal. In practice people take great care to produce efficient code and structure their proof development to achieve it. In contrast, proof certificates that are generated by machines cannot be directly controlled, neither in efficiency nor in size.

Also, just to rub Thorsten’s patience, I also claim that termination and decidability are irrelevant in practice. They are poor approximations to efficiency, in fact they are no approximations at all, and they just hinder expressivity and ease of proof developments. Real programmers use general recursion.

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.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, Air Force Materiel Command, USAF.