Towards Extraction of Continuity Moduli in Coq

Yannick Forster[1], Dominik Kirst[1], and Florian Steinberg[2]

[1] Saarland University
Saarland Informatics Campus, Saarbrücken, Germany
{forster,kirst}@ps.uni-saarland.de
[2] INRIA Saclay
Paris, Framce
florian.steinberg@inria.fr

Abstract

We report on a work-in-progress extraction of continuity information for Coq functionals on Baire space, i.e. of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$. The extraction is implemented as a MetaCoq plugin and generates a certified modulus function, given a term in the System T fragment of Coq. In fact, the extraction first reifies Coq definitions into a syntactic representation of System T and subsequently employs a constructive and informative continuity theorem for System T following Escardó.

It is a well-studied property of constructive mathematics that the functions definable in a purely constructive setting à la Bishop are computable. As a consequence, definable functions over Baire space into natural numbers are continuous.

The latter has been established explicitly for various phrasings of constructive mathematics, for instance Gödel's System T, expressing the primitive recursive functions of higher type. For more profound constructive foundations, such as the dependent type theory underlying the Coq proof assistant, it is clear that at least T-definable functions can be shown continuous in the system itself. We provide a first step towards exploiting such continuity information by implementing a plugin for automatically extracting the modulus of continuity of T-definable Coq functionals of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$.

Such a functional f on Baire space is called *continuous* if it only accesses finitely many positions of every input sequence $\alpha : \mathbb{N} \to \mathbb{N}$, i.e. if there is a function $\mu_f : (\mathbb{N} \to \mathbb{N}) \to \mathcal{L}(\mathbb{N})$ such that for every α it holds that $f\,\alpha = f\,\beta$ for every β agreeing with α on the positions listed in $\mu_f\,\alpha$. The function μ_f is called the *modulus of continuity* of f and can be extracted for all T-definable Coq functionals by first reifying into a syntactic representation of System T (Section 2) using the MetaCoq framework [3] and then executing a constructive and informative continuity proof for System T (Section 1) as implemented by Escardó in Agda [1]. The Coq code is available at https://www.ps.uni-saarland.de/extras/modulus-extraction/.

1 Extracting Continuity Moduli from System T

We follow Escardó's Agda development [1] to implement a Coq procedure that computes and verifies the modulus of continuity for T-definable functionals, i.e. functionals that are the denotation of a term of System T. Using standard techniques from programming language semantics, Escardó gives a compact mechanisation that straightforwardly translates to Coq. As intended for the calculus of constructions at the core of Coq's type theory, most of the logical statements from the Agda proof can be placed in the (impredicative) propositional universe $\mathbb{P}$ while only the definition of continuity remains in the (predicative) computational hierarchy $\mathbb{T}$, so that the modulus function can be extracted. Moreover, as in (one version of) Escardó's proof, we rely on an intrinsically typed Church-style representation of System T, i.e. do not model untyped syntax.

2 A Modulus Extraction Plugin

We utilise MetaCoq to reify Coq's System T fragment syntactically into an inductive type representing untyped System T syntax, reminiscent of reification into the untyped λ-calculus [2]. MetaCoq provides an inductive type `Ast.term` mirroring the OCaml datatype used to implement Coq and a monad `TemplateMonad` which can be used to access effects like unfolding of names, quoting a Coq term into its `Ast.term` representation or unquoting an `Ast.term` representation back into a Coq term.

Our plugin thus consists of a monadic program `Reify` translating from `Ast.term` into a type `SystemT.term`. Monadic programs can be executed using a vernacular command. To execute our plugin, a user can type `MetaCoq Run (Reify r f)` to reify `f` into System T automatically and add the result to the environment as definition named `r`. The reification function is essentially the (partial) identity, just renaming constructors of Coq (e.g. Coq's application `Ast.tApp`) into constructors of System T (e.g. `SystemT.app`). The two datatypes are displayed below, the alignment hints at how the translation works:

```
Module Ast.
  Inductive term : Set :=
    tRel : nat -> term
  | tConstruct : inductive -> nat ->
      universe_instance -> term
  | tFix : mfixpoint term -> nat -> term.
  | tLambda : name -> term -> term -> term
  | tApp : term -> term -> term
  (* ... *).
End Ast.
```

```
Module SystemT.
  Inductive term : Type :=
  | var : nat -> term
  | zero : term
  | succ : term
  | rec : type -> term
  | lambda : type -> term -> term
  | app : term -> term -> term.

End SystemT.
```

As a second step, we translate the untyped System T representation to the intrinsically typed representation for the continuity proof by a certified type inference procedure for T. In the last step we utilise the continuity theorem for System T from above to implement a plugin function called `ExtractModulus`. Given a functional, it reifies the functional into System T, infers typing information, employs the continuity theorem, and checks that the denotation of the System T representation is indeed the initial functional. The plugin can be called as `MetaCoq Run (ExtractModulus mod f)` where the modulus of `f` is saved as the definition `mod` together with a proof that it indeed is the modulus of continuity.

3 Future Directions

We see the current implementation as ground for further investigations in the extraction of continuity information in Coq. In the current state, the plugin has hardly any for practical applications. To make it useful in applications like computable real analysis [5] we want to extend to more base types in System T like $\mathbb{B}$, sums, pairs, lists, or rational numbers.

Furthermore, Coq functions are mostly defined using a match/fix representation of recursion instead of a recursor, which our reification can not yet deal with.

MetaCoq allows users to verify plugins in principle. For our plugin, this would mainly be a verification of the reification function, which would be eased by relying on the verified type inference function for Coq [4].

Lastly, we would like to extend the continuity notion and proofs to functions not expressible in System T to a larger fragment of Coq's type theory [6].

References

[1] M. Escardó. Continuity of Gödel's System T definable functionals via effectful forcing. *Electronic Notes in Theoretical Computer Science*, 298:119–141, 2013.

[2] Y. Forster and F. Kunze. A Certifying Extraction with Time Bounds from Coq to Call-By-Value Lambda Calculus. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:19. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[3] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter. The MetaCoq Project. June 2019.

[4] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter. Coq Coq correct! verification of type checking and erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, 4(POPL):8, 2019.

[5] F. Steinberg, L. Théry, and H. Thies. Quantitative Continuity and Computable Analysis in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:21. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

[6] C. Xu. A syntactic approach to continuity of T-definable functionals, 2019.