Decidable Type-Checking for Bidirectional Martin-Löf Type Theory

Meven Lennon-Bertrand and Neel Krishnaswami

University of Cambridge, UK

Abstract

In this work, we describe a presentation of dependent type systems rooted in bidirectional ideas, carefully separating at the syntax level between inferring and checking terms. This leads to a system which requires exactly the annotations needed to decide type-checking. Moreover, it readily embeds the two most usual ways to handle typing in dependent type systems, either by restricting to a certain subclass of terms that do not need annotations (as done in Agda), or by demanding certain annotations to be provided (as done in Coq), providing an elegant unifying framework.

1 Dependent type-checking is more subtle than you think

While a large body of work has been devoted to showing decidability of conversion for various complex dependent type systems, decidability of typing has attracted comparatively little interest. However, it is a more subtle question than on can at first think.

Type-checking for dependent type systems is, in general, undecidable [3]. The main culprits are (β-)redexes: when considering $f\ u$, a type-checker typically infers a type $\Pi x\colon A.B$ for the function f, then checks u against A. However, when the function f is an abstraction $\lambda x.t$, there is nowhere this A can be obtained from! Trying to infer A from u runs into a similar issue.

There are two standard approaches to get out of this difficulty. The first one is to restrict the terms fed to the type-checker to a subset for which type-checking becomes decidable again. The kernel of the Agda proof assistant, for instance, only manipulates terms in normal forms [9], for which type-checking is generally decidable whenever conversion is [1, 5].

The second approach is to decorate terms with type annotations, to ensure one can always *infer* a type for any typable term. For instance, in Coq or Lean, the kernel only deals with annotated abstractions $\lambda x\colon A.t$, which contain exactly the information we were missing in the redex earlier. Again, this leads to decidable type-checking when conversion is decidable [6].

In practice, these limitations are mitigated at the user level using unification, allowing for a syntax which is more flexible than the kernel's. Still both approaches still have significant drawbacks. Because Agda can only represent normal forms, it has to eagerly normalize terms, and cannot type-check intermediate steps of its reduction machine. Because Coq can only represent terms that infer, it can be quite inefficient, as annotations create a lot of redundancy. Finally, elaboration based on unification is inherently incomplete. This makes them less predictable, forcing users to get an intuition as to which unification problems their tool can solve. Designing a system which has a complete, and thus predictable, type-checking, and has less of the aforementioned shortcomings, thus seems like a desirable goal.

2 A bidirectional analysis

To better understand the issue, we can turn to bidirectional typing [4], an analysis of type-checking algorithms which emphasizes the difference between two *modes*, between which most

such algorithms alternate: checking–where the type is known–and inference–where it is to be found. In this view, the issue with our redex is that in an application $f\ u$ we want the function f to infer a type, but an abstraction $\lambda x.t$ can only be reasonably checked. Both solutions sketched in Section 1 are quite radical: Agda forces f to be a neutral term which always infers, while Coq demands that *all* terms infer a type. What if, instead, we simply demanded that f inferred a type, whatever way it achieves this? What if we separated, already in the syntax, between inferring terms and checking terms to be able to express this demand?

This idea has already appeared in the literature [5, 8]. Both works, however, use it only to relax Agda-style type-checking, by adding an annotation $t :: A$ to a language otherwise completely devoid of them, allowing writing a β-redex as $(\lambda x.t :: \Pi x\colon A.B)\ u$. But this is heavier than Coq-style annotations, because in general only the domain annotation is really needed. So, let us simply throw these in as well! Altogether, the functional fragment of such a language looks as follows, divided in the two, mutually defined checking and inferring terms:

$$c ::= \underline{i} \mid \lambda x.c \qquad\qquad i ::= c :: A \mid x \mid i\ c \mid \lambda x\colon A.i \mid \Pi x\colon A.B \mid \square_k$$

with both typed and untyped abstractions, types (in the inferring fragment, as we luckily can infer their type), and $\underline{i}$, the converse of annotation that forgets that its argument is inferring. While not presented here for lack of space, this presentation easily extends to virtually any feature present in modern dependent type systems, such as inductive types or negative records.

Because we design the language to respect the structure of bidirectional algorithms, these work perfectly, and typing is decidable for systems in this fashion as soon as conversion is. Moreover, both earlier approaches can be carved out as subsystems: the first, by using neither $t :: A$ nor annotated abstraction; the second, by restricting to the inferring fragment, using no checking term but $\underline{i}$. We thus get two proofs of decidability for the price of one.

3 Substitution, reduction and conversion

Of course, this system would not be any good without a well-behaved conversion. Before coming to it, however, we must beware of substitution. The naïve reduction of $(\lambda x\colon A.t)\ u$ to $t[x := u]$ is incorrect, as it does not preserve modes: the variable x infers, but u merely checks, and so replacing one by the other is incorrect. Rather, the correct β-redex is instead $t[x := u :: A]$, as it is *mode-preserving*. If we care about preservation of typability during a small step reduction chain, typically to be able to read back terms to users, then this is the correct substitution rule.

The second point of interest are of course annotations. Here we can take inspiration from the transport of observational equality [2, 10] or the casts of gradual typing [7]: annotations reduce when both the type and terms are canonical, and propagate down: $(\lambda x.t) :: (\Pi x\colon A.B) \to \lambda x\colon A.(t :: B)$. Combining this with the previous rule for β-redex, we recover the earlier rule of McBride [8]. Conversely, we can erase useless annotations: $\underline{\lambda x\colon A.t} \to \lambda x.\underline{t}$. Finally, we also want some form of extensionality for annotations: $\underline{t :: A}$ should be convertible to t.[1] Just as for casts on reflexivity in the latest version of observational equality [11], however, we do not want to see this as a reduction rule, but rather as an equation only needed on neutral terms. In the end, we get a system close to that of Pujet and Tabareau [11], and we expect being able to adapt their technique to show decidability of our conversion.

If, however, we are only interested in evaluation to decide conversion, we know that normal forms do not contain annotations. Why, then, bother with maintaining those? Instead, we can erase them in a normalization by evaluation approach, similar to e.g. Gratzer et al. [5]. This speeds up conversion-checking, at the cost of losing a well-typed reduction sequence.

[1] Note that it does not make sense to compare simply $t :: A$ and t, as they do not have the same mode.

References

[1] Andreas Abel and Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". In: *Logical Methods in Computer Science* Volume 8, Issue 1 (Mar. 2012). DOI: 10.2168/LMCS-8(1:29)2012. URL: https://lmcs.episciences.org/1045.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*. PLPV '07. Freiburg, Germany: Association for Computing Machinery, 2007, pp. 57–68. ISBN: 9781595936776. DOI: 10.1145/1292597.1292608.

[3] Gilles Dowek. "The undecidability of typability in the Lambda-Pi-calculus". In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 139–145.

[4] Jana Dunfield and Neel Krishnaswami. "Bidirectional Typing". In: *ACM Computing Surveys* 54.5 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3450952.

[5] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. "Implementing a Modal Dependent Type Theory". In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341711.

[6] Meven Lennon-Bertrand. "Bidirectional Typing for the Calculus of Inductive Constructions". PhD thesis. Nantes Université, 2022.

[7] Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. "Gradualizing the Calculus of Inductive Constructions". In: *ACM Transactions on Programming Languages and Systems* 44.2 (Apr. 2022). ISSN: 0164-0925. DOI: 10.1145/3495528.

[8] Conor McBride. "Types Who Say Ni". 2022.

[9] Ulf Norell. "Towards a practical programming language based on dependent type theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007.

[10] Loïc Pujet. "Computing with Extensionality Principles in Type Theory". PhD thesis. Nantes Université, 2022.

[11] Loïc Pujet and Nicolas Tabareau. "Impredicative Observational Equality". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023). DOI: 10.1145/3571739.