Implementing a Typechecker for an Esoteric Language: Experiences, Challenges, and Lessons

Alex Rice

University of Edinburgh, UK
alex.rice@ed.ac.uk

The type theory CATT [FM17] is a dependently typed language which models weak ∞-categories [Ben20]. Recent work has introduced CATT_{su} [FRVR22] and CATT_{sua} [FRV24], which extend CATT with a non-trivial definitional equality relation and respectively model strictly unital ∞-categories and strictly unital and associative ∞-categories. Normalisation algorithms for both of these theories are given by demonstrating the existence of a reduction system which is both strongly terminating and confluent. Naively implementing a typechecker based on such a system can be inefficient, due to the overhead of generating each intermediate term produced by these reductions.

The reductions found in these systems are non-standard, with neither theory containing lambda abstraction, application, or beta reduction, preventing techniques from existing literature (e.g. [Abe13, GSB19, GSA+22, AHS95]) from being directly applicable. These reductions are also syntactically complex, and have non-trivial interactions, which motivates further study of normalisation procedures for these theories.

In the talk I will introduce an implementation of CATT, CATT_{su}, and CATT_{sua}, which can be found at:

https://github.com/alexarice/catt-strict

Instead of delving too heavily on the specifics of the languages being typechecked, the talk will instead focus on the overall structure of the tool, some of the decisions made in its construction, and more generally my experience and the lessons learnt on the way.

When implementing a typechecker, there are multiple fundamental design decisions with different tradeoffs. The chosen design was optimised to achieve the following objectives, which don't necessarily fall within the core remit of a typechecker or interpreter.

- Type Inference: type information can be omitted in places where it can be inferred.
- Efficient evaluation: by utilising Normalisation by Evaluation (NbE) [Abe13, BSV91], evaluation of larger terms is near instantaneous.
- Preservation of variable names: variable names are often chosen by the user to convey content, but named variables are problematic to deal with due to alpha equivalence. Our tool utilises de Bruijn levels to represent variables but stores the original variable names in the context.
- Top-level bindings: Terms can be bound to global symbols. Furthermore, the tool is careful not to eagerly replace a top-level symbol by its definition, as similarly to the above point, the name of a top level symbol often conveys the intent of the user, making the term easier to read.
- Detailed error reporting: The majority of a user's interaction with a typechecker will likely be receiving error messages while debugging. The errors in our tool take the following form:

```
Error: Given term "x" does not match inferred term "y"
    ╭─[examples/test.catt:19:34]
    │
 19 │ def error_test x{f}y = comp (x{f}x)
    │                                  ┬
    │                                  ╰── Given term
────╯
```

Displaying such errors requires a correspondence between parsed terms in the memory of the tool and their location in the text.

While these problems, in particular the first two, have been individually studied, figuring out how to combine these into a single tool can be difficult. This is amplified when working on a non-standard language, as much of the existing literature focuses on the lambda calculus and its extensions.

The core of the typechecker is based on a bidirectional typechecking algorithm (see [DK21]), which splits the various typing rules into inference rules and checking rules. Contrary to more familiar type theories, all types in Catt are inferable, and we instead find a split between rules where the *context* can be inferred or checked.

The form of NbE implemented in the tool is largely inspired by the paper "Implementing a modal dependent type theory" [GSB19], although we note that the theory Catt is vastly different to the type theory they present.

The tool contains 3 main representations of syntax:

- Raw syntax: the raw syntax is intended to match the written syntax of the language as closely as possible, and represents variables by their names. Its primary purpose is to be the target of parsing and the source of printing; terms are only displayed by converting them to raw syntax first.

  The data structure implementing the raw syntax has a generic type parameter which can be used to add optional annotations to each constructor. We instantiate this type parameter to a type of "text locations" when the syntax arises from parsing and can otherwise instantiate it to the unit type. These (optional) text locations enable more informative error information.

- Core syntax: this syntax represents typechecked terms, and uses de Bruijn levels. Our typechecking procedure takes raw syntax and produces core syntax.

- Normal form syntax: this represents the possible normal forms of evaluation within the theory. The NbE evaluation algorithm converts a core syntax term to a normal form syntax term, and these terms can be quoted back to core syntax terms.

In practice, the use of all three types of syntax are heavily intertwined, due to the nature of dependent typing. During the talk, I will explain how this split simplifies various stages of the typechecking algorithm.

I will also explain what I believe is the largest flaw of this setup, the difficulty of debugging; core and normal form syntax is hard to print in an informative way, and implementation errors were often not caught early, making it difficult to implement the non-trivial reductions of these theories.

References

[Abe13] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. Habilitation thesis, Ludwig-Maximilians-Universität München, 2013.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, volume 953, pages 182–199, 1995.

[Ben20] Thibaut Benjamin. *A Type Theoretic Approach to Weak Omega-Categories and Related Higher Structures*. Theses, Institut Polytechnique de Paris, November 2020.

[BSV91] Ulrich Berger, Helmut Schwichtenberg, and R. Vermuri. An inverse of the evaluation functional for typed Lambda-calculus. In R. Vermuri, editor, *6th Annual IEE Symposium on Logic in Computer Science (LICS'91)*, pages 203–211, Amsterdam, 1991. Ludwig-Maximilians-Universität München.

[DK21] Jana Dunfield and Neel Krishnaswami. Bidirectional Typing. *ACM Comput. Surv.*, 54(5):98:1–98:38, May 2021.

[FM17] Eric Finster and Samuel Mimram. A type-theoretical definition of weak ω-categories. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, June 2017.

[FRV24] Eric Finster, Alex Rice, and Jamie Vicary. A Syntax for Strictly Associative and Unital ∞-Categories. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '24, pages 1–13, New York, NY, USA, July 2024. Association for Computing Machinery.

[FRVR22] Eric Finster, David Reutter, Jamie Vicary, and Alex Rice. A Type Theory for Strictly Unital ∞-Categories. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, pages 1–12, New York, NY, USA, August 2022. Association for Computing Machinery.

[GSA+22] Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. Controlling unfolding in type theory, October 2022.

[GSB19] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a modal dependent type theory. *blott: An Implementation of a Modal Dependent Type Theory*, 3(ICFP):107:1–107:29, July 2019.