Artifact report: Generic bidirectional typing for dependent type theories

Thiago Felicissimo(✉)

Université Paris-Saclay, INRIA, LMF, ENS Paris-Saclay, Gif-sur-Yvette, France
thiago.felicissimo@inria.fr

Abstract. We report on the implementation of a generic bidirectional algorithm for dependent type theories, following the proposal of the paper "Generic bidirectional typing for dependent type theories".

In [5] we have proposed a general definition of dependent type theories supporting bidirectional typing, and established an equivalence between their declarative and bidirectional type systems. The crucial property satisfied by the bidirectional system is its decidability for normalizing theories, which allowed for its implementation in OCaml in the tool `BiTTs` [6] which we describe here.

1 A quick introduction to the implementation

Let us first start with a concrete example of how to use the tool. Because the algorithm implemented is theory-independent, the first step to use it is to specify the theory we want to work in. This is done with the commands `sort`, `constructor`, `destructor` and `rewrite` which specify respectively sort, constructor, destructor and rewrite rules. For instance, the following declarations define the theory $\mathbb{T}_{\lambda\Pi}$ given in [5, Example 6], constituting a minimalistic Martin-Löf Type Theory with dependent functions.

```
sort Ty ()
sort Tm (A : Ty)
constructor Π () (A : Ty, B{x : Tm(A)} : Ty) : Ty
constructor λ (A : Ty, B{x : Tm(A)} : Ty) (t{x : Tm(A)} : Tm(B{x})) : Tm(Π(A, x. B{x}))
destructor @ (A : Ty, B{x : Tm(A)} : Ty) (t : Tm(Π(A, x. B{x}))) (u : Tm(A)) : Tm(B{u})
rewrite @(λ(x. t{x}), u) --> t{u}
```

Once the theory is specified, we can start writing and typechecking terms inside it. For instance, supposing we have also added a Tarski-style universe `U`, we can check the following definition of the polymorphic identity function.

```
let idU : Tm(Π(U, a. Π(El(a), _. El(a)))) := λ(a. λ(x. x))
```

To typecheck this definition, the tool first verifies that the sort given in the annotation is indeed well-typed, and then checks the body of the definition

S. Weirich (Ed.): ESOP 2024, LNCS 14576, pp. 171–175, 2024.
https://doi.org/10.1007/978-3-031-57262-3_7

against the sort. If all the steps succeed, the identifier is added to a global scope of top-level definitions and becomes available to be used in the rest of the file.

Supposing that the underlying theory is valid, Theorem 2 of [5] ensures that, if the implementation says that a term is well-typed, then the term is indeed well-typed in the declarative type system of the theory. Note however that the implementation does not currently check if the supplied theory is valid. Extending the implementation to check this automatically is future work, so for the time being this verification is left to the user.

Finally, we also provide commands for evaluating terms to normal form and checking that two terms are definitionally equal. For instance, assuming we have added natural numbers to the theory and defined factorial, we can use these commands to compute the factorial of 3 and check that it is equal to 6.

```
let fact3 : Tm(ℕ) := @(fact, S(S(S(0))))
eval fact3

let 6 : Tm(ℕ) := S(S(S(S(S(0)))))
check fact3 = 6
```

The implementation also comes with some examples of theories that can be defined in the framework, along with some examples of terms written in these theories. In the directory `examples/` we can find the following files:

- `mltt.bitt` : Martin-Löf Type Theory with a type-in-type Tarski-style universe, Π and Σ types, lists, booleans, and the unit, empty and W types.
- `mltt-coquand.bitt` : Martin-Löf Type Theory with a hierarchy of (weak) cumulative Coquand-style universes and universe polymorphism, with Π types and natural numbers.
- `hol.bitt` : Higher-Order Logic (also known as Simple Type Theory) with implication and universal quantification.

2 The implementation

The core of the implementation can be separated into two main parts: the typechecking and the normalization algorithms. Let us now discuss them in detail.

Normalization

Because the theories we support are dependently-typed, typechecking terms requires equality checking, which in turn requires reducing terms to normal form. In order to do so, we have implemented an untyped variant of *Normalization by Evaluation (NbE)*, inspired by the works of Coquand [4], Abel [1] and Kovacs [8]. In NbE, terms are evaluated into a separate syntax of runtime values, in which binders are represented by closures and free variables by unknowns. Values can then be compared for equality by entering closures and recursively evaluating and comparing their bodies. One of the benefits of this approach is that, by

using de Bruijn indices in the syntax of regular terms but de Bruijn *levels* in the syntax of values, we completely avoid the need of implementing substitution or index-shifting functions.

Let us go through the main functions used to implement normalization. In the following, we only discuss those that operate on terms, but each one has a counterpart for metavariable substitutions. First, because the definitional equality of theories is generated by customizable rewrite systems, rewriting requires matching against patterns. This is done by the function

```
val match_tm : p_tm -> v_tm -> v_msubst
```

which matches a term value against a term pattern and produces a metavariable substitution of values (the prefix `p_` stands for pattern, while `v_` stands for value).

This is then used in the function

```
val eval_tm : tm -> v_msubst -> v_subst -> v_tm
```

which evaluates a term under a `v_subst` mapping occurring variables to values and a `v_msubst` mapping occurring metavariables to values or closures. This is done by recursively evaluating subterms, then trying to match against one of the rewrite rule's left hand sides and finally recursively evaluating the right hand side under the metavariable substitution returned by matching.

Finally, values can be checked for equality with the function

```
val equal_tm : v_tm -> v_tm -> int -> unit
```

which recursively enters and evaluates closures while checking the result for equality, and raising an exception when the two given terms are not equal. The third argument is used for generating fresh unknowns when entering closures.

Typechecking

The typechecking algorithm is composed of four main functions, each one implementing one of the judgment forms of the bidirectional system of [5].

Inference $\Gamma \vdash t \Rightarrow T$ and checking $\Gamma \vdash t \Leftarrow T$ are implemented by the functions

```
val infer : v_ctx -> v_subst -> tm -> v_tm
val check : v_ctx -> v_subst -> tm -> v_tm -> unit
```

the first returning the inferred sort and the second returning unit on success. Note that, following works such as [4,8,7], we tightly integrate it with the NbE algorithm by asking all inputs to be already in the syntax of values, with the exception of the subject of the typing judgment. Compared with the usual inference and checking judgments, note also the addition of the second argument `v_subst` used to map the variables of the context `v_ctx` to unknowns for when needing to evaluate the subject.

The third judgment $\Gamma \mid \mathbf{v} : \Theta \vdash \mathbf{u} \Leftarrow \Xi$ used to typecheck metavariable substitutions is then rendered as the function

```
val check_msubst : v_ctx -> v_subst -> v_msubst -> msubst -> mctx -> v_msubst
```

in which the argument corresponding to Θ is omitted for it is computationally irrelevant. We also return the value of the checked metavariable substitution, which comes in handy when coding the recursive case of its definition. Finally, the last judgment $\Gamma \vdash T \Leftarrow \mathsf{sort}$ is implemented by the function

```
val check_sort : v_ctx -> v_subst -> tm -> unit
```

whose type signature follows the same reasoning as above.

Differences with respect to the theory

We highlight some relevant differences regarding the theory presented in [5].

First, we do not support matching inside binders, which restricts the set of patterns we can write. For instance, while the pattern λ`(x.t{x})` is accepted, the pattern λ`(x.S(t{x}))` (assuming that `S` is a constructor) would be rejected by the implementation. This is because matching against it would require matching inside a closure and then reading back the result into the syntax of terms, which would be highly inefficient with our NbE setup. Thankfully, matching inside binders is almost never needed and none of our provided examples require it.

Second, even though the inference system for matching and the proof of decidability of conversion in [5] employ the maximal outermost strategy, our NbE normalizer uses instead a call-by-value strategy. The maximal outermost strategy has the theoretical advantage over call-by-value of being normalizing, which means that it always terminates for weak normalizing theories. However, most theories used in practice are either strong normalizing or not normalizing at all. Moreover, call-by-value can be implemented very easily using our described NbE setup, which is the reason we opted for it instead.

Third, instead of defining the typing functions over a specific grammar of checkable/inferable terms as done in [5], we define them over the grammar of (regular) terms. This means that these functions might discover in the process that the term given is not checkable/inferable, in which case an error is given.

Finally, as seen in Section 1 our implementation also extends the bidirectional system with top-level definitions, which is crucial for allowing to write terms in a user-friendly manner.

3 Future work

The current implementation is still a prototype and can be extended in various ways. In particular, error handling is still rudimentary and improving it will be key in order to make `BiTTs` more user-friendly.

We also plan to further test our implementation with larger and more realistic examples. In particular, we would like to compare it with typecheckers for Dedukti [2,3], a framework aimed at providing a universal typechecker geared towards proof-system interoperability. Because Dedukti has no support for erased arguments, its terms are highly-annotated, which can have an important impact on performance. Our support for non-annotated syntaxes should therefore allow for shorter typechecking times, an hypothesis we hope to confirm with these tests.

References

1. Abel, A.: Normalization by evaluation: Dependent types and impredicativity. Habilitation. Ludwig-Maximilians-Universität München (2013)
2. Assaf, A., Burel, G., Cauderlier, R., Delahaye, D., Dowek, G., Dubois, C., Gilbert, F., Halmagrand, P., Hermant, O., Saillard, R.: Dedukti: a logical framework based on the $\lambda\,\pi$-calculus modulo theory (2016), unpublished
3. Blanqui, F., Dowek, G., Grienenberger, É., Hondet, G., Thiré, F.: Some axioms for mathematics. In: Kobayashi, N. (ed.) 6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference). LIPIcs, vol. 195, pp. 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.20, https://doi.org/10.4230/LIPIcs.FSCD.2021.20
4. Coquand, T.: An algorithm for type-checking dependent types. Science of Computer Programming **26**(1-3), 167–177 (1996)
5. Felicissimo, T.: Generic bidirectional typing for dependent type theories (2023)
6. Felicissimo, T.: BiTTs (Jan 2024). https://doi.org/10.5281/zenodo.10500598, https://doi.org/10.5281/zenodo.10500598
7. Gratzer, D., Sterling, J., Birkedal, L.: Implementing a modal dependent type theory. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–29 (2019)
8. Kovács, A.: elaboration-zoo (2023), https://github.com/AndrasKovacs/elaboration-zoo