Andromeda 2.0

Anja Petković[1]

[1] University of Ljubljana, Slovenia

Logic seminar,
Stockholm, 13.11.2019

j.w.w. Andrej Bauer and Philipp G. Haselwarter

[1] This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

1. Hi, I'm Anja from University of Ljubljana. Thank you for inviting me to your seminar.
2. I will give a demo of Andromeda 2.0. It is work in progress, so if you are brave enough to try it out, you may encounter bugs and missing features.
3. Installation instructions: http://www.andromeda-prover.org/install.html
4. The files are compatible with the andromeda version at commit 3170f2e

What is Andromeda?

- Proof assistant designed to support user-definable type theories
- Very different from Andromeda 1.0
- General type theories (in the sense of Bauer, Haselwarter and Lumsdaine), related work: Brunerie
- Trusted kernel Nucleus + Andromeda Meta Language (AML)
- OCaml-like syntax
- file extension .m31

1. Andromeda is in its essence a proof assistant, designed to support user-definable type theories. Meaning the user is able to specify which rules of type theory hold and then use them accordingly.
2. For those who know about the previous versions of Andromeda, they supported extensional type theory with Type:Type. Now we actually define our type theory by specifiing rules, we will see how exactly we do that a little later.
3. We can input general type theories in the sense of BHL, with which I think most of you are familiar with, for example extensional MLTT, intensional MLTT, book HoTT etc. It cannot support type theories with special features like cubical TT, because we cannot express the external interval with rules. Guillaume has also been working on a definition of general type theories, that is a little different from what we have here, but nonetheless many ideas are similar.
4. Andromeda is comprised of a trusted kernel called Nucleus and the so called Andromeda Meta Language (AML), which supplies the usual functionality of a programming language. We will just briefly take a look at what we can do, but will not focus so much on that.
5. Andromeda is coded in Ocaml and its syntax resembles it
6. supports unicode
7. Origin of the name and file extension

Quick summary of AML syntax

- typed language
- ;;
- lists
- let binding
- functions
- pattern matching

1. file AML-intro.m31
2. AML is a strongly typed language
3. We end the line with double colon ;;. Sometimes we may omit it, if it is clear from the syntax that it is actually there.
4. types in AML are pre-fixed by ml, so mltypes, to be distinguished from types in TT
5. we have some built in types like mlunit, mlist (polymorphic), some are in module ML (like option types ML. Some ..., ML.None) and we can define our own types
6. we use the syntax let .... in .... to bind variables in AML. We can also make functions by using let name args = ...
7. pattern matching is the default eliminator for lists, inductive types etc.
8. we also have special mltypes for judgements, boundaries and derivations - we will see shortly.
9. AML also supports defining your own inductive types and handler in eff-style (for algebraic effects), but we will not get into that

Quick summary of AML syntax

- typed language
- ;;
- lists
- let binding
- functions
- pattern matching

Also supports:

- inductive types
- handlers

1. file AML-intro.m31
2. AML is a strongly typed language
3. We end the line with double colon ;;. Sometimes we may omit it, if it is clear from the syntax that it is actually there.
4. types in AML are pre-fixed by ml, so mltypes, to be distinguished from types in TT
5. we have some built in types like mlunit, mlist (polymorphic), some are in module ML (like option types ML. Some ..., ML.None) and we can define our own types
6. we use the syntax let .... in .... to bind variables in AML. We can also make functions by using let name args = ...
7. pattern matching is the default eliminator for lists, inductive types etc.
8. we also have special mltypes for judgements, boundaries and derivations - we will see shortly.
9. AML also supports defining your own inductive types and handler in eff-style (for algebraic effects), but we will not get into that

Concepts from general type theories

- 4 kinds of judgements
- boundaries
- contexts (handled automatically by Nucleus)
- abstraction (primitive)
- rule

Also:

- derived rules
- congruence rules
- equality checker: WIP, future work

1. This is sort of a table of contents for today's talk. We will not go in this order, but these are the concepts I would like to present.
2. You are already familiar with these from general type theoies (and your own intuition as well), we'll se how they are implemented in Andromeda
3. contexts are handled automatically by Nucleus and the user does not have to deal with them explicitly. We do however get the contexts of judgements back.
4. Abstraction, unlike in general TTs is a primitive notion. We have a so called abstracted judgement, that binds specified variables. This is useful for specifying rules.
5. Transitivity and symmetry we can define, but Nucleus believes in them and uses them.

Judgements

- 4 kinds of judgements

$$\Gamma \vdash \mathbb{N} \text{ type}$$
$$\Gamma \vdash z : \mathbb{N}$$
$$\Gamma \vdash \mathbb{N} \equiv \mathbb{N}$$
$$\Gamma \vdash sz \equiv sz : \mathbb{N}$$

- AML type *judgement*
- keyword *rule*

1. file basic-judgements.m31
2. We start with judgements. We have 4 kinds of judgements (show). I will start with an easy example, $\mathbb{N}$.
3. We introduce judgements by using the rules. When we apply the rules, we get judgements. Rules with no parameters are constants, so judgements.
4. The name of the rule will be the constructor
5. We give a rule iteratively, it is important what comes first, for typechecking in the Nucleus.

Abstraction as a premise in the rule

Induction on natural numbers

$$\frac{\Gamma, x : \mathbb{N} \vdash C \textit{ type} \qquad \Gamma \vdash \text{base} : C(z) \qquad \Gamma, n : \mathbb{N}, c_n : C(n) \vdash \text{step} : C(sn) \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash \mathbb{N}_{\text{ind}}(C, \text{base}, \text{step}, k)) : C(n)}$$

β-rule for zero

$$\frac{\Gamma, x : \mathbb{N} \vdash C \textit{ type} \qquad \Gamma \vdash \text{base} : C(z) \qquad \Gamma, n : \mathbb{N}, c_n : C(n) \vdash \text{step} : C(sn)}{\Gamma \vdash \mathbb{N}_{\text{ind}}(C, \text{base}, \text{step}, z) \equiv \text{base}) : C(z)}$$

β-rule for successor

$$\frac{\Gamma, x : \mathbb{N} \vdash C \textit{ type} \qquad \Gamma \vdash \text{base} : C(z) \qquad \Gamma, n : \mathbb{N}, c_n : C(n) \vdash \text{step} : C(sn) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbb{N}_{\text{ind}}(C, \text{base}, \text{step}, sn) \equiv \text{step}(n, \mathbb{N}_{\text{ind}}(C, \text{base}, \text{step}, n)) : C(sn)}$$

1. We usually think of contexts as lists of vriables with their types. So when we write $\Gamma, x : A \vdash z : \mathbb{N}$, we know what the distinguished last variable is. But what if contexts are not lists? Moreover, the context in the rule basically Γ, the special variables belong to the judgement on the right more than to the context.
2. Let us see how we would say these same things using abstractions

Abstraction as a premise in the rule

Induction on natural numbers

$$\frac{\Gamma \vdash \{x : \mathbb{N}\}C \ \textit{type} \qquad \Gamma \vdash \mathsf{base} : C(z) \qquad \Gamma \vdash \{n : \mathbb{N}\}\{c_n : C(n)\}\mathsf{step} : C(sn) \qquad \Gamma \vdash k : \mathbb{N}}{\Gamma \vdash \mathbb{N}_{\mathsf{ind}}(C, \mathsf{base}, \mathsf{step}, k)) : C(n)}$$

β-rule for zero

$$\frac{\Gamma \vdash \{x : \mathbb{N}\}C \ \textit{type} \qquad \Gamma \vdash \mathsf{base} : C(z) \qquad \Gamma \vdash \{n : \mathbb{N}\}\{c_n : C(n)\}\mathsf{step} : C(sn)}{\Gamma \vdash \mathbb{N}_{\mathsf{ind}}(C, \mathsf{base}, \mathsf{step}, z) \equiv \mathsf{base}) : C(z)}$$

β-rule for successor

$$\frac{\Gamma \vdash \{x : \mathbb{N}\}C \ \textit{type} \qquad \Gamma \vdash \mathsf{base} : C(z) \qquad \Gamma \vdash \{n : \mathbb{N}\}\{c_n : C(n)\}\mathsf{step} : C(sn) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbb{N}_{\mathsf{ind}}(C, \mathsf{base}, \mathsf{step}, sn) \equiv \mathsf{step}(n, \mathbb{N}_{\mathsf{ind}}(C, \mathsf{base}, \mathsf{step}, n)) : C(sn)}$$

1. code in the file basic-judgements.m31, paragraph end

Abstraction

Abstraction is a primitive notion.

- $\{x\colon \mathbb{N}\}sx$
- instantiation of abstraction
- free variables constructed with *fresh*
- abstracting free variables with *abstract*

1. file basic-judgement.m31 paragraph abstraction
2. abstraction is primitive and not as an arity of a symbol as in general type theories
3. instantiation of abstraction using curly braces.
4. we abstract the entire judgement! We cannot equate two abstractions for example.
5. abstracted judgements serve as the arguments for the rules. Let us code up the induction for natural numbers.

Dependent sum

$$\frac{\Gamma \vdash A \ \textit{type} \qquad \Gamma \vdash \{x : A\}B \ \textit{type}}{\Gamma \vdash \Sigma(A, B) \ \textit{type}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \ \textit{type} \\ \Gamma \vdash \{x : A\}B \ \textit{type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\{a\}\end{array}}{\Gamma \vdash \mathsf{pair}(A, B, a, b) : \Sigma(A, B)}$$

$$\frac{\Gamma \vdash A \ \textit{type} \qquad \Gamma \vdash \{x : A\}B \ \textit{type} \qquad \Gamma \vdash s : \Sigma(A, B)}{\Gamma \vdash \pi_1(A, B, s) : A}$$

$$\frac{\Gamma \vdash A \ \textit{type} \qquad \Gamma \vdash \{x : A\}B \ \textit{type} \qquad \Gamma \vdash s : \Sigma(A, B)}{\Gamma \vdash \pi_2(A, B, s) : B\{\pi_1(A, B, s)\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \ \textit{type} \\ \Gamma \vdash \{x : A\}B \ \textit{type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B\{a\}\end{array}}{\Gamma \vdash \pi_1(A, B, \mathsf{pair}(A, B, a, b)) \equiv a : A}$$

1. file dependent-sum.m31
2. how do we do the second projection? We need convert and congruence rules first.
3. But to really make use of these things, we use derived rules.

Derived rules

Rules, we can *derive* from existing rules.

$$\frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash s(s(s(z))) : \mathbb{N}}$$

Recursion on $\mathbb{N}$ from induction on $\mathbb{N}$.

1. file basic-judgements.m31, part derived rules
2. Derived rules are the kind of rules we can derive from previous ones
3. Derived rule is NOT a tactic: if it were a tactic, then every time we use a tactic we make a whole derivation. But when we use derived rule, all the intermediate checking is done only once and we directly get the correct judgement.
4. We use derived rules to make sort of parametric judgements
5. Another typical example would be deriving app-f usig J-rule.

Congruence rules and convert

- *convert tm tyEq*
- *congruence* makes an instance of a congruence rule
- congruence rule for Π-types:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash A' \text{ type} \qquad \Gamma, x : A \vdash B \text{ type} \\ \Gamma, x : A' \vdash B' \text{ type} \qquad \Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \equiv B'}{\Gamma \vdash \Pi(A, B) \equiv \Pi(A', B')}$$

1. See file dependent-product.m31
2. Convert takes a term judgement $a : A$ and a type equation $A \equiv A'$ and returns a term judgement $a : A'$.
3. Mathematically convert does not do anything, but we have to convince the nucleus the types match. Since Andromeda is not proof-relevant, the resulting judgement will not now which proof of the equality we used. Every term also comes with its *nautral* type.
4. Computation/primitive function/builtin congruence makes an instance of a congruence rule. We first list the lhs and rhs and then enough of the presuppositions, missing equations.
5. Show second projection of the dependent sum in file dependent-sum.m31

Boundaries

4 kinds of boundaries

$$\Gamma \vdash ?\ \textit{type}$$
$$\Gamma \vdash ? : A$$
$$\Gamma \vdash A \equiv B$$
$$\Gamma \vdash a \equiv b : A$$

1. file boundaries.m31
2. Boundary is a data structure in the Nucleus and we also have pattern matching on boundaries in AML
3. Nucleus sees them as types of metavariables (for example in derived rules)
4. Programer can understand boundary as an unfinished goal
5. We can abstract boundaries just like judgements.

Match on judgements and boundaries

- We can do pattern matching on boundaries and judgements.
- Bound variables become free!
- Annotation necessary
- match on judgement = inversion

1. file boundaries.m31
2. When we pattern match, we can introduce new free variables from bound ones. Andromeda knows how to handle that. We can use abstract on those later.
3. Deconstructor *match* takes apart a judgement in AML and acts like inversion.
4. We cannot pattern match on proofs of equations, because Andromeda is proof irrelevant and the proof is gone.
5. Show checking judgements against boundaries.

Tarski universes

```
rule U type ;;
rule El (_ : U) type ;;

rule Nat type ;;
rule nat : U ;;
rule El_nat : El nat == Nat ;;
```