Effective Metatheory for Type Theory

Philipp G. Haselwarter

January 11, 2022

Fakulteta za matematiko in fiziko
Univerza v Ljubljani

Hi everybody, my name is Philipp Haselwarter, thank you for coming to the presentation of my thesis *effective metatheory for type theory*. Before we begin, I'd like to thank Andrej for making this thesis possible.

Thesis overview

1. The aim of this thesis was to give an effective metatheory for type theory, and this is achieved in three main parts. Now I'll give a brief overview of the motivation for each of these parts, and say a few words about my solutions. In the rest of the presentation I will talk about the contributions of each part.
2. pause
3. pause
4. pause

Thesis overview

1. Finitary type theories
   - Motivation: No precise definition of *type theories*.
   - Solution: Define FTTs, prove general metatheorems.

1. pause
2. The first step towards providing a general metatheory for type theory is to define what type theories *are*. We come up with new type theories all the time, but there is no mathematically precise definition of "type theories". As a result, there can be no general metatheory, and theorems such as admissibility of substitution or inversion principles are re-proven over and over.
3. My solution to this issue is the notion of *finitary type theories*. Finitary type theories encompass a wide range of familiar type theories such as Martin-Löf type theory, the calculus of constructions, or book HoTT. We can faithfully recover familiar examples, and also prove general metatheorems that we expect to hold for type theories.
4. pause
5. pause

Thesis overview

1. Finitary type theories
   - Motivation: No precise definition of *type theories*.
   - Solution: Define FTTs, prove general metatheorems.
2. Context-free type theories
   - Motivation: "Exotic" FTTs can't directly be implemented.
   - Solution: No explicit contexts, track variables. Same metatheorems plus strengthening, construct translations.

1. pause
2. pause
3. The second step towards an effective metatheory is to give a formulation of finitary type theories that is suitable for computer implementation.
4. A proof assistant should *assist* the user in the construction of a proof. We need a proof assistant that can handle *user defined* type theories. Finitary type theories are not directly suitable for implementation. For example, the equality reflection rule of extensional Martin-Löf type theory introduces a strong form of proof irrelevance, which causes strengthening to fail.
5. So, here I propose a reformulation of finitary type theories without explicit contexts. This takes care of the strangeness of rules such as equality reflection. We also recover the same metatheorems that we had for finitary type theories. We prove that strengthening *does hold* for context free type theories, and we give a translation between the two formalisms.
6. pause

Thesis overview

1. Finitary type theories
   - Motivation: No precise definition of *type theories*.
   - Solution: Define FTTs, prove general metatheorems.
2. Context-free type theories
   - Motivation: "Exotic" FTTs can't directly be implemented.
   - Solution: No explicit contexts, track variables. Same metatheorems plus strengthening, construct translations.
3. Effectful metalanguage for type theories
   - Motivation: Computer support for user defined type theories.
   - Solution: Andromeda ML, with bidirectional evaluation, effect operations & runners.

1. pause
2. pause
3. pause
4. The final missing piece for an effective metatheory is to embed context-free type theories into a programming language. This programming language should help users write the algorithms that we expect from proof assistants, and allow them to manipulate judgements of context-free type theories.
5. Here, I introduce the Andromeda metalanguage. It has the usual features of an ML-style language, including higher order functions, and so on. It allows the user to define their own context-free type theory, and build judgements of that theory. It assists the user in these tasks through *bidirectional evaluation* and *operations and runners*.

Part 1: Finitary type theories

- define terms and types, judgements, rules, and type theories

1. So, what are finitary type theories? I won't say too much about the precise definition, but it is pretty much exactly what you would expect.
2. Finitary type theories encompass a wide variety of known theories, so long as they use intuitionistic contexts and can be phrased with the traditional four judgement forms for types, terms, and type- and term-equality.
3. pause
4. pause

Part 1: Finitary type theories

- define terms and types, judgements, rules, and type theories
- Examples: Martin-Löf TTs, CoC, book HoTT, ...

1. pause
2. Examples include tame theories such as simple type theories, or intensional type theory, but also wild theories such as extensional type theory. Theories with other judgement forms such as cubical type theory or for instance linear type theories are not currently within the scope of finitary type theories.
3. pause

Part 1: Finitary type theories

- define terms and types, judgements, rules, and type theories
- Examples: Martin-Löf TTs, CoC, book HoTT, ...
- well-formedness criteria: raw, finitary, standard

1. pause
2. Before attacking the metatheory, we formulate three general well-formedness criteria.
   Roughly speaking, a raw type theory is simply well-scoped and requires that each rule includes enough premises to make sense of the metavariables.
   Finitary type theories rule out circular dependencies between rules.
   A *standard* type theory features fully annotated syntax.
3. pause

Part 1: Finitary type theories

- define terms and types, judgements, rules, and type theories
- Examples: Martin-Löf TTs, CoC, book HoTT, ...
- well-formedness criteria: raw, finitary, standard
- metatheorems:
  - raw: admissibility of substitution and equality substitution,
  - raw: admissibility of instantiation of metavariables and equality instantiation,
  - raw: derivability of presuppositions,
  - finitary: admissibility of "economic" rules
  - standard: inversion principles,
  - standard: uniqueness of typing.

1. pause
2. pause
3. We can now prove a series of metatheorems, as we would expect them for type theories. Raw theories admit substitution and derive presuppositions, finitary theories admit certain "economic" rules, and standard theories admit inversion principles.

Part 2: Context-free type theories

Context-free type theories are...

- a reformulation of FTTs without explicit contexts
- translatable from and to FTTs
- the basis for implementation (Andromeda 2)

1. Implementing finitary type theories is tricky, because we have to accommodate type theories that are not very well-behaved.

   So instead of implementing them directly, we introduce context-free type theories, which are a reformulation of finitary type theories without explicit contexts.
2. But we'll see that we lose nothing by using this reformulation, because we can translate theories and judgements and derivations between the two formalisms.
3. pause
4. pause

Part 2: Context-free type theories

Context-free type theories are...

- a reformulation of FTTs without explicit contexts
- translatable from and to FTTs
- the basis for implementation (Andromeda 2)

Problems with implementing finitary type theories:

- joining contexts (when forward chaining)
- strengthening fails (e.g. equality reflection)
- effectful metalanguage can cause scope extrusion

$$\frac{\Gamma_1 \vdash s : A \qquad \Gamma_2 \vdash t : A \qquad \Gamma_3 \vdash p : \mathrm{Eq}(A, s, t)}{??? \vdash s \equiv t : A}$$

$\lambda(x : A).\ r := x\,;\ x$

1. pause
2. Let's focus for a moment on the difficulties we encounter if we want to implement finitary type theories.
   If we think of derivation trees being constructed from the premises to the conclusion, we have to join together different contexts, while respecting the dependency graph of each context.
   In the rule to the right, we have three premises with different contexts gamma 1, gamma 2, and gamma 3. What context should we use for the conclusion?
   Strengthening can fail for certain type theories. For example, the conclusion of the equality reflection rule here on the right does not mention the term p, so we cannot read the variables used in a derivation off the conclusion of a judgement.
   And finally, if we work in an effectful meta-language, variables can escape their scope. For example, while type-checking this definition of the identity function here at the bottom right, we can smuggle the variable x out of its scope by storing it in the reference r.
3. pause

Part 2: Context-free type theories

Context-free type theories are...

- a reformulation of FTTs without explicit contexts
- translatable from and to FTTs
- the basis for implementation (Andromeda 2)

Problems with implementing finitary type theories:

- joining contexts (when forward chaining)
- strengthening fails (e.g. equality reflection)
- effectful metalanguage can cause scope extrusion

$$\frac{\Gamma_1 \vdash s : A \qquad \Gamma_2 \vdash t : A \qquad \Gamma_3 \vdash p : \mathrm{Eq}(A, s, t)}{??? \vdash s \equiv t : A}$$

$\lambda(x : A).\ r := x\,;\ x$

1. pause
2. pause
3. So how do we address these problems?

   Instead of storing the typing information in the context, each variable gets annotated with its type. The connection with the original calculus is established via translation theorems in both directions.

   Now, the main source of difficulty giving a context-free presentation in our setting comes from the kind of proof irrelevance exhibited for instance by the equality reflection rule and from the resulting failure of strengthening.

$$a : \mathbb{N} \vdash a + 0 : \mathbb{N} \quad \rightsquigarrow \quad a^{\mathbb{N}} + 0 : \mathbb{N}$$

1. No contexts
2. Every variable is tagged with a type

1. So, how do we do it? Consider the judgement at the top of the slide. We start by deleting the contexts from our judgements. The judgement "in context a of type nat, a plus zero has type nat" becomes instead "variable a *annotated* with the type nat plus zero has type nat".
   This solves two of our problems: we don't have to worry about combining contexts anymore, and if a variable escapes its scope, we can still make sense of it because it carries sufficient information.
2. pause
3. pause
4. pause

$$a : \mathbb{N} \vdash a + 0 : \mathbb{N} \quad \rightsquigarrow \quad \mathrm{a}^{\mathbb{N}} + 0 : \mathbb{N}$$

$$\frac{A \text{ type} \qquad s : A \qquad t : A \qquad p : \mathrm{Eq}(A, s, t)}{s \equiv t : A}$$

$$A \rightsquigarrow \mathrm{A}^{\square \text{ type}} \qquad s \rightsquigarrow \mathrm{s}^{\square : A} \qquad t \rightsquigarrow \mathrm{t}^{\square : A} \qquad p \rightsquigarrow \mathrm{p}^{\square : \mathrm{Eq}(A,s,t)}$$

1. No contexts
2. Every variable is tagged with a type
3. Every metavariable is tagged with the boundary of its judgement ("meta-type")

1. pause
2. A general definition of type theories must include a definition of rules, which means we need a formal theory of metavariables. The metavariables in this rule are A, s, t, and p. Just like we do with variables, we annotate each metavariable with its "meta-type".
3. pause
4. pause

$$a : \mathbb{N} \vdash a + 0 : \mathbb{N} \quad \rightsquigarrow \quad a^{\mathbb{N}} + 0 : \mathbb{N}$$

$$\frac{A \text{ type} \qquad s : A \qquad t : A \qquad p : \mathrm{Eq}(A, s, t)}{s \equiv t : A \text{ by } \{p\}}$$

$$A \rightsquigarrow A^{\square \text{ type}} \qquad s \rightsquigarrow s^{\square : A} \qquad t \rightsquigarrow t^{\square : A} \qquad p \rightsquigarrow p^{\square : \mathrm{Eq}(A,s,t)}$$

1. No contexts
2. Every variable is tagged with a type
3. Every metavariable is tagged with the boundary of its judgement ("meta-type")
4. Proof-irrelevant rules (e.g. equations) must record all variables in assumption sets

1. pause
2. pause
3. And finally, we address the failure of certain rules to record all of the variables that were used to derive their premises by introducing *assumption sets*. Instead of storing the entire term p, the context-free equality reflection rule traverses p and collects the variables it contains. This information is then stored in the conclusion as curly braces p.
4. pause

$$a : \mathbb{N} \vdash a + 0 : \mathbb{N} \quad \rightsquigarrow \quad a^{\mathbb{N}} + 0 : \mathbb{N}$$

$$\frac{A \text{ type} \qquad s : A \qquad t : A \qquad p : \mathrm{Eq}(A, s, t)}{s \equiv t : A \text{ by } \{\!|p|\!\}}$$

$$\frac{s : A \qquad A \equiv B \text{ by } \alpha}{\kappa(s, \beta) : B} \quad \text{with } \beta = \alpha \cup \{\!|A|\!\}$$

$$A \rightsquigarrow A^{\square \text{ type}} \qquad s \rightsquigarrow s^{\square : A} \qquad t \rightsquigarrow t^{\square : A} \qquad p \rightsquigarrow p^{\square : \mathrm{Eq}(A,s,t)}$$

1. No contexts
2. Every variable is tagged with a type
3. Every metavariable is tagged with the boundary of its judgement ("meta-type")
4. Proof-irrelevant rules (e.g. equations) must record all variables in assumption sets
5. Conversion records assumption sets

1. pause
2. pause
3. pause
4. This change to the structure of equality judgements means that we also have to change the conversion rule. We record the assumption set alpha used in the derivation of the equation A equals B and the variables used to derive A as the conversion term kappa s beta.

Theorem

There is a judgement-level translation from CFTT to FTT.

Proof idea: erase typing annotations and replace them with a suitable context.

$$\mathcal{J} \quad \mapsto \quad \Gamma_{\{\mathcal{J}\}} \vdash \lfloor \mathcal{J} \rfloor$$

$$\frac{\mathcal{D}_{CF}}{\mathcal{J}} \quad \Longrightarrow \quad \frac{\mathcal{D}_{CX}}{\Gamma_{\{\mathcal{J}\}} \vdash \lfloor \mathcal{J} \rfloor}$$

1. Let's connect contex-free and finitary type theories.
   Our first translation theorem tells us that we can safely use context free type theories to derive FTT judgements.

   It's important to note here that the *construction* of the concluding judgement *does not depend on derivability*. This matters for an implementation, because we want to be able to translate a CF judgement to an FTT judgement without storing its entire derivation. The theorem then *only* relies on the existence of a derivation to show that the translated judgement is *derivable* as well.
2. pause

Theorem

There is a judgement-level translation from CFTT to FTT.

Proof idea: erase typing annotations and replace them with a suitable context.

$$\mathcal{J} \quad \mapsto \quad \Gamma_{\{\mathcal{J}\}} \vdash \lfloor \mathcal{J} \rfloor$$

$$\frac{\mathcal{D}_{CF}}{\mathcal{J}} \quad \Longrightarrow \quad \frac{\mathcal{D}_{CX}}{\Gamma_{\{\mathcal{J}\}} \vdash \lfloor \mathcal{J} \rfloor}$$

Theorem

There is a derivation-level translation from FTT to CFTT.

Proof idea: induction on derivation, collecting assumptions.

$$\frac{\mathcal{D}_{CX}}{\Gamma \vdash \mathcal{J}} \quad \Longrightarrow \quad \frac{\mathcal{D}_{CF}}{\mathcal{J}^+}$$

$$\lfloor \mathcal{J}^+ \rfloor \; = \; \mathcal{J}$$

1. pause
2. We also have a completeness theorem: Any judgement derivable in a finitary type theory *with contexts* can be translated to a derivable judgement in the corresponding *context-free* theory.
3. This proof proceeds by induction on the derivation, and here instead the construction of the judgement *does* depend on the derivation.
4. Finally, let me say that these translations are *robust*. They work not only for standard type theories, but also for finitary theories. If we do start with a type theory satisfying certain good properties such as tightness, these properties are *preserved* by the translations. [TODO: cut this point?]

Part 3: Andromeda's metalanguage for CFTTs

An effectful programming language for user-defined context-free type theories.

- embed context-free type theories in a ML-style language

1. CFTT was designed with an effectful metalanguage in mind. In the third part of my thesis I introduce AML, the Andromeda meta-language. It embeds standard context-free type theories into a programming language in the tradition of Milner's ML.
2. pause
3. pause
4. pause
5. pause

Part 3: Andromeda's metalanguage for CFTTs

An effectful programming language for user-defined context-free type theories.

- embed context-free type theories in a ML-style language
- user definable rules and derived rules
- effective versions of metatheorems of standard context-free type theories

1. pause
2. In addition to smart constructors for the structural rules that all type theories include, AML allows the user to define their own rules. We also implement effective versions of the metatheorems of standard type theory. For example, inversion principles can be accessed via pattern matching.
3. pause
4. pause
5. pause

Part 3: Andromeda's metalanguage for CFTTs

An effectful programming language for user-defined context-free type theories.

- embed context-free type theories in a ML-style language
- user definable rules and derived rules
- effective versions of metatheorems of standard context-free type theories
- extend bidirectional typing to bidirectional evaluation
- operations & runners for proof development with local hypotheses

1. pause
2. pause
3. Besides embedding context-free type theories as a sublanguage, AML has two mechanisms for working with type theory.
4. Bidirectional evaluation is a generalisation of bidirectional typing. It helps the user by propagating typing information through a program.
5. AML also has effect operations and runners that can be used to work with local hypothesis, and also to implement known techniques from proof assistants such as universes or implicit arguments.
6. pause
7. pause

Part 3: Andromeda's metalanguage for CFTTs

An effectful programming language for user-defined context-free type theories.

- embed context-free type theories in a ML-style language
- user definable rules and derived rules
- effective versions of metatheorems of standard context-free type theories
- extend bidirectional typing to bidirectional evaluation
- operations & runners for proof development with local hypotheses
- implemented in the Andromeda 2 prover

1. pause
2. pause
3. pause
4. And, of course, AML is implemented in the Andromeda 2 prover.
5. pause

Part 3: Andromeda's metalanguage for CFTTs

An effectful programming language for user-defined context-free type theories.

- embed context-free type theories in a ML-style language
- user definable rules and derived rules
- effective versions of metatheorems of standard context-free type theories
- extend bidirectional typing to bidirectional evaluation
- operations & runners for proof development with local hypotheses
- implemented in the Andromeda 2 prover

Conjecture

AML is sound and complete for standard context-free type theories.

1. pause
2. pause
3. pause
4. pause
5. AML should be sound and complete for standard context free type theories. Completeness is relatively easy to see: we just translate a derivation to the corresponding AML rule applications. Soundness on the other hand is a bit more involved, as it has to deal with operations and runners, and completing this proof is still work in progress.

Part 3: Bidirectional evaluation in AML

- typing information flows between computations

$$c \rightsquigarrow r$$

$$c \mathbin{@} \mathcal{B} \checkmark r$$

1. I will say two words about bidirectional evaluation, but I don't want to kill you with ten pages of operational semantics.
2. The basic idea of bidirectional evaluation is to use contextual information and "push it up the derivation tree" as it were. This flow of information allows users to omit typing annotations when they are obvious from the context.
   We do this by splitting the evaluation relation into synthesising and checking. [TODO: explain the relations]
   This is particularly useful because standard context-free type theories require full annotations.
3. pause

- typing information flows between computations

$$c \rightsquigarrow r$$
$$c \mathbin{@} \mathcal{B} \checkmark r$$

- user-definable checking computations allow omission of annotations

$$\frac{c_b \mathbin{@} \{x{:}A\}\ \square : B \checkmark \{x{:}A\}\ b : B}{\texttt{lambda}(c_b) \mathbin{@} \Pi(A, \{x\}B) \checkmark \lambda(A, \{x\}B, \{x\}b) : \Pi(A, \{x\}B)}\ \text{Chk-Lambda-val}$$

1. pause
2. Let's look at an example. Here we have a rule for forming a lambda abstraction.
   If we required full annotations, we would have to specify both the domain and codomain types as well as the body, which is c_b here.
   AML allows us to define lambda as a computation that is run in checking mode, and use the typing information coming to us from the outside, by reading domain and codomain off the Pi-type we're checking against.

Part 3: Operations and runners in AML

- operations: an escape hatch allowing the use of local hypotheses
- runner: can use the information to perform user-specified computations

```
let r = runner
  | coerce-to (J, bdry) →
    match (J, bdry) with
    | ((_ : ?A), (□ : ?B)) → let e = eqchk A B in convert J e
    | _ → coerce-to (J, bdry)
in with r run comp
```


1. Finally, I'll say a word about operations and runners.
2. Operations provide a sort of "escape hatch", which allows users to locally modify the behaviour of type theoretic algorithms such as an equality checker.
   An operation triggered in a program has access to local variables as well as the current checking boundary.
   Once an operation is triggered, it bubbles up through the program like a resumable exception, until it encounters a matching runner.
   Runners are pieces of user specified code that can use the local information to provide extensions to standard algorithms.
3. pause
4. pause

Part 3: Operations and runners in AML

- operations: an escape hatch allowing the use of local hypotheses
- runner: can use the information to perform user-specified computations

```
let r = runner
  | coerce-to (J, bdry) →
    match (J, bdry) with
    | ((_ : ?A), (□ : ?B)) → let e = eqchk A B in convert J e
    | _ → coerce-to (J, bdry)
in with r run comp
```


1. pause
2. Here we have an example of a runner that handles conversion of the judgement J to the boundary bdry. This could be part of a standard library. The runner calls the eqchk algorithm to produce evidence that A equals B.
3. pause

Part 3: Operations and runners in AML

- operations: an escape hatch allowing the use of local hypotheses
- runner: can use the information to perform user-specified computations

```
let r = runner
  | coerce-to (J, bdry) →
    match (J, bdry) with
    | ((_ : ?A), (□ : ?B)) → let e = eqchk A B in convert J e
    | _ → coerce-to (J, bdry)
in with r run comp
```


1. pause
2. pause
3. Bidirectional evaluation interacts nicely with operations: if a type theoretic computation evaluates to a judgement that does not have boundary that the contexts expects, a *coerce* operation is triggered.
   A user-installed runner can be used to rectify the problem, for example by decoding a term in a universe to a type.

Conclusion

Related & future work:

- update the implementation
- formalise examples!
- more effects: runners are overly restrictive
- further analysis of bidirectionality
- generalise to other judgement forms (cubical)
- modal type theories?

1. Much related work exists. Valery Isaev and especially Taichi Uemura have recently given general definitions of type theories.

Conclusion

Related & future work:

- update the implementation
- formalise examples!
- more effects: runners are overly restrictive
- further analysis of bidirectionality
- generalise to other judgement forms (cubical)
- modal type theories?

1. Much related work exists. Valery Isaev and especially Taichi Uemura have recently given general definitions of type theories.
2. When it comes to future work, there are some obvious next steps.
3. Much of the analysis of bidirectional evaluation has not yet been implemented, and should be added to Andromeda.
4. We should experiment and put Andromeda to work. For example, we could formalise the construction of semi-simplicial types in HTS.
5. Bidirectional evaluation allows us to reuse some of the information when we apply type theoretic rules, but no rule-specific analysis is performed, and all rules are synthesising. In the work on equality checking for finitary type theories with Anja Petković and Andrej Bauer we analyse rules and define a general notion of computation rule.
6. Building on work by Pfenning, Dunfield, and Krishnaswami, A similar analysis might allow us to exploit bidirectional typing information in a more rule-specific way.