Programming language techniques for proof assistants

Lecture 2
A monadic type checker

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
- 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
- Lecture 4: Variables as computational effects

1. In today's lecture we're going to implement Faux Type Theory. Our goal is a small reference implementation that showcases technique. At the same time, we do want it to be reasonably useful and reasonably fast.
2. We have no desire to implement everything from scratch. We will use parsing tools and other libraries when appropriate. Our focus is on transcribing the core type-checking and equality-checking algorithms in a principled manner.

Lecture 2

A monadic type checker

Components of a proof checker

- Parser:
  - lexer: split the input string into a sequence of tokens
  - parser: convert a sequence of tokens into a syntax tree
- Core:
  - inference & checking modes
  - normalization
  - equality
- Top level:
  - Command-line arguments
  - Top level commands
  - REPL (Read–Evaluate–Print–Loop)

1. Our proof checker has three main parts: a parser, a type checker, and a top level. We shall discuss each component separately.
2. To be quite honest, this view is a bit outdated. We should also have a language server protocol (LSP) for interaction with an editor.

Data types

- Input strings
- Tokens
- Input syntax trees:
  - record file locations (for error reporting)
  - variables are strings
- Internal terms and types:
  - no locations
  - support bound variables and substitution
- Context:
  - map concrete variable names (strings) to abstract variables (`Bindlib.var`)
  - map abstract variables to their types and optional definitions

1. We can frame the task also in terms of data types that will drive the development.
2. Parsing takes an input string, breaks it up into tokens, and produces a concrete syntax tree. They're "concrete" in the sense that the variable names are still just strings, and that they tree faithfully captures the concrete input syntax.
   Strings are predefined, token will be managed by the parser generator tools, but we do have to define our own syntax trees. We want such trees to store file locations so that error reporting can tell the user where the error is.
3. The type checker will convert input syntax trees to internal terms and types. These are different from the input syntax trees:
   - they do not record locations: they have already been checked ans so cannot be the source of an error
   - they have to properly handle bound variables and support substitution
   - they capture the underlying type theory, not user input
4. We also need a datatype representing a typing context. We shall split it into maps. The first one is used to map variables as strings to abstract variables as represented by `Bindlib` (a home-made implementation might map then to de Bruijn indices). The second one maps abstract variables to their types and optional definitions.
5. There may be other auxiliary datatypes of course, for example, one that represents top-level commands.

Implementation language – OCaml

- Why OCaml?
  - We are in France.
  - Rocq, Dedukti, Lambdapi, etc., are implemented in OCaml.
- Developer environment:
  - OPAM – OCaml Package Manager
  - Dune – a composable build system for OCaml
- Libraries:
  - `sedlex` – Unicode-friendly lexer generator
  - `menhir` – LR(1) parser generator
  - `bindlib` – manipulation of data structures with bound variables

1. We shall implement the proof checker in OCaml.
2. Of course, we could use a number of other languages, but OCaml is certainly a good choice. Real proof assistants are implemented in it. It has good development environment, and there are libraries that will take care of standard tasks, such as parsing and manipulation of bound variables.

Parsing

- Lexical analysis:
  - Break up the input string into *lexemes* (substrings) and map them to *tokens*.
  - `lib/parsing/lexer.ml` – lexemes are specified by regular expressions
  - `lib/parsing/parser.mly` – tokens are specified at the beginning
- Parsing:
  - `lib/parsing/parser.mly` – the grammar generates syntax trees (with file locations)
  - `lib/parsing/syntax.ml` – the type of syntax trees:

```
type tm = tm' Location.t
and tm' =
  | Var of string
  | Let of string * tm * tm
  | Type
  | Prod of (string * ty) * ty
  | Lambda of (string * ty option) * tm
  | Apply of tm * tm
  | Ascribe of tm * ty
and ty = tm
```


1. Let us proceed to the first part, which is parsing. This is to be considered a "solved problem".
2. Briefly, the task is to convert an input string to a syntax tree, which is done in two phases.
3. Lexical analysis breaks up the string into separate *lexemes*, small pieces of text such as keywords, punctuation marks, and numerals. Lexemes are mapped to *tokens*, which are just a more abstract representation of lexemes.
4. Parsing takes a sequence of lexemes and converts them to a syntax tree, following the rules of a formal grammar.
5. We are not going to do all this by hand (although I recommend that you do it once, and then never again), but rather use the Sedlex lexer and Menhir parser generator. We do not have the time to go into the technical details here, but feel free to ask me about them later.
6. For us, the interesting part is the datatype of input syntax trees, shown here. Notes:
   - The datatype of input terms tm is wrapped by a utility datatype `Location.t` which stores file locations. The parser adds the locations during parsing.
   - There is no difference between terms tm and types ty, as there are no syntactic markers that could be used to tell them apart.
   - Variables are strings, directly provided by the parser.
   - As a convenience for the user, we allow λ-abstractions without type annotations. The target type theory still has typed λ-abstractions, so the type checker will have to supply missing types.
   - We added *type ascription*, which allows the user to specify a type. Its effect is to switch to checking mode. (Exercise: write down the rules of type ascription.)

Internal terms and types

- lib/core/TT.ml:

```
type tm =
  | Var of var
  | Let of tm * ty * tm binder
  | Type
  | Prod of ty * ty binder
  | Lambda of ty * tm binder
  | Apply of tm * tm

and ty = Ty of tm

and var = tm Bindlib.var

and 'a binder = (tm, 'a) Bindlib.binder
```


1. The internal terms and types will be produced by the type checker. They are similar to the input syntax trees, but now the same.
2. There are no file locations anymore. We have no use for them, because checked terms and types cannot be the source of an error. The user is the only source of errors.
3. Note that local definitions store the type of the local variable. (Exercise: find out where this type is used in the code, and you will understand why we are recording it.)
4. We use Bindlib's variables. We do *not* want to know how they work, only that they work and that they deal with the intricacies of bound variables and substitution. One does have to read the Bindlib documentation to learn how to use it, but this is time well spent. I guarantee you that it takes less time to learn how to use Bindlib than it does to debug a home-made implementation of de Bruijn indices. I know, I tried several times.
5. If a term or a type binds a variable, it should be wrapped by `Bindlib.binder`.
6. We make terms `tm` and types `ty` formally different by wrapping the latter in a `Ty` constructor. This is a bit of a shortcut, because not every term can be a type (consider `Lambda`), but in view of `Type : Type` it is convenient to have an easy way of converting terms to types and vice versa.

Contexts – lib/core/context.ml

```
module IdentMap = Map.Make(struct
                             type t = string
                             let compare = String.compare
                           end)

module VarMap = Map.Make(struct
                           type t = TT.var
                           let compare = Bindlib.compare_vars
                         end)

type t =
  { idents : TT.var IdentMap.t ;
    vars : (TT.tm option * TT.ty) VarMap.t }
```


1. The third main datatype is that of contexts.
2. We use OCaml's map datatype to represent the two components of a context.
3. The idents field maps identifiers (strings) to variables.
4. The vars field maps variables to information associated with them.

Core functionality – `lib/core/typecheck.ml`

$$\texttt{infer} : \texttt{Syntax.tm} \to \texttt{Context.t} \to \texttt{TT.tm} \times \texttt{TT.ty}$$

$$\texttt{check} : \texttt{Syntax.tm} \to \texttt{TT.ty} \to \texttt{Context.t} \to \texttt{TT.tm}$$

1. Let us now consider the core functionality, namely type inference and checking. These are represented by two functions, as shown.
2. Inference takes an input term and a context, and outputs a checker terms and its type.
3. Checking takes an input term, a *checked* type and a context, and outputs a checked term.
4. Keep in mind that the typing information does not reveal an important fact: errors can happen. These will be implemented as exceptions. (Not trying to start a religious war, but in Haskell the presence of errors would be visible because Haskell *simulates* exceptions as sums.)
5. You might find it odd that context is passed as the last argument, rather than the first. After all, we write $\Gamma \vdash t : A$ and not $t : A \dashv \Gamma$. There is a good reason for this.
6. When we implement these functions, we will end up passing the context into recursive calls. This is the kind of boilerplate code that we want to avoid, which we can by using a bit of functional programming techniques.

A monad for contexts – lib/core/context.ml

```
type 'a m = Context.t -> 'a

val infer : Syntax.tm -> (TT.tm * TT.ty) m
val check : Syntax.tm -> TT.ty -> TT.tm m


module Monad =
struct
  let ( let* ) c1 c2 ctx =
    let v1 = c1 ctx in
    c2 v1 ctx

  let ( >>= ) = ( let* )

  let return v ctx = v
end
```


1. We define a datatype (constructor) m as shown and wrap the output in it.
2. Now, m is a monad, the reader monad to be precise. (Not trying to start a religious war, but Haskellers told you so.)
3. The OCaml let $*$ notation is the equivalent of Haskell's do for monadic computations.
4. If you do not know about monads, consider this a programming trick that allows us to use OCaml let $*$ syntax to pass along contexts implicitly.

The core

- `lib/core/context.ml` (79 lines)
- `lib/core/typecheck.ml` (200 lines)
  - What's up with TT.tm_ and Typecheck.infer_ and other underscores?
- `lib/core/norm.ml` (97 lines)
  - We implement both the weak-head normal forms and call-by-value evaluation.
- `lib/core/equal.ml` (91 lines)

1. At this point, it is best to just read the code. There isn't that much of it.
2. When reading one of the files, you should first read the .mli file, which specifies the interface, and then .ml file, which contains the implementation.
3. Here we're going to take a peek, just to see if the code resembles the bidirectional rules and the two-phase equality checking from Lecture 1.
4. The code seems to have two versions of everything: TT.tm and TT.tm_, TT.infer and TT.check_? This is a consequence of how Bindlib works.
5. Briefly, in order Bindlib for to do its magic, it wants to participate in the construction of terms by equipping them with extra information about variables. This is called *boxing*, see Bindlib.box. The actual terms is obtained from a boxed one using Bindlib.unbox.
6. Convention: the underscore indicates that we're working with the boxed version of a datatype.

Putting it all together

- Top level – lib/core/toplevel.ml

`load "⟨file⟩.ftt"`	load a file
`axiom` $c : t$	declare an undefined constant c of type t
`def` $x := e$	define x to be e
`def` $x : t := e$	define x of type t to be e
`infer` e	infer the type of e
`eval` e	evaluate e

- Main executable – bin/fauxtt.ml

1. Finally, we put it all together by implementing a top level. From a conceptual point of view there is nothing inspiring here, but it may be useful to see concretely how it's all composed.

Example: Church numerals

```
def numeral :=
  Π (A : Type), (A → A) → (A → A)
def zero : numeral :=
  λ (A : Type) (f : A → A) (x : A) ⇒ x
def succ : numeral → numeral :=
  λ n A (f : A → A) (x : A) ⇒ f (n A f x)
def ( + ) : numeral → numeral → numeral :=
  λ m n A (f : A → A) (x : A) ⇒ m A f (n A f x)
def ( * ) : numeral → numeral → numeral :=
  λ m n A (f : A → A) (x : A) ⇒ m A (n A f) x

(* A trick to see the numerals *)
axiom N : Type
axiom Z : N
axiom S : N → N

eval (thousand N S Z)
```


1. As an illustrative example, let us look at the Church encoding of natural numbers.
2. The idea: the number n is encoded as “compose n times”.

Exercises

1. Install OCaml and OPAM.
2. Get Faux type theory and compile it.
3. Is it really the case that equality is used precisely once in `lib/core/typecheck.ml`?
4. Why do the internal local definitions TT.Let store the type?
5. Find the code for type ascription and transcribe the bidirectional typing rules from it.