Sprinkles of extensionality for your vanilla type theory

Jesper Cockx[1] and Andreas Abel[2]

[1] DistriNet – KU Leuven
[2] Department of Computer Science and Engineering – Gothenburg University

Dependent types can make your developments (be they programs or proofs) dramatically safer by allowing you to prove that you implemented what you intended. Unfortunately, they can also make your developments dramatically more boring by requiring you to elaborate those proofs in often painstaking detail. For this reason, dependently typed languages typically allow you to cheat by postulating some facts as axioms. However, type theory is not just about which types are inhabited; it's about how things *compute*. Computationally, an axiom is a stuck term, so it prevents you from evaluating your programs. What if you could postulate not just axioms, but also arbitrary rewrite rules?

1. A typical frustration for people new to proof assistants like Agda or Coq is that $0 + x$ evaluates to x for arbitrary x, but $x+0$ doesn't. Of course, a lemma for $x+0 = x$ is easy to prove, but having to appeal to this lemma explicitly in all subsequent uses is bothersome. By adding a rewrite rule $x+0 \longrightarrow x$, you can get the automatic application of this lemma. Similarly, you can add a rule $x + \texttt{suc}\ y \longrightarrow \texttt{suc}\ (x + y)$, or $(x + y) + z \longrightarrow x + (y + z)$.

2. Allais, McBride, and Boutillier (2013) present a series of rewrite rules (which they call ν-rules) for functions on pairs and lists. For example, they have a rule for concatenating with an empty list ($l +\!\!+ \texttt{[]} \longrightarrow l$), but also rules for simplifying expressions involving `map` and `fold` (e.g. $\texttt{map}\ (\lambda x.\, x)\ l \longrightarrow l$).

3. Homotopy type theory (The Univalent Foundations Program, 2013) presents the concept of *higher inductive types*, inductive types that have not only regular constructors, but also path constructors that introduce additional equalities. However, current implementations of higher inductive types only have evaluation rules for applying a function to a regular constructor, but not for applying them to a path constructor. By adding rewrite rules to the eliminator of a higher inductive type, working with higher inductive types becomes easier and much more natural.

4. In observational type theory (Altenkirch, McBride, and Swierstra, 2007), the equality type $x =_A y$ is defined by case analysis on the type A. This can be emulated in other theories by a few custom rewrite rules, thus giving you the advantages of observational type theory, such as functional extensionality.

5. Custom rewrite rules also make it possible to define *shallow embeddings* of other languages in your language, making it possible to import developments into your own language without losing their computational properties. In fact, some languages like Dedukti (Boespflug, Carbonneaux, and Hermant, 2012) are built completely around this concept.

All these examples show how nice it can be to mix up the already sweet taste of vanilla intensional type theory with some extensional sprinkles in the form of rewrite rules. For this purpose we added a new facility to Agda, available from version 2.4.2.4 onwards with some improvements in 2.5.1, allowing you to specify proof-irrelevant rewrite rules that are plugged

into the evaluation mechanism of the typechecker. Concretely, you can declare the identity type to be the rewrite relation by the pragma {-# BUILTIN REWRITE _≡_ #-}, and then declare the proof plus0 : $x + 0 \equiv x$ to be a rewrite rule by the pragma {-# REWRITE plus0 #-}. By adding this lemma as a rewrite rule, it holds for arbitrary open terms, thus simplifying the definition of functions involving natural numbers in their types. For example, if you also add plusSuc : $(x\ y : \mathbb{N}) \to x + (\mathtt{suc}\ y) \equiv \mathtt{suc}\ (x + y)$ as a rewrite rule, then the following proof just works, instead of complaining that $x + 0 \neq x$ or $x + (\mathtt{suc}\ y) \neq \mathtt{suc}\ (x + y)$:

```
plus-comm  :  (x y : ℕ) → x + y ≡ y + x
plus-comm  zero    y  =  refl
plus-comm  (suc x) y  =  cong suc (plus-comm x y)
```

Our implementation of rewrite rules can also handle examples 2 and 3 without problems. We haven't tried 4 or 5 yet, but we see no reason why these examples wouldn't work as well.

By adding more definitional equalities, the size of your proof terms can be reduced quite drastically. Rewrite rules also allow advanced users to experiment with new evaluation rules, without actually modifying the typechecker. On the other hand, you shouldn't just use any kind of sprinkles. Even more than with axioms, rewrite rules can break the type system completely: not only soundness, but also confluence, termination, and even subject reduction are in danger. This means typechecking may become undecidable, and the typechecker may loop indefinitely. For example, adding a rewrite rule $x + y \longrightarrow y + x$ causes the typechecker to loop. It is possible (though quite difficult) to regain these properties by checking confluence, termination, and completeness of the added rewrite rules (Blanqui, 2005). An example of a fully developed system incorporating these checks is CoqMT (Strub, 2010). But no matter how good these checks are, there will always be cases where they are too restrictive and stand in the way of experimentation. So we'd rather let you decide which rewrite rules you want to add, as long as you promise to be extra careful. After all, sometimes it is more important to be able to experiment freely than it is to be 100% safe, and this is precisely the kind of situations we aim for. What can *you* do with the power of arbitrary rewrite rules? We invite you to try it for yourself!

References

Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: A sound and complete decision procedure, formalized. In *Workshop on Dependently-typed Programming*, 2013.

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Programming languages meets program verification*, 2007.

Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 2005.

Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. In *Second International Workshop on Proof Exchange for Theorem Proving*, 2012.

Pierre-Yves Strub. Coq modulo theory. In *Computer Science Logic*, 2010.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.