Andromeda 2.0

Philipp G. Haselwarter

j.w.w. Andrej Bauer Peter LeFanu Lumsdaine

Foundations Seminar, FMF
08.11.2018

1. The work I will present is being done jointly with Andrej Bauer and Peter Lumsdaine. I hope Peter won't take offence in my mentioning his name in the context of Andromeda, but the connection should be clear in a moment.
2. There are roughly four parts to this talk. I will first talk about general type theories, then briefly give you an idea of how the Andromeda prover works. Then I'll tell you how we implemented general type theories in Andromeda, and finally I want to give a quick demo.

General Type Theories: Motivation

We want meta theorems such as:

Proposition (Uniqueness of typing)

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $\Gamma \vdash A \equiv B$.

1. We want to study dependent type theories. People have been doing this for decades, and they prove theorems such as "uniqueness of typing"
2. Instead of proving such statements for each theory of dependent types – there is a new one roughly every week – we would like to prove them once and for all for a wider class of theories.
3. More importantly, some theorems want to talk about "type theories", or "models of type theories", not just about one type theory, or models for one type theory. say something about initiality here?
4. The class has to be large enough to encompass many established real-world examples, such as MLTT, MLTT with equality reflection, or with universes, or Homotopy Type Theory. Because type theory is notoriously finicky, we are also formalising these definitions and the theorems we prove about them in Coq.

General Type Theories: Motivation

We want meta theorems such as:

Proposition (Uniqueness of typing)

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $\Gamma \vdash A \equiv B$.

- for which type theory?
- stable under extensions of the theory?
- ...

1. We want to study dependent type theories. People have been doing this for decades, and they prove theorems such as "uniqueness of typing"
2. Instead of proving such statements for each theory of dependent types – there is a new one roughly every week – we would like to prove them once and for all for a wider class of theories.
3. More importantly, some theorems want to talk about "type theories", or "models of type theories", not just about one type theory, or models for one type theory. say something about initiality here?
4. The class has to be large enough to encompass many established real-world examples, such as MLTT, MLTT with equality reflection, or with universes, or Homotopy Type Theory. Because type theory is notoriously finicky, we are also formalising these definitions and the theorems we prove about them in Coq.

General Type Theories: Motivation

We want meta theorems such as:

Proposition (Uniqueness of typing)

If $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $\Gamma \vdash A \equiv B$.

- for which type theory?
- stable under extensions of the theory?
- ...

We need a mathematically precise formulation of a class of type theories to answer such questions. Our suggestion: General Type Theories.

1. We want to study dependent type theories. People have been doing this for decades, and they prove theorems such as "uniqueness of typing"
2. Instead of proving such statements for each theory of dependent types – there is a new one roughly every week – we would like to prove them once and for all for a wider class of theories.
3. More importantly, some theorems want to talk about "type theories", or "models of type theories", not just about one type theory, or models for one type theory. say something about initiality here?
4. The class has to be large enough to encompass many established real-world examples, such as MLTT, MLTT with equality reflection, or with universes, or Homotopy Type Theory. Because type theory is notoriously finicky, we are also formalising these definitions and the theorems we prove about them in Coq.

General Type Theories: Overview 1

- Four judgement forms:
  1. $\vdash A \;\; \text{type}$
  2. $\vdash t : A$
  3. $\vdash A \equiv B$
  4. $\vdash s \;\equiv\; t : A$
- structural rules: variables, conversion, etc.

1. These questions kept Peter up at night, and he got Andrej and me on board developing this notion of general type theories. I will not go into detail here, because that would be a talk in itself, but only sketch enough so that I can talk about how it relates to Andromeda.
2. (bullet 1) Contexts are implicit in schemata, but tacitly we view all of these judgements as living over "some context".
3. We make no simplifying assumptions like that we always work in a universe or that there is a type of all types.
4. The structural rules are at the core of dependent type theory, and we are committed to them.
5. If you think there should be a judgement for contexts: They are a derived notion.

General Type Theories: Overview 2

- a signature mapping rule names to rules
- a well-ordering on the signature precludes circularity
- three notions of rule, incrementally singling out "good rules"
  - closure conditions
  - flat rules
  - well-typed rules
- each judgement in a rule is given over a local context

1. A theory is then given by a signature, by which we mean a well-ordered collection of further schematic rules, each extending the theory defined thus far. Such an extension specifies a closure rule that the derivability relation has to satisfy.
2. In the development with Peter, we are very careful to present these theory extensions in stages, so that they can be independent. This is important because we want to be as general as possible, for instance, we don't want to introduce finiteness assumptions unnecessarily.
3. However, if we want to implement these things on a computer, we know that certain restrictions will hold; for instance rules will always be given to us in a linear order, and their number will be finite because the source code is.
4. An important point of GTT as formalised is that we pay close attention to morphisms of signatures and translations of theories. This is not implemented in Andromeda yet, but Anja will be giving a seminar on it next week.

Example (1)

$$\frac{\Gamma,\ x{:}A \vdash B\ \ \text{type} \qquad \Gamma,\ x{:}A \vdash t : B}{\Gamma \vdash \lambda_{(x:A)}\, t : \Pi_{(x:A)}\, B}\ \lambda\text{-intro}$$

1. This is a reasonable statement of the introduction rule for lambda: Given a type family B and a term for an abstract x, we can form a dependent function from A to B. However, this style is not suitable for GTTs.
2. Remember that we only care about the local part of the context, the rest is implicit. We don't want to see it. We require that every meta-variable be introduced as the "head" of a judgement, so for instance we want to know exactly what A is before we encounter it in the local context of B.
3. Notice also how the local contexts already tell us how to bind things, so the annotation in the conclusion are superfluous. Finally, the name of the rule, λ here in the conclusion, identifies the rule in the signature. There is no need to repeat the arguments – a type A, a family B, a term t – because this rule takes exactly the premises. The result is fully annotated syntax, for instance the "bad" rule does not know about the codomain of the function. If you think this is annoying, let me remind you that some type theories such as ETT require full annotations, and we want to cover them in GTT.

Example (1)

$$\xcancel{\dfrac{\Gamma,\ x{:}A \vdash B\ \text{type} \qquad \Gamma,\ x{:}A \vdash t : B}{\Gamma \vdash \lambda_{(x:A)}\, t : \Pi_{(x:A)}\, B}\ \lambda\text{-intro}}$$

1. This is a reasonable statement of the introduction rule for lambda: Given a type family B and a term for an abstract x, we can form a dependent function from A to B. However, this style is not suitable for GTTs.
2. Remember that we only care about the local part of the context, the rest is implicit. We don't want to see it. We require that every meta-variable be introduced as the "head" of a judgement, so for instance we want to know exactly what A is before we encounter it in the local context of B.
3. Notice also how the local contexts already tell us how to bind things, so the annotation in the conclusion are superfluous. Finally, the name of the rule, λ here in the conclusion, identifies the rule in the signature. There is no need to repeat the arguments – a type A, a family B, a term t – because this rule takes exactly the premises. The result is fully annotated syntax, for instance the "bad" rule does not know about the codomain of the function. If you think this is annoying, let me remind you that some type theories such as ETT require full annotations, and we want to cover them in GTT.

Example (1)

$$\xcancel{\dfrac{\Gamma,\ x{:}A \vdash B\ \ \text{type} \qquad \Gamma,\ x{:}A \vdash t : B}{\Gamma \vdash \lambda_{(x:A)}\, t : \Pi_{(x:A)}\, B}\ \lambda\text{-intro}}$$

$$\dfrac{\vdash A\ \ \text{type} \qquad x{:}A \vdash B\ \ \text{type} \qquad x{:}A \vdash t : B}{\vdash - : \Pi\, A\, B}\ \lambda$$

1. This is a reasonable statement of the introduction rule for lambda: Given a type family B and a term for an abstract x, we can form a dependent function from A to B. However, this style is not suitable for GTTs.
2. Remember that we only care about the local part of the context, the rest is implicit. We don't want to see it. We require that every meta-variable be introduced as the "head" of a judgement, so for instance we want to know exactly what A is before we encounter it in the local context of B.
3. Notice also how the local contexts already tell us how to bind things, so the annotation in the conclusion are superfluous. Finally, the name of the rule, λ here in the conclusion, identifies the rule in the signature. There is no need to repeat the arguments – a type A, a family B, a term t – because this rule takes exactly the premises. The result is fully annotated syntax, for instance the "bad" rule does not know about the codomain of the function. If you think this is annoying, let me remind you that some type theories such as ETT require full annotations, and we want to cover them in GTT.

Example (1)

$$\xcancel{\dfrac{\Gamma,\ x{:}A \vdash B\ \ \text{type} \qquad \Gamma,\ x{:}A \vdash t : B}{\Gamma \vdash \lambda_{(x:A)}\, t : \Pi_{(x:A)}\, B}\ \lambda\text{-intro}}$$

$$\dfrac{\vdash A\ \ \text{type} \qquad x{:}A \vdash B\ \ \text{type} \qquad x{:}A \vdash t : B}{\vdash - : \Pi\, A\, B}\ \lambda$$

Used as

$$\lambda\, A\, B\, t$$

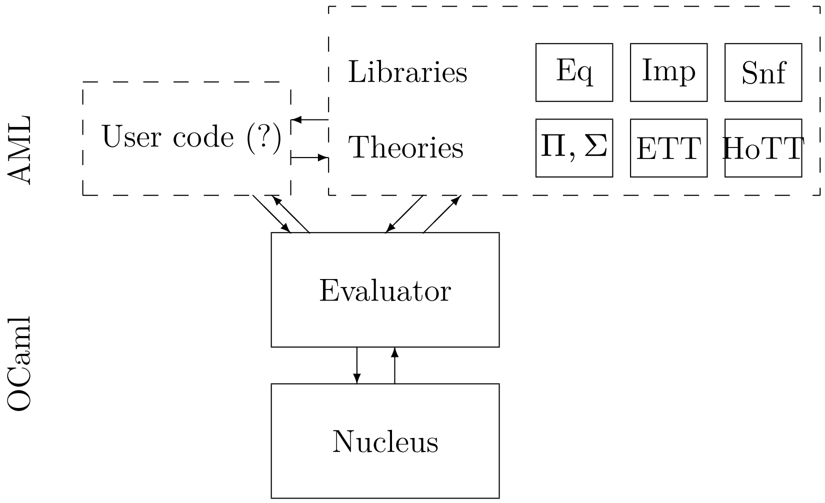
1. This is a reasonable statement of the introduction rule for lambda: Given a type family B and a term for an abstract x, we can form a dependent function from A to B. However, this style is not suitable for GTTs.
2. Remember that we only care about the local part of the context, the rest is implicit. We don't want to see it. We require that every meta-variable be introduced as the "head" of a judgement, so for instance we want to know exactly what A is before we encounter it in the local context of B.
3. Notice also how the local contexts already tell us how to bind things, so the annotation in the conclusion are superfluous. Finally, the name of the rule, λ here in the conclusion, identifies the rule in the signature. There is no need to repeat the arguments – a type A, a family B, a term t – because this rule takes exactly the premises. The result is fully annotated syntax, for instance the "bad" rule does not know about the codomain of the function. If you think this is annoying, let me remind you that some type theories such as ETT require full annotations, and we want to cover them in GTT.

Example (2)

$$\dfrac{\vdash A \;\; \text{type} \qquad x{:}A \vdash B \;\; \text{type} \qquad x{:}A \vdash t : B}{\vdash - : \Pi\, A\, B}\;\lambda$$

1. Let me explain a little bit the terminology for rules here.

Andromeda



1. Roughly, Andromeda has three components: the Nucleus and the Evaluator which are written in OCaml and some libraries written in the Andromeda Meta Language.
2. The Nucleus here at the foundation implements the structural rules such as conversion and variable handling, meta-theorems, alpha-equality, and for some reason export to JSON.
3. The Evaluator executes the User code written in AML and calls the Nucleus to deal with the type theory part of an AML program. To the user, the type theory part of AML looks like smart constructors.
4. The user may use some libraries we provide, for instance facts about equality or implicit arguments to make writing proofs less cumbersome.
5. But because we cannot hope to write a decision procedure for type checking, this is not a one-way process. Instead, when the evaluator has questions it can't figure out by itself, it asks the libraries and the user program. So control is handed back and forth between the different components.

GTT in AML 1

- ML/Eff style meta language for GTTs
- four built-in abstract data types for the judgement forms
- judgements are abstracted to indicate local contexts
- static checking of judgement forms and lengths of local contexts
- dynamic checking of the judgements themselves
- user-defined GTT rules are also checked

1. The Andromeda meta-language, AML
2. Local contexts are the only form of binders present in the language. The usual things that you expect to be binding, such as a lambda abstraction, or a Π-type are "binding" by their way of handling local contexts.
3. Statically: Scoping, arities
4. The different stages in the life cycle of a rule also make an appearance in Andromeda, as parsing, ML type-checking, and evaluation of the rule declaration, i.e. checking the actually well-formedness of the rule
5. Judgements do not carry their contexts. Suppose they did. When we want to combine judgements constructed in different contexts, we would be forced explicitly use weakening and exchange rules, or construct context morphisms. It would be much too laborious. Instead, each variable simply carries its type.

GTT meta-theory in the nucleus

Andromeda relies on the following meta theorems:

1. Uniqueness of typing
2. Presuppositions theorem (“Sanity” in Ljubljana parlance)
3. inversion principles
4. admissibility of substitution

1. Andromeda’s nucleus believes in certain meta-theorems that we have proved for GTT’s. They are exploited in various ways internally, and exported in the interface to AML.
2. substitution is actually baked into GTT at the moment, but it might disappear. We do however want it in Andromeda.
3. pattern matching

Derivations modulo proof-irrelevance

```
type ty =
  | TypeConstructor of Name.constructor * argument list
  | TypeMeta of type_meta * term list
and term =
  | TermAtom of atom
  | TermBound of bound
  | TermConstructor of Name.constructor * argument list
  | TermMeta of term_meta * term list
  | TermConvert of term * assumption * ty
and eq_type = EqType of assumption * ty * ty
and eq_term = EqTerm of assumption * term * term * ty
and atom = { atom_name : Name.atom ; atom_type : ty }
and 't meta = { meta_name : Name.meta ; meta_type : 't }
and assumption = (ty, premise_boundary) Assumption.t
```

1. So what's actually implemented in the nucleus? What do the data-types look like, and how do we implement meta-theorems? Actually, "desirable meta-theorems" have been informing the implementation of Andromeda for a long time, even before we knew about GTT, so we had a good idea of what we would need to do.
2. There are no terms. There are only derivations.
3. Type- and term-formers are stored. Structural rules are also stored, in particular conversion.
4. Derivations of equality judgements are not stored. Instead, we keep only the conclusion and the set of assumptions that was used (necessary for reconstructing the context and proving meta-theorems)
5. While we do not want to bother the user with context management, we can only make sense of judgements if we know what context they were constructed in.
6. Andromeda and GTT slightly differ in their treatment of meta-variables. In GTT, they are simply treated as temporary extensions of the signature. Because we aren't paranoid, we do not store the rules we used in the assumptions, but we do have to keep track of the meta variables. So instead of adding them to the signature, each meta variable knows its own arity.

Demo

Demo

Thank you

Bonus: Why strengthening matters

- Strengthening is usually formulated in terms of free variables of a judgement
- because of proof irrelevance, this has to be relativised to free variables in the derivation of a judgement

- inversion principles are gimped without strengthening, because all sub-terms of a judgement would share the same context