Bidirectional Typing for Internal Solvers Using Term Reconstruction

In this talk, we present a minimal lambda calculus with dependent types and bidirectional type checking which would allow internal implementation of SMT solvers, without external script languages or metaprogramming features. For demonstration, we implement a solver for an equational theory of monoids.

Additional Key Words and Phrases: dependent types, bidirectional typing, partial inversion

ACM Reference Format:
. 2025. Bidirectional Typing for Internal Solvers Using Term Reconstruction. In *Proceedings of International Conference on Functional Programming (ICFP'25).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Being able to formally verify a complex mathematical theorem or to prove that a critical piece of software runs correctly is always a delight. However, the work on such a task almost certainly would require to prove lots of technical lemmas which are nigh trivial for a human mind but which computer might struggle to solve automatically. To overcome this, either some kind of proof search is run or specific solvers of decidable theories are employed. Here we focus on solvers of the second kind; they are usually implemented either outside of the prover (Coq [6], Arend [1]) or inside but still require metaprogramming features to work properly (Lean [4], Agda [7]). However, it would be much better if solvers could be written internally and without extra machinery to allow for faster adoption of such solvers, their observability and verifyability. Turns out that this is possible via smart application of automatic function inversion and unification algorithms inside general bidirectional typing discipline.

2 SKETCH OF A SOLVER

Basic calculus is Martin-Löf type theory with universes [3], a typed lambda-calculus which can be viewed as a terse notation for proofs in higher-order intuitionistic logic. MLTT includes **types** for formal description of mathematical structures (which can be viewed as sets under set-theoretical semantics) and **terms** which describe constructions of type inhabitants (under set-theoretical semantics, terms correspond to elements of said sets). As a logical system, MLTT is a system of natural deduction with rules for type formation ($\Gamma \vdash \tau$ type), term introduction and term elimination ($\Gamma \vdash t : \tau$), type equality ($\Gamma \vdash \sigma \equiv \tau$), term equality ($\Gamma \vdash t \equiv u : \tau$) and context well-formedness ($\vdash \Gamma$ context). Most notably, MLTT includes rules necessary for declaration of inductive types (interpreted as well-founded sets), e.g. types of a) natural numbers; b) initial segments of natural number line $\underline{n}$; c) expressions in an algebraic signature with variables ranging over any given type. In addition, it contains a special mechanism to treat types as elements of special types U_i called "universes" which we do not cover in this section. On the other hand, MLTT can be viewed as a total functional

Author's address:

programming language which allows writing down functional algorithms AND prove their correctness in the same language. All this machinery allows us to write down a general schema of an internal solver for some decidable theory:

- Let $F : (n : \mathbb{N}) \to \mathbf{type}$ be a dependent type of formulas in a language of a theory with variables taken from $\underline{n}$;
- Let $\mathcal{M}$ be a type of all models of a theory;
- Let P be an embedding function which interprets formulas of a theory as internal propositions of MLTT:
$$P : (n : \mathbb{N}) \to (M : \mathcal{M}) \to (C : \underline{n} \to M) \to F\ n \to \mathbf{prop};$$
- Then, an *internal proof* Prf of a formula ϕ is a proof that it holds universally, over all models and interpretations:
$$\mathrm{Prf}(n : \mathbb{N})(\phi : F\ n) := (M : \mathcal{M}) \to (C : \underline{n} \to M) \to P\ n\ M\ C\ \phi$$
- Let $\mathcal{D}$ be a decision function which, in fact, expresses the logic of a solver inside MLTT:
$$\mathcal{D} : (n : \mathbb{N}) \to (\phi : F\ n) \to (\mathrm{Prf}\ n\ \phi \vee (\mathrm{Prf}\ n\ \phi \to \bot));$$
- Then, we would like to define a "wrapper" of a type
$$(n : \mathbb{N}) \to (\phi : Fn) \to (\pi : \mathrm{Prf}\ n\ \phi) \to (M : \mathcal{M}) \to (C : \underline{n} \to M) \to (\mathcal{D}\ n\ \phi = \mathrm{inl}\ \pi) \to P\ n\ M\ C\ \phi.$$
The type of a wrapper is constructed in such a way that arguments can be provided to it iff the decision function states that a formula is, in fact, valid.

In order for this framework to be practical, however, we have to solve two problems:

(1) A typing relation $\Gamma \vdash t : \tau$ has to be decidable;
(2) All of the wrapper's arguments (except, maybe, for C) have to be inferrable from resulting type.

To this end, we develop a special bidirectional typing system [2] for Martin-Löf type theory with implicit arguments. Inference of arguments is achieved via function inversion as in Haskell^{-1} [5] (here, we invert P to infer ϕ) and via unification algorithm embedded in the typing relations (here, we unify $\mathcal{D}\ n\ \phi$ with $\mathrm{inl}\ \pi$). Note that this gives rise to general inference system which might have wider applications.

3 THE MOST IMPORTANT INFERENCE RULES

$$\textsc{app}\ \frac{\Gamma \vdash f\ \$\ t, \Delta : T}{\Gamma \vdash f\ t\ \$\ \Delta : T} \qquad \textsc{var}\ \frac{x : U \in \Gamma \qquad \Gamma \vdash U\ @\ \Delta : T}{\Gamma \vdash x\ \$\ \Delta : T} \qquad \textsc{@-pi}\ \frac{\Gamma; \sigma \vdash t \Leftarrow T; \tau \qquad \Gamma; \rho \vdash U[x := t]\ @\ \Delta : V; \sigma}{\Gamma; \rho \vdash (x : T) \to U\ @\ t, \Delta : V; \tau}$$

$$\textsc{@-check}\ \frac{\Gamma \vdash T = U \Leftarrow \mathcal{U}}{\Gamma \vdash T\ @\ \cdot \Leftarrow U} \qquad \textsc{@-infer}\ \frac{}{\Gamma \vdash T\ @\ \cdot \Rightarrow T}$$

4 ALL ELABORATION RULES

$$\textsc{Type}\quad\frac{}{\Gamma \vdash \mathcal{U}_l \Rightarrow \mathcal{U}_{l+1}}$$

$$\textsc{Pi}\quad\frac{\Gamma;\sigma \vdash T \Rightarrow \mathcal{U}_{l_T};\tau \qquad \Gamma, x : T;\rho \vdash U \Rightarrow \mathcal{U}_{l_U};\sigma}{\Gamma;\rho \vdash (x : T) \to U \Rightarrow \mathcal{U}_{l_T \vee l_U};\tau}$$

$$\textsc{Abs-0}\quad\frac{\Gamma;\sigma \vdash T \Leftarrow \mathcal{U};\tau \qquad \Gamma, x : T;\rho \vdash f \Rightarrow U;\sigma}{\Gamma;\rho \vdash \lambda(x : T).f \Rightarrow (x : T) \to U;\tau}$$

$$\textsc{Pair}\quad\frac{\Gamma \vdash f \Rightarrow T \qquad \Gamma \vdash s \Leftarrow U[x := f]}{\Gamma \vdash (f, s : U) \Rightarrow (x : T) \times U}$$

$$\textsc{Id}\quad\frac{\Gamma \vdash t \Leftarrow T \qquad \Gamma \vdash u \Leftarrow T \qquad \Gamma \vdash T \Rightarrow \mathcal{U}_l}{\Gamma \vdash t =_T u \Rightarrow \mathcal{U}_l}$$

$$\textsc{App}\quad\frac{\Gamma;\sigma \vdash f \ @\ t, \zeta \Rightarrow r;\sigma'}{\Gamma;\sigma \vdash f t \ @\ \zeta \Rightarrow r;\sigma'}$$

$$\textsc{Abs}\quad\frac{x = t : T, \Gamma;\sigma \vdash f \ @\ \zeta \Rightarrow r;\sigma'}{\Gamma;\sigma \vdash \lambda(x : T).f \ @\ t, \zeta \Rightarrow r;\sigma'}$$

REFERENCES

[1] Arend. 2023. Arend Theorem Prover. https://arend-lang.github.io.

[2] Thiago Felicissimo. 2024. Generic bidirectional typing for dependent type theories. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 143–170.

[3] Per Martin-Löf. 1980. *Intuitionistic Type Theory.* Bibliopolis.

[4] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635.

[5] Finn Teegen, Kai-Oliver Prott, and Niels Bunkenburg. 2021. Haskell^{-1}: automatic function inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell* (Virtual, Republic of Korea) *(Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 41–55. https://doi.org/10.1145/3471874.3472982

[6] The Coq Development Team. 2024. The Coq Reference Manual – Release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman.

[7] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda.* https://plfa.inf.ed.ac.uk/22.08/