# Derivations as Computations

Andrej Bauer
University of Ljubljana

ICFP 2019
24th ACM SIGPLAN International Conference on Functional Programming
Berlin – August 2019

Thank you for the invitation, I am greatly honored to be here at ICFP. And it's my first time to this excellent conference and this excellent city.

In this talk I would like to present ongoing work on implementing a proof assistant with user-definable type theories, called Andromeda (version 2). I will present an approach that uses PL techniques that you will hopefully find useful in other contexts as well.

**Philipp Haselwarter**



This is joint work with Philipp Haselwarter, my PhD student who is also attending. You should find him and talk to him to find out what's behind the curtain.

Some of the theoretical parts involving general type theories were done in cooperation with Peter Lumsdaine from Stockholm University.

# Proof assistant wish list

- small trusted kernel
- user-definable dependent type theories
- including user-definable judgemental equality
- no commitment to an ambient type theory
- support common proof-development techniques



To set the stage, let me put up a wish list for a proof assistant that one may want or need to perform experiments in type theory. Our motivation arose from trying to implement a proof assistant for a type theory that was a moving target and lacked good computational properties, namely Vladimir Voevodsky's "homotopy type system".

Here is the wish list:

- We want to keep the trusted code base as small as possible.
- We want user-definable dependent type theories, where the user may manipulate judgmental equalities in arbitrary ways.
- The ambient meta-theory should be very simple, for instance we may want a type theory without product types, or perhaps we do not want to use universes.
- All the while, the proof assistant should support the common techniques that support proof development, such as implicit arguments and coercions, automated equality checking, etc.

We believe Andromeda 2 has a good chance to meet these requirements.

## Has this been done before?

- Coq, Agda, Lean, …
- Isabelle
- Dedukti
- Twelf, Abella, Beluga



Of course, we are not the first ones to think about systems that allow one to describe and use a type theory.

## Has this been done before?

- Coq, Agda, Lean, … – fixed type theory
- Isabelle – commits to ambient HOL
- Dedukti – commits to ambient Πλ-theory
- Twelf, Abella, Beluga – reason *about* theories



Often enough a proof assistant is available that supports a type theory that will serve the purpose. Some of the most popular proof assistants are baed on such expressive type theories. However, if a change is desired or needed, you may be at the mercy of the implementors.

Some proof assistants are specifically designed to support user-specified logics and type theories, and these are broadly of two kinds, according to their primary purpose: tools like Isabelle and Dedukti allow the user to actually *use* the custom theory, while tools like Twelf, Abella and Beluga support meta-level reasoning *about* the the custom theory.

Andromeda is more like the first kind, i.e., the user describes a type theory for the purpose of actually using it. There is no direct support for proving meta-theorems about it. Or to make this point more specific: in Andromeda the user can specify a normalization procedure in a programming meta-language, but they cannot prove inside Andromeda that the procedure works.

# Talk outline

1. General type theories
2. “Derivations as computations”
3. Implementation



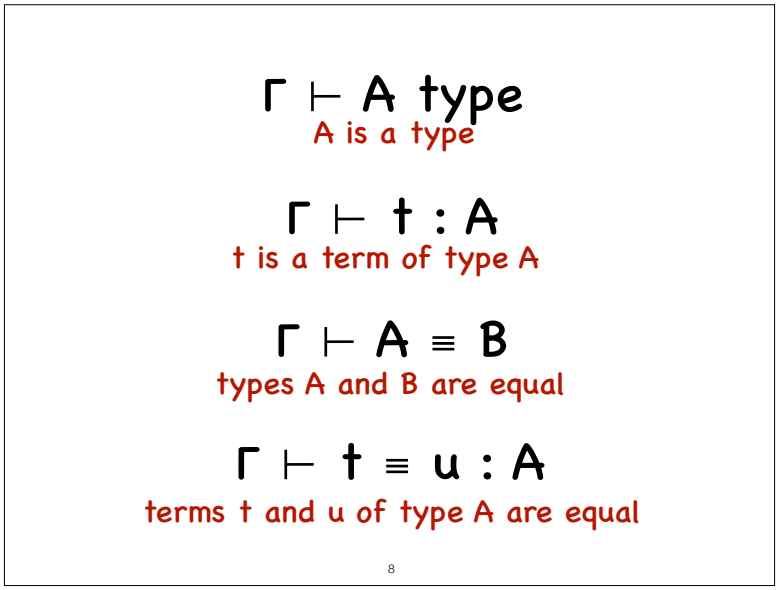
Here is the plan of the rest of the talk.

- We are going to look at what sort of type theories are supported by Andromeda.
- Then we will explain the motto “derivations as computations”, and how it relates to proof assistants.
- And lastly we will look at how the ideas have been implemented in Andromeda 2.

# 1. General type theories

There are many kinds of type theory. In fact the concept of type theory is open ended and difficult to subsume in a single mathematical formalism – which not a bug but a feature.

Our goal was to support a wide variety of type theories, but with a well understood meta-theory and semantics. In a separate ongoing project we are developing such meta-theory with Peter Lumsdaine. As the meta-theory of type theories can be quite technical, I am going to skip most details and give a broad outline.




We support any type theory that uses the four basic judgment forms, in the style of Martin-Löf's type theory, namely: that something is a type, that something is a term, that two type are judgmentally equal, and that two terms are judgmentally equal. The judgements are hypothetical, i.e., they are under a typing context.

Using such judgements, the user may introduce arbitrary term and type constructions and posits inference rules.

There are certain technical restrictions on what inference rules are acceptable, in order to guarantee that the type theory has good meta-theoretic properties, but decidable equality checking is *not* one of them, as one of the examples we want to include is extensional type theory.

One way to explain our approach here is that type theory is more like generalized algebra, rather than a programming language. However, when the user specifies a type theory with computational content, the system will allow them to take advantage of it.

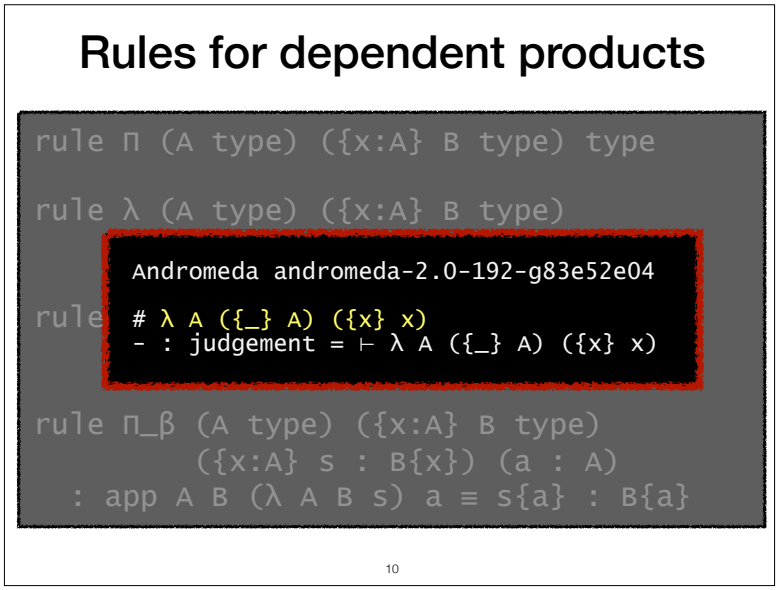
## Rules for dependent products

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

rule λ (A type) ({x:A} B type)
       ({x:A} e : B{x}) : Π A B

rule app (A type) ({x:A} B type)
         (s : Π A B) (a : A) : B{a}

rule Π_β (A type) ({x:A} B type)
         ({x:A} s : B{x}) (a : A)
  : app A B (λ A B s) a ≡ s{a} : B{a}
```



To give you an idea of how this is done in Andromeda, here are the rules for dependent products.

Each rule has a name, which is also the name of the constructor to form the respective term or type.




This is how the user would then write down the identity function.

The syntax is not very friendly, but let us not get derailed by Wadler's Law.

## Equality reflection

```
rule Id (A type) (a : A) (b : A) type

rule equality_reflection
 (A type) (a:A) (b:A) (p:Id A a b) :
 a ≡ b : A
```



Here is the rule of equality reflection. It says that propositionally equal terms are judgmentally equal.

This rule makes extensional type theory undecidable: in order to prove an equality, one has to possibly inhabit an arbitrarily complicated identity type.

# Examples

- Martin-Löf intensional type theory
- Extensional type theory (equality reflection)
- “Book” homotopy type theory
- Universes a la Tarski (including Type : Type)
- Simply-typed and untyped λ-calculus



To give you an idea of the scope of type theories under consideration, let us look at some examples and non-examples.

- Both the intensional and the extensional versions of Martin-Löf type theory are examples.
- Another example is homotopy type theory, as presented in the HoTT book.
- One can have universes, including “type in type”, but they have to be a la Tarski, i.e., we need to distinguish types and their codes. This is so because in the Martin-Löf-style type theories there is a sharp distinction between terms and types. A type cannot be a term, and a term cannot be a type.
- Of course, one need not use fully-blown depndent types, and instead restrict to a simply type theory. One example we experimented with was the untyped λ-calculus.

# Non-examples

- Cubical & cohesive type theory
- Linear type theory
- Pure type systems
- System F
- Universes a la Russell



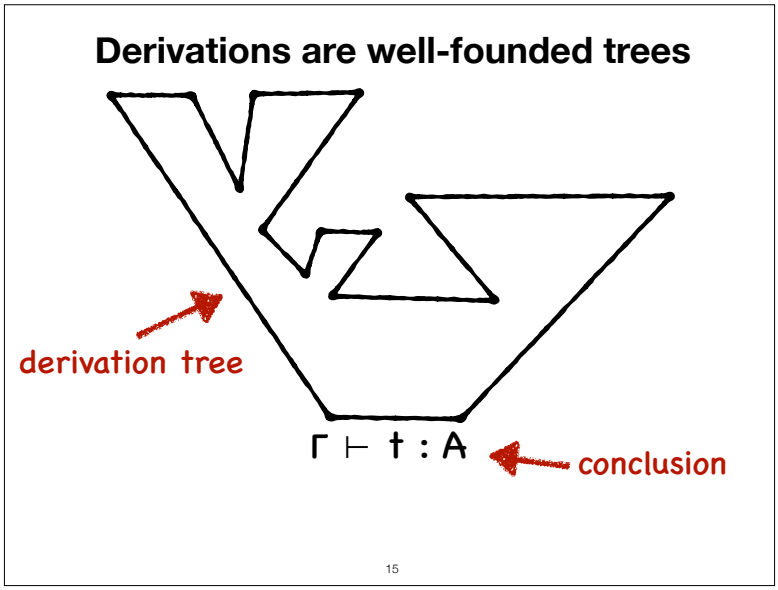
As for non-examples, there are two kinds.

First, there are type theories which treat some types in a special way, or have special conditions on how typing contexts are treated. Thus cubical type theory and various linear type theories are out.

The other kind of non-examples are theories that are incompatible with the Martin-Löf-style conception of types. For instance, System F quantifies over all types, and Russell-style universes confound a type with the term that represents it. However, one can often get a good approximation to such systems by formalizing them in some slightly indirect manner. Anyhow, we hope to improve on this in the future.

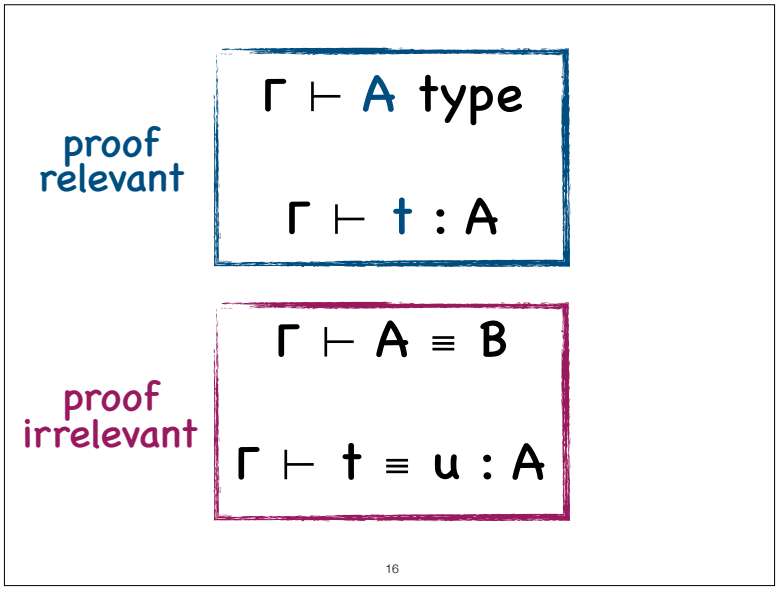
## 2. “Derivations as computations”

Having seen that the scope of general type theories is reasonably wide, we should think about how one might implement them. Rather than doing this in an ad-hoc manner, we want to first identify some guiding principles.



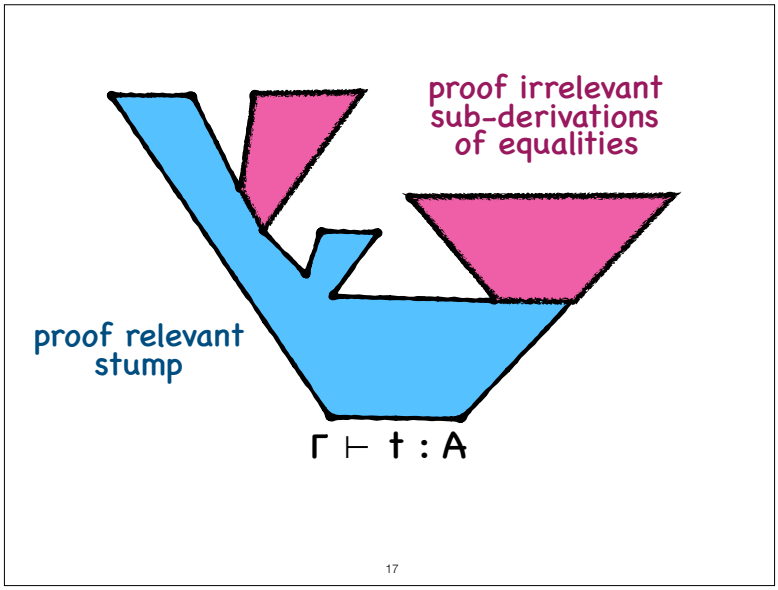

A basic task is to *justify* a given judgement, say that a term t has type A. The form of such justification is a *derivation*, which is a well-founded (finite) tree whose nodes are instances of the given inference rules, and the root  is the desired judgement.

To check that a term t has a type A means that we have to produce such a derivation one way or another. Where does it come from?

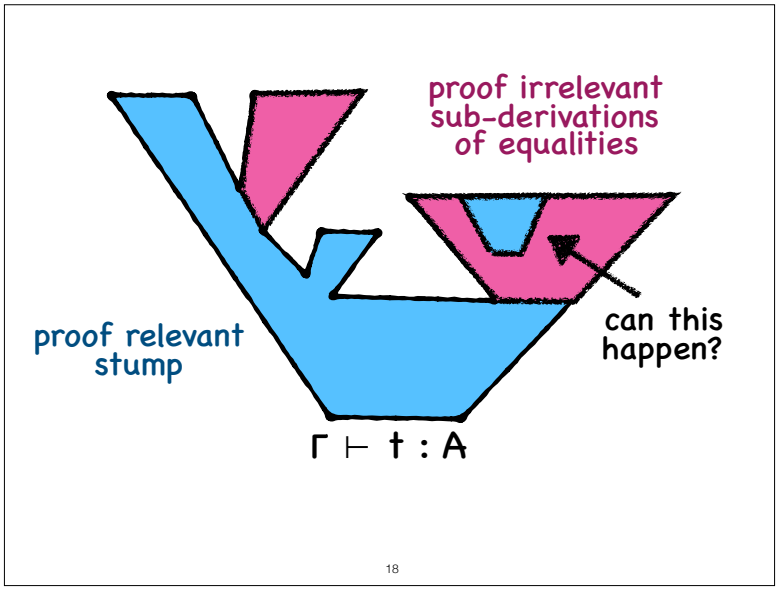



To answer the question, let us first observe that there two kinds of judgements:

- the judgements stating that something is a type and that a term has a type, are *proof relevant* in the sense that they *record* information (namely the type and the term).
- judgmental equalities are *proof-irrelevant*, because they state what equations hold, but do not record any justification for them.



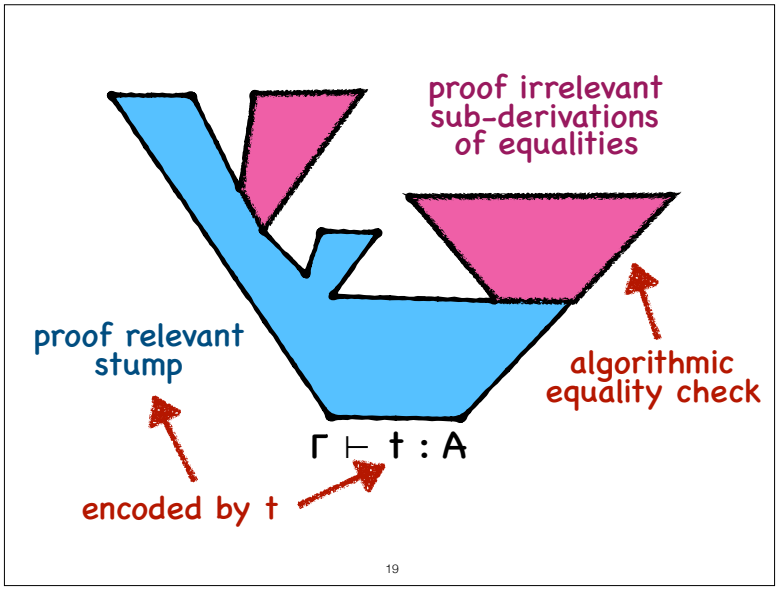

Consequently, a derivation has a proof relevant part (blue), in which only the rules for type and terms are used, and a proof irrelevant parts which derive equalities (red).




Can the relevant and irrelevant parts be interleaved? Can a proof-irrelevant part contain a further sub-derivation that is proof-relevant? In principle yes, and this it is allowed by our system.

However, when proof relevant parts appear inside the proof-irrelevant parts, they do not get recorded in the final term. This is the a source of many complications, as it typically renders equality checking undecidable.

The so-called equality reflection rule from extensional type theory is precisely of this sort.




A computationally well-behaved type theory disallows proof-relevant sub-derivations inside the proof-irrelevant ones. This way the derivations can be reconstructed algorithmically:

- the proof-relevant stump is read off the term t,
- the proof-irrelevant derivations are reconstructed by an equality checking algorithm, usually based on normalization of terms. And because these bits are proof irrelevant, it doesn't matter *which* derivation is found by the algorithm.

A proof assistant does not actually store such a full derivation tree in memory, as it can be huge. What does it actually do?

# A proof assistant is …

```
isDerivable : judgement → bool
```



A very naive way of understanding a proof assistant is to think of it as a function which gets a judgement and checks whether the judgement is derivable. The derivation is *implicit* in the execution trace. It is stored in time, not in space!

However, if proof assistants really worked this way, they would not be helpful at all. The user would have to provide judgements, with all the details, and the machine would just accept or reject them.

# A proof assistant is …

```
elaborate : source → judgement option
```



In reality, the user provides *incomplete* information. They write down source code that *looks like* a term or a type, but lacks certain information (for instance, they may omit implicit arguments). It is the job of the proof assistant to *elaborate* the source to an actual judgement, or fail.

This view is less naive, but still not good enough. We should be free to organize the passage from the source to the judgement in a flexible way.

# A proof assistant is …

```
elaborate : source → T judgement
```



So why not allow other effects as well (embodied here by a monad T) and put them to some good use!

# A proof assistant is …

```
evaluate : source → T judgement
```

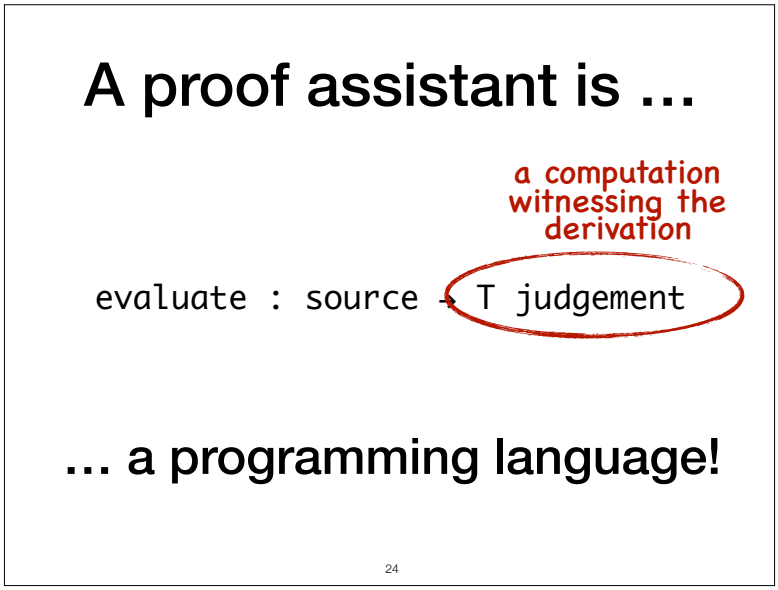
## … a programming language!



Driving this idea to its logical conclusion suggests that a proof assistant should be construed as a programming meta-language.  The famous LCF proof assistant from the 1970's was based on this idea. We are just upgrading it to dependent types and modern PL techniques.

Let us not forget however, we still have to guarantee that only derivable judgements can be computed. This is achieved by making the datatype of judgements abstract that only a trusted kernel may manipulate.

Let me mention in passing that Bob Atkey has done some very interesting related work in which he wrapped a type-checker for a programming language into a monad, with very interesting results.




Thus we arrive at our motto: derivations as computations.

To be quite precise, the computation (program) itself does not yet guarantee that the judgement is derivable, as it can possibly fail when executed. It is really *its execution and successful evaluation to a value* that witnesses the derivability of a judgement.

This situation mirrors the *act of verifying* that a derivation tree is valid, and even less informally, the act of reading and understanding a proof. But enough philosophy, let us see how we can use PL techniques to take advantage of our motto.

## Proof assistant techniques

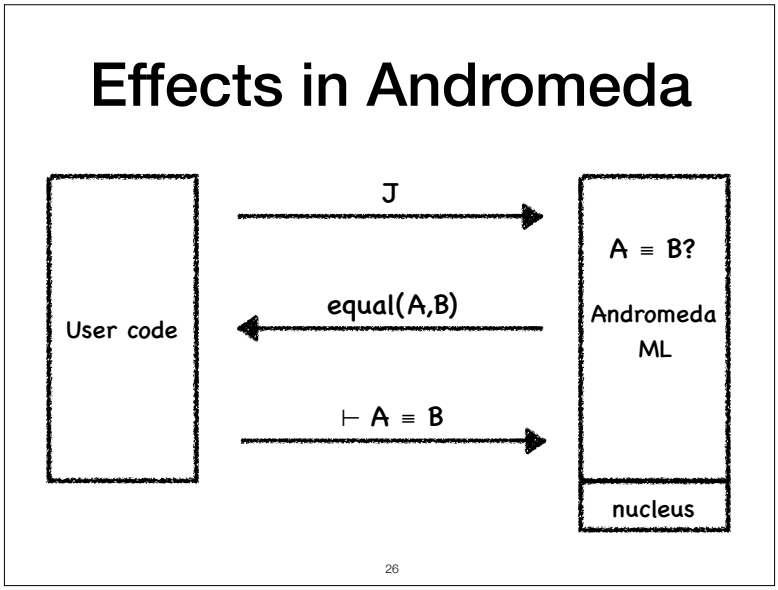
- Equality checking & normalization
- Universe management
- Implicit coercions
- Type classes
- Meta-variables (“Evars”) & unification
- Tactics

A modern proof assistant employs a number of techniques that help with formalization. These are implemented by the designers of the proof assistant, and can be quite complex.

They rely on the particularities of the type theory that is implemented. But in Andromeda we have user-definable type theories, so what should we do? What can we do?

There is little that we *can* do, other than to make it the users’ burden to provide such techniques. Of course, we do not expect every user to implement their own equality checker and type class mechanism – instead the experts would design a library that would deal with a particular type theory.
From a PL perspective we need a mechanism that makes such libraries possible.

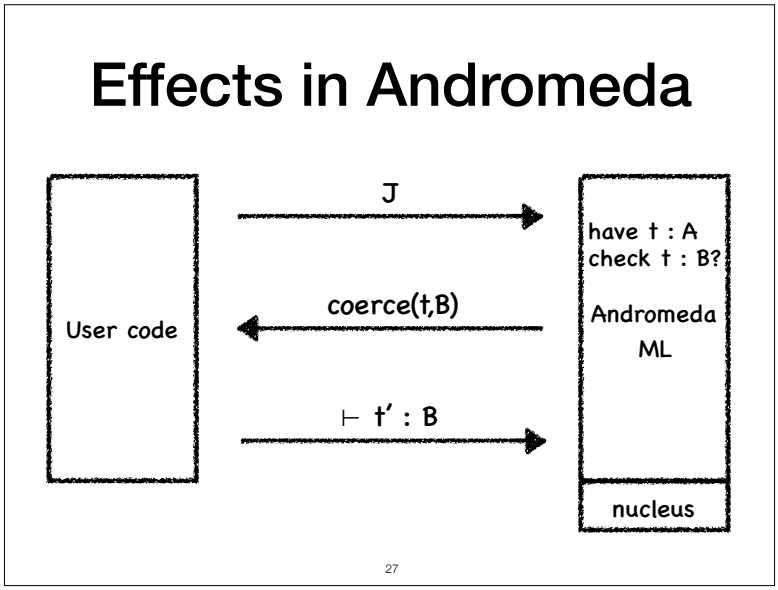
In Andromeda we use algebraic operations and handlers. Let us see how this works in the case of equality checking.




In Andromeda we use algebraic operations and handlers, in the style of the Eff programming language. Here is a typical use.

- The user code wants to compute some judgement
- For the judgement to be valid, some equality A ≡ B needs to hold. Because Andromeda has no built-in equality checker (it cannot, there may not even be one), it consults the user code by triggering an operation equal(A,B)
- The operation is intercepted by a handler which computes the desired judgement A ≡ B. Of course, it does so by further evaluation of code, which may trigger even more operations.
- The judgement is passed back to the interpreter, and evaluation may proceed.

In a sense this is just a fancy callback, and in fact we have not had the need for the full power of handlers. It is quite likely that a simpler mechanism would suffice, although it needs to be dynamic and local, as sometimes one has to use local assumptions to derive an equality.




In fact, an equality check is triggered when there is a mismatch between the actual and expected type of a term. Suppose that, during evaluation of some code J, we need to check that t has type B, but we discover it actually has type A.

We could run the operation equal(A,B) to ask for an equality A ≡ B, like on the previous slide, and then use conversion to cast t to type B. However, if A and B are not equal, the user-space will fail.

Instead ML can trigger an operation coerce, which is handed the term t and the expected type B. It expects to be given back some term t' of type B. In case A ≡ B, t' may be just converted t, but in general it could be anything.

This opens the door for implicit coercions and other techniques that modify a term in order to give it the correct type. For instance, this is how we can automatically convert the code of a type (element of a Tarski universe), to the type it encodes, and vice versa.

We have found the effects & handlers mechanism to be very flexible. But there is nothing special here about using handlers. It should be quite possible to organize this sort of interaction between the proof assistant and the user code in some other fashion, say through callbacks or continuations.

# 3. Implementation

Lastly, let me say a few word about the implementation, and show you an example.

# Andromeda

- LCF-style proof assistant
- abstract data type `judgement`
- trusted nucleus (3232 lines of OCaml code)



Andromeda is implemented in OCaml, of which around 3200 lines of code comprise a trusted nucleus. The nucleus does *only* basic type-theoretic operations, such as applications of inference rules, and inversion of those.

The ML evaluator and user code are of course not trusted.

# Soundness & completeness

J derivable ⇔ ∃p:judgement . p ⇝ J

**Soundness**
Only derivable judgements can be computed.

**Completeness**
Every derivable judgement can be computed.



There is a standard correctness and completeness guarantee.

Philipp Haselwarter's forthcoming PhD thesis is concerned with the question whether Andromeda is sound and complete.

# Andromeda 1 & 2

- Andromeda 1
  - fixed type theory: equality reflection & Type:Type
  - user library: equality checking & implicit arguments
- Andromeda 2
  - user-definable type theory
  - user library: forthcoming (had to attend ICFP)



The original Andromeda, version 1, implemented extensional type theory with equality reflection and Type : Type. We also provided a small user library which implemented (in user space) an equality checking algorithm that worked well, but was slow.

When we wanted to get rid of Type : Type we noticed that, with a bit of work, we could actually get rid of everything, and let the user define type theory entirely. We are almost done, the main thing to still port is an equality checking algorithm that will cover type theories that enjoy normalization. This time we will implemented it in OCaml (naturally, outside the trusted nucleus).


Thank you for your attention. If you like to participate in the efforts, or be our Guniea pig, you can find us online.