Programming language techniques for proof assistants

Lecture 1
From declarative to algorithmic type theory

Andrej Bauer
University of Ljubljana

International School on Logical Frameworks and Proof Systems Interoperability
Université Paris–Saclay, September 8–12, 2025

Overview

- Lecture 1: From declarative to algorithmic type theory
  - A declarative presentation of type theory
  - Bidirectional type checking
  - Algorithmic equality checking
- Lecture 2: A monadic type checker
  - Parsing, bound variables and substitution
  - A monad for typing contexts
  - A monadic proof checker
- Lecture 3: Holes and unification
  - Postponed computations as holes
  - Unification
  - A holey type checker
- Lecture 4: Variables as computational effects
  - Algebraic operations and handlers
  - Holes as computational effects
  - A handler-based type checker

Lecture 1

From declarative to algorithmic type theory

What is a type theory?

- A collection of inference rules
- Quotient inductive inductive type (QIIT)
- Second-order generalized algebraic theory (SOGAT)
- A small representable map category

1. There are many views of what a type theory is, ranging syntactic to semantic ones. Each view enriches our understanding of type theory and has its uses.
2. For implementation purposes, *syntactic presentations* are a good starting point.

Faux Type Theory – syntax & judgement forms

Expressions:

$t, u, A, B ::=$	x	variable	
	$\mid \mathsf{Type}$	universe	
	$\mid \Pi_{(x:A)}\, B$	product	(x bound in B)
	$\mid t\, u$	application	
	$\mid \lambda(x:A).\, t$	function	(x bound in t)
	$\mid \mathsf{let}\, x := t\, \mathsf{in}\, u$	local definition	(x bound in u)

Judgement forms:

$\Gamma \vdash t : A$	"In context Γ, term t has type A."
$\Gamma \vdash t \equiv_A u$	"In context Γ, terms t and u of type A are equal."

A context Γ maps variables to their types and definitions, if any:

$x : A$	"x has type A"
$x := t : A$	"x equals t and has type A"

1. We're going to work with a small type theory, called "faux type theory", that only has some basic features: a universe, dependent products, and local definitions.
2. Textbook presentations of type theory often omit local definitions, since they can always be eliminated by substitution. However, working without them in practice would be cumbersome, so we include them to illustrate their implementation.
3. The first judgement form asserts that a term t has a type A in context Γ.
4. The second judgement form asserts terms t and u have type A and are equal. This is *typed* equality. (Untyped equality would read just $\Gamma \vdash t = u$ without any type information. In principle, it would allow us to compare terms of different types.)
5. A context Γ holds information about variables. It always specifies the type of a variable, and optionally its definition (to be used in combination with local definitions).
   We are not very precise about what a context really looks like here. Is it a list? A dictionary? It does not matter much how it is represented, so long as it provides information about variables.

Caveat about local definitions

Isn't "$\mathsf{let}\, x := u \,\mathsf{in}\, t$" the same as "$(\lambda(x : A).\, t)\, u$"?

No!

- When checking the type of t in $\mathsf{let}\, x := u \,\mathsf{in}\, t$ we know that $x \equiv u$.
- When checking the type of t in $\lambda(x : A).\, t$ we only know that $x : A$.

Exercise: Find $u : A$ and $\mathsf{let}\, x := u \,\mathsf{in}\, t$ that type-checks but $(\lambda(x : A).\, t)\, u$ does not.

1. I have seen people fall victim, myself included, to the following fallacy.
2. Can't we get rid of local definitions by replacing them with function applications?
3. No! If we go back to the typing rules we see that in one case the information $x \equiv u$ is available and in the other not.

Faux Type Theory – terms and types

$$\textsc{Tm-Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textsc{Ty-Type}\ \frac{}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}} \qquad \textsc{Ty-Pi}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi_{(x:A)}\, B : \mathsf{Type}}$$

$$\textsc{Tm-App}\ \frac{\Gamma \vdash t : \Pi_{(x:A)}\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]} \qquad \textsc{Tm-Lambda}\ \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\, t : \Pi_{(x:A)}\, B}$$

$$\textsc{Tm-Let}\ \frac{\Gamma \vdash t : A \qquad \Gamma, x := t : A \vdash u : B}{\Gamma \vdash (\mathsf{let}\, x := t\, \mathsf{in}\, u) : B[t/x]}$$

1. Next, let us look at the rules of Faux Type Theory. The presentation given here is of the sort one usually finds in books and papers, namely as a collection of *rules*.
2. We are assuming you have seen such rules before. Briefly, a rule has a number of *premises*, written above the line, and a *conclusion* below the line. We give each rule a name, written in light gray.
3. In the variable rule, the top line should be read as "According to Γ, x has type A." In particular, $(x : A) \in \Gamma$ holds also when Γ stores a definition $x := t : A$.
4. Our type theory claims that Type is a Type. This is one of the reasons for calling it "faux", because such a type theory is inconsistent in the sense that every type is inhabited. (Exercise: write down a term of type $\Pi_{(A\,:\,\mathsf{Type})}\, A$.) Nevertheless, such theories can still be useful as models of (partial) computation, and in fact our theory is Turing complete.
5. An alternative to $\mathsf{Type} : \mathsf{Type}$ would be the introduction of a hierarchy of universes $\mathsf{Type}_0 : \mathsf{Type}_1 : \cdots$, or perhaps just $\mathsf{Type} : \mathsf{Kind}$. In the implementation, doing so would bring in additional complexity that we prefer to avoid. (Exercise: remove $\mathsf{Type} : \mathsf{Type}$.)
6. Observe how the rule Tm-Let stores a local definition in the context.

$$\textsc{Tm-App}\quad \frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash B : \mathsf{Type} \qquad \Gamma \vdash t : \Pi_{(x:A)}\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B[u/x]}$$

$$\textsc{Tm-Lambda}\quad \frac{\Gamma, x : A \vdash B : \mathsf{Type} \qquad \Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).\, t : \Pi_{(x:A)}\, B}$$

- **Presupposition theorem:**
  - if $\Gamma \vdash t : A$ then $\Gamma \vdash A : \mathsf{Type}$,
  - if $\Gamma \vdash t \equiv_A u$ then $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$.
- **Uniqueness of typing:**
  - if $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ then $\Gamma \vdash A \equiv_{\mathsf{Type}} B$.
- **Weakening:**
  - if $\Gamma \vdash t : A$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash t : A$.
- **Strengthening:**
  - if $\Delta \vdash t : A$ and $\Gamma \subseteq \Delta$ and $\mathsf{FV}(t : A) \subseteq |\Gamma|$ then $\Gamma \vdash t : A$.

1. Let us stop here to comment on how the rules are written. Let us compare the rules for application and functions. Why does Tm-Lambda check that A is a type but Tm-App does not? In fact, why are we not checking that B is a type in context $\Gamma, x : A$?
2. If one were very pedantic, one would add additional premises, shown in blue. These are called *presuppositions* – they ensure that the contexts and types appearing in other judgements are well-formed.
3. It turns out that the presuppositions are redundant, because the rules on the previous slide satisfy the *Presuppositions theorem*, which says that the constituent parts of a derivable judgement are well-formed. (Note: we always assume that Γ does not contain any "garbage". Exercise: make this idea precise and prove the Presuppositions theorem.)
4. In implementation we prefer the *economic* rules that omit presuppositions, so long as the Presuppositions theorem is valid. Concretely, in Tm-App this allows us to skip re-checking that A and B are well-formed types (apply the Presuppositions theorem to the premises).
5. There are several other desirable meta-theorems that we shall rely on in the implementation. The theorems as stated here are imprecise (what does $\Gamma \subseteq \Delta$ mean?), see background material for exact versions.

Faux Type Theory – equality

Eq-Refl
$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv_A t}$$

Eq-Sym
$$\frac{\Gamma \vdash u \equiv_A t}{\Gamma \vdash t \equiv_A u}$$

Eq-Tran
$$\frac{\Gamma \vdash s \equiv_A t \qquad \Gamma \vdash t \equiv_A u}{\Gamma \vdash s \equiv_A u}$$

Eq-Conv
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv_{\mathsf{Type}} B}{\Gamma \vdash t : B}$$

Eq-Def
$$\frac{(x := t : A) \in \Gamma}{\Gamma \vdash x \equiv_A t}$$

Eq-Pi-Ext
$$\frac{\Gamma, x : A \vdash t\,x \equiv_B u\,x : B}{\Gamma \vdash t \equiv_{\Pi_{(x:A)}\,B} u}$$

Eq-Comp
$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda(x : A).\, t)\, u \equiv_{B[u/x]} t[u/x]}$$

Eq-Let
$$\frac{\Gamma \vdash t : A \qquad \Gamma, x := t : A \vdash u : B}{\Gamma \vdash (\mathsf{let}\, x := t\, \mathsf{in}\, u) \equiv_{B[t/x]} u[t/x]}$$

.

1. Let us proceed with equality rules. They come in several batches.
2. The first one states that equality is an equivalence relation.
3. The conversion rule Eq-Conv states that a type may be replaced with an equal one.
4. Eq-Def states that a variable is equal to its definition. This is sometimes called a "δ-rule".
5. The most interesting are the rules for application and functions. Eq-Comp is also known as "β-rule" and is a *computation rule*. In the implementation it will be used to compute normal forms of terms. It has a clear left-to-right direction: the left-hand side serves as a *pattern* against which we can match a term, if possible, and rewrite it to the right-hand-side.
6. Eq-Ext is an *extensionality rule*. It does not have a direction, but we can tell that it should be used when we compare terms at a product type. (Exercise: show that in the presence of other rules, the extensionality rule is bi-direvable form the η-rule for function.)
7. Finally, there is a rule for local definitions which explains that a definition may be eliminated by substitution.
8. We now have all the rules of Faux type theory. Do they indicate how to implement a proof checker? Not quite. It is not clear which rule to use in a given situation. For example, conversion Eq-Conv can *always* be applied, as well as reflexivity, symmetry and transitivity. We need to do something about this.

Bidirectional type checking

Replace $\Gamma \vdash t : A$ with two judgements forms:

$\Gamma \vdash t \rightsquigarrow A$	"In context Γ, infer that term t has type A."
$\Gamma \vdash t \leftsquigarrow A$	"In context Γ, check that term t has type A."

Desiderata:

- The rules should be *deterministic* and *syntax-driven*:
  - at most one rule applies in any situation, and
  - the candidate rule is apparent from the syntax of t.
- **Soundness** of bidirectional type-checking:
  - if $\Gamma \vdash t \rightsquigarrow A$ then $\Gamma \vdash t : A$, and
  - if $\Gamma \vdash t \leftsquigarrow A$ then $\Gamma \vdash t : A$.
- **Completeness** of bidirectional type-checking:
  - if $\Gamma \vdash t : A$ then $\Gamma \vdash t \leftsquigarrow A$.

1. If you look at the rules, some of allow us to *infer* the type by looking at the term and the context (Tm-Var, Ty-Type, Ty-Pi, Tm-Lambda). In some situations, the type is given and we must *check* that the term has it (both premises of Ty-Pi, second premise of Tm-App).
2. This gives us the idea to split the judgement $\Gamma \vdash t : A$ into two judgements.
3. $\Gamma \vdash t \rightsquigarrow A$ *infers* (computes, synthesizes) the type A of t.
4. $\Gamma \vdash t \leftsquigarrow A$ *checks* that t has the given type A.
5. The notation is suggestive:

   $t \rightsquigarrow A$ indicates that t *produces* the type A

   $t \leftsquigarrow A$ indicates that t *consumes* the type A
6. We aim to satisfy certain requirements.
7. The bidirectional rules should be deterministic so that they can serve as the basis of an efficient algorithm (that does not have to perform excessive amounts of backtracking).
8. The bidirectional rules should be syntax-driven, i.e., which rule to use should be clear from the syntax of expressions.
9. We require soundness and completeness to ensure that the declarative and bidirectional rules present the same type theory.

Bidirectional Faux type theory – rules I

$$\textsc{Infer-Var}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x \rightsquigarrow A} \qquad \textsc{Infer-Type}\ \frac{}{\Gamma \vdash \mathsf{Type} \rightsquigarrow \mathsf{Type}} \qquad \textsc{Infer-Pi}\ \frac{\Gamma \vdash A \lessdot\!\sim \mathsf{Type} \qquad \Gamma, x : A \vdash B \lessdot\!\sim \mathsf{Type}}{\Gamma \vdash \Pi_{(x:A)}\, B \rightsquigarrow \mathsf{Type}}$$

$$\textsc{Infer-App}\ \frac{\Gamma \vdash t \rightsquigarrow \Pi_{(x:A)}\, B \qquad \Gamma \vdash u \lessdot\!\sim A}{\Gamma \vdash t\, u \rightsquigarrow B[u/x]} \qquad \textsc{Infer-Lambda}\ \frac{\Gamma \vdash A \lessdot\!\sim \mathsf{Type} \qquad \Gamma, x : A \vdash t \rightsquigarrow B}{\Gamma \vdash \lambda(x : A).\, t \rightsquigarrow \Pi_{(x:A)}\, B}$$

1. We already observed that some rules want to be inferring.
2. Notice how in some of the premises we switch to checking rules. There is going to be an interplay between the two kinds of rules.

Bidirectional Faux type theory – rules II

Let-binding is direction-agnostic:

$$\textsc{Infer-Let}\quad \frac{\Gamma \vdash t \rightsquigarrow A \qquad \Gamma, x := t : A \vdash u \rightsquigarrow B}{\Gamma \vdash (\mathsf{let}\, x := t\, \mathsf{in}\, u) \rightsquigarrow B[t/x]}$$

$$\textsc{Check-Let}\quad \frac{\Gamma \vdash t \rightsquigarrow A \qquad \Gamma, x := t : A \vdash u \leftsquigarrow B}{\Gamma \vdash (\mathsf{let}\, x := t\, \mathsf{in}\, u) \leftsquigarrow B}$$

There is a rule for switching from checking to inference:

$$\textsc{Infer-Check}\quad \frac{\Gamma \vdash t \rightsquigarrow A \qquad \Gamma \vdash A \equiv_{\mathsf{Type}} B}{\Gamma \vdash t \leftsquigarrow B}$$

Note: this is the *only* rule in which equality applies.

1. Local definitions are agnostic, the inferring or checking simply passes through them.
2. The rule Infer-Check serves as an interface between the checking and inferring mode. It is used when we are asked to check a term that wants its type to be inferred. This is the only rule that mentions equality. In contrast to Eq-Conv, it can be used in only one situation: when we're trying to check an inferring term.

Algorithmic equality checking

Algorithmic checking of $\Gamma \vdash t \equiv_A u$:

- *type-directed phase*: apply extensionality rules for A, if any,
- *normalization phase*: normalize t and u and compare the normal forms.

Judgement forms:

$\Gamma \vdash t \hookrightarrow u$	"term t normalizes to u"
$\Gamma \vdash t \simeq_A u$	"terms t and u of type A are equal"
$\Gamma \vdash t \overset{\text{nf}}{\cong} u$	"normal terms t and u are equal"

1. Next, we proceed with turning the equality rules into algorithmic form.
2. We are going to use an equality-checking algorithm that has two phases. (Many implementations have only the normalization phase, and so they incorporate η-rules instead of extensionality rules.)
3. Now there are three auxiliary judgement forms: normalization (also called conversion and computation), typed equality of general terms, and untyped equality of normalized terms.
4. Exercise: state soundness and completeness of algorithmic type-checking with respect to declarative judgemental equality.

Normalization

NORM-TYPE

$$\frac{}{\Gamma \vdash \mathsf{Type} \hookrightarrow \mathsf{Type}}$$

NORM-PI

$$\frac{}{\Gamma \vdash \Pi_{(x:A)}\, B \hookrightarrow \Pi_{(x:A)}\, B}$$

NORM-LAMBDA

$$\frac{}{\Gamma \vdash \lambda(x:A).\, t \hookrightarrow \lambda(x:A).\, t}$$

NORM-VAR-NEUT

$$\frac{(x:A) \in \Gamma \qquad (x := \_ : \_) \notin \Gamma}{\Gamma \vdash x \hookrightarrow x}$$

NORM-VAR-DEF

$$\frac{(x := t : A) \in \Gamma \qquad \Gamma \vdash t \hookrightarrow u}{\Gamma \vdash x \hookrightarrow u}$$

NORM-LET

$$\frac{\Gamma \vdash t[u/x] \hookrightarrow v}{\Gamma \vdash (\mathsf{let}\, x := u\, \mathsf{in}\, t) \hookrightarrow v}$$

NORM-APP-COMP

$$\frac{\Gamma \vdash t \hookrightarrow \lambda(x:A).\, t' \qquad \Gamma \vdash t'[u/x] \hookrightarrow v}{\Gamma \vdash t\, u \hookrightarrow v}$$

NORM-APP-NEUT

$$\frac{\Gamma \vdash t \hookrightarrow t' \neq \lambda(\_:\_).\, \_}{\Gamma \vdash t\, u \hookrightarrow t'\, u}$$

1. Let us first deal with normalization. In programming language theory we would call such rules *big-step operational semantics* and the normal forms *values*.
2. The universe, products, and functions are normal, and so is a variable without a definition.
3. Variables with definitions and local definitions always make a normalization step. Application makes a step when a function is applied.
4. These rules employ a *lazy* evaluation strategy: in NORM-LET and NORM-APP-COMP the term u is substituted directly, without being evaluated first. This is a source of possible inefficiency: if u is duplicated it may be normalized twice. Haskell-style lazy languages avoid such problems by employing the *call-by-need* strategy.
5. What do the normal expressions (outputs of $\hookrightarrow$) look like?

Weak head-normal forms

Normal expression:

$n ::=$	x	variable
$\mid$	Type	universe
$\mid$	$\Pi_{(x:A)}\, B$	product
$\mid$	$\lambda(x:A).\, t$	function
$\mid$	$x\, t_1 \cdots t_n$	spine

Desideratum:

- **Preservation Lemma:**
  - If $\Gamma \vdash t : A$ and $\Gamma \vdash t \hookrightarrow u$ then $\Gamma \vdash u : A$.
- **Progress Lemma:**
  - If $\Gamma \vdash t : A$ then there exists (a unique) normal u such that $\Gamma \vdash t \hookrightarrow u$.

1. Normalization outputs one of the normal expressions.
2. Not every normal expression makes sense, for instance $\mathsf{Type}\, x\, y$. (Exercise: prove that $\mathsf{Type}\, x\, y$ does not have a type.) However, if we normalize an expression that does have a type, we should get a normal one that has the same type.
3. The stated property is akin to the so-called Preservation lemma from programming language theory, which states that evaluation preserves typing.
4. Preservation is accompanied by the Progress lemma, which states that evaluation does not get stuck. Together they form the Safety Theorem, “If a term has a type then it normalizes.” In programming language terminology it says that type-checked programs are safe to run (they do not crash). In type theory theorems like this go under the name “subject reduction”.

Eq-Ext-Pi

$$\frac{\Gamma \vdash A \hookrightarrow \Pi_{(x:B)}\, C \qquad \Gamma, x : B \vdash t\, x \simeq_C u\, x}{\Gamma \vdash t \simeq_A u}$$

Eq-Ext-Eq-Nf

$$\frac{\Gamma \vdash A \hookrightarrow \_ \neq \Pi_{(\_)}\, \_ \qquad \Gamma \vdash t \hookrightarrow t' \qquad \Gamma \vdash u \hookrightarrow u' \qquad \Gamma \vdash t' \overset{\text{nf}}{\simeq} u'}{\Gamma \vdash t \simeq_A u}$$

Eq-Nf-Type

$$\frac{}{\Gamma \vdash \mathsf{Type} \overset{\text{nf}}{\simeq} \mathsf{Type}}$$

Eq-Nf-Pi

$$\frac{\Gamma \vdash A \simeq_{\mathsf{Type}} C \qquad \Gamma, x : A \vdash B \simeq_{\mathsf{Type}} D}{\Gamma \vdash \Pi_{(x:A)}\, B \overset{\text{nf}}{\simeq} \Pi_{(x:C)}\, D}$$

Eq-Nf-Spine

$$\frac{\begin{array}{c}(x : A) \in \Gamma \\ \Gamma \vdash A \hookrightarrow \Pi_{(x_1:A_1)} \cdots \Pi_{(x_n:A_n)}\, B \\ \Gamma \vdash t_i \simeq_{A_i[t_1/x_1,\ldots,t_{i-1}/x_{i-1}]} u_i \quad \text{for } i = 1, \ldots, n\end{array}}{\Gamma \vdash x\, t_1 \cdots t_n \overset{\text{nf}}{\simeq} x\, u_1 \cdots u_n}$$

1. Here finally are the equality rules. The typed equality is the "extensionality" phase. It applies the extensionality rule Eq-Pi-Ext if possible, otherwise it proceeds to the normalization phase.
2. The normalization phase compares terms structurally. Note that the premises refer back to typed equality. It is important that it be able to reconstruct the types at which to compare the sub-terms.

Exercises

1. Find someone to work with.
2. Write down the rules for the unit type.
3. Write down the rules for dependent sums.
4. Install the implementation of Faux Type Theory from `https://github.com/andrejbauer/faux-type-theory`.
5. Use the implementation to derive $\Pi_{(A\,:\,\mathsf{Type})}\, A$.

1. Here are some exercises for you to work on. Find a partner or a small group and do them together. If you do not know anyone, ask the person sitting next to you if they'd like to work together.
2. When writing down the rules, do not forget about equality rules, and write down both the declarative and bidirectional versions.
3. I am of course available for any questions you might have, in person and on Zulip.
4. Please show me your solutions—that's important feedback for me.

Background & further reading

Software:

- Andrej Bauer: `https://github.com/andrejbauer/faux-type-theory`
- András Kovács: `https://github.com/AndrasKovacs/elaboration-zoo`
- Christophe Raffalli, Rodolphe Lepigre et al.: Bindlib library.

Papers:

- Christopher A. Stone, Robert Harper: Extensional equivalence and singleton types, ACM Trans. Comput. Log. 7(4): 676-722 (2006)
- Philipp G. Haselwarter: Effective Metatheory for Type Theory, PhD thesis, University of Ljubljana, 2021.
- Anja Petković Komel: Meta-analysis of type theories with an application to the design of formal proofs, PhD thesis, University of Ljubljana, 2021.
- Andrej Bauer, Anja Petković Komel: An extensible equality checking algorithm for dependent type theories, Logical Methods in Computer Science, January 19, 2022, Volume 18, Issue 1.
- Philipp G. Haselwarter, Andrej Bauer: Finitary Type Theories With and Without Contexts, Journal of Automated Reasoning 67, 36 (2023).