Equivalence Between Typed and Untyped Conversion Algorithms

Meven Lennon-Bertrand

Inria – LS2N, Université de Nantes, Nantes, France

Abstract

This talk shall present a new take into the innocent-looking but difficult issue of relating the typed, judgmental and the untyped, rewriting-based approaches to conversion for dependent type theories, by considering algorithmic presentations of those.

1 A Tale of Two Conversions

An important component of dependent type systems is how they incorporate computation. This is usually done by means of a conversion rule, which allows replacing a type by one related to it by the conversion relation $\cong$. There are, however, two quite different approaches to defining that relation. In the first, that we will hereafter call *typed* conversion, it is presented by means of a judgement $\Gamma \vdash t \cong t' : A$, described by inference rules in much the same way as typing. The type is used in the rules, as showcased in the following examples:

$$\textsc{Fun}\ \frac{\Gamma, x : A \vdash f\ x \cong g\ x : B}{\Gamma \vdash f \cong g : \Pi x : A.B} \qquad \beta\ \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x : A.t)\ u : B[x := u]}$$

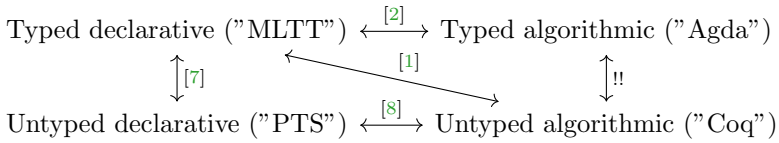
The second, that we call *untyped* conversion, is built on top of an untyped reduction relation: conversion is taken as the symmetric, reflexive, transitive closure of reduction.

The two definitions have been used concurrently for a long time. Typed conversion is a prominent characteristic of Martin-Löf Type Theory [5], and is the main source of inspiration for the implementation of Agda. Untyped conversion is the one favoured in Pure Type Systems (PTS) [3], and Coq follows this approach by implementing an untyped conversion routine.

While the two approaches are conceptually close, proving the equivalence between them is difficult. The best attempts to date are that of Siles and Herbelin [7], and of Abel and Coquand [1]. Siles and Herbelin's strategy is purely syntactic, and thus requires a weak meta-theory, but does not cover extensionality rules such as Fun above. Abel and Coquand handle these extensionality rules, but they need a meta-theory powerful enough to prove normalization of the object system. Thus, their approach applies only to systems that are not logically weak – they do not handle an impredicative sort of propositions, as present in Coq.

2 Enter the Algorithmic Presentations

These presentations of Section 1 – which we will call *declarative* – are quite remote from actual implementations. Thus, other presentations – that we call *algorithmic* – have been introduced, which aim to be much closer to the actual implementations of conversion checking than their declarative counterparts. In particular, they remove the transitivity rule, which is very problematic when trying to build an implementation. Such presentations have been given in [2] for typed conversion, and in MetaCoq [8] for untyped conversion, and in both cases proven equivalent to their declarative counterparts. The general picture thus looks like this:



Now, what about the equivalence in the right column? Can we prove it? With what tools, and what logical power? This is of particular interest because the equivalence of [8] does not currently handle extensionality rules. Indeed, to the best of our knowledge, giving an untyped, declarative presentation of Fun in a setting with other types than functions is an open problem. Proving the equivalence might give a way out, by moving the study to typed conversion where the picture is clearer – see [2] and its extensions [4, 6] –, and seeing the untyped algorithm as a mere optimization of the typed one. This way, we could relate a conversion-checking algorithm such as that of Coq to a typed specification, extensionality rules are less difficult to handle.

3 Proofs!

The equivalence. The two algorithmic conversions are built around the same idea, namely the interleaving of (weak-head) reduction and structural rules. Terms – and when applicable their type – are reduced to weak-head normal forms, and then structural rules may be applied based on the structure of those. The difference between the two systems resides in the way these structural rules work: in the typed variant, they are primarily type-directed – akin to Fun –, while the untyped variant only uses information available from the term. Showing the equivalence thus amounts to proving that both approaches give the same result, and so in particular that the type information is actually superfluous.

In the case of functions, what we need is that the two following rules, together with a symmetric version of λ-R and generic rules for neutral terms, can emulate Fun.

$$\lambda\text{-R}\quad \frac{\Gamma, x : A \vdash t \cong n\ x \qquad n \text{ is not a } \lambda}{\Gamma \vdash \lambda x : A.t \cong n} \qquad \lambda\text{-Cong}\quad \frac{\Gamma, x : A \vdash t \cong t'}{\Gamma \vdash \lambda x : A.t \cong \lambda x : A'.t'}$$

In both directions, the main difficulty is to show that the algorithm properly maintains well-typedness invariants. This in turn relies on standard meta-theoretical properties (reflexivity of conversion, stability by substitution, subject reduction...). In the direction from typed to untyped conversion, the key ingredient is then to show injectivity of η-expansion, that is that if $\Gamma, x : A \vdash f\ x \cong g\ x$ and both f and g have a function type in Γ, then $\Gamma \vdash f \cong g$. This is done by analysing the possible weak-head normal forms for both f and g. The other direction amounts to reconstructing the typing information, using the well-typedness invariants, since Fun directly subsumes all four undirected rules. A formalization is ongoing to check the details of this proof sketch.

What power do we need? All the required meta-theoretic properties are easy consequences of [2]. However, due to the very structured form of algorithmic conversion, most of them imply normalization, so they must be assumed if we want to keep using a low logical power in our proof of equivalence. A middle ground is to only assume one single property and derive all its consequences, as is done by *e.g.* MetaCoq, where a single normalization axiom is enough to fuel the whole development. Stability by substitution is strong enough to imply all other hypothesis, but it is currently unclear whether normalization would also suffice. Again, the ongoing formalization should provide a more solid setting to investigate this subtle point.

References

[1] Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue.

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[3] Henk Barendregt. An introduction to generalized type systems. *Journal of Functional Programming*, 1:125–154, April 1991.

[4] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without k. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, January 2019.

[5] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory.* Number 1 in Studies in Proof Theory. Napoli: Bibliopolis, 1984.

[6] Loïc Pujet and Nicolas Tabareau. Observational equality: Now for good. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[7] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *J. Funct. Program.*, 22(2):153–180, 2012.

[8] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.