Practical Sized Typing for Coq

Jonathan Chan

Proof assistants based on dependent type theory rely on the termination of recursive functions and the productivity of corecursive functions to ensure two important properties: logical consistency, to disallow proving false propositions; and decidability of type-checking, to allow checking that a program proves a given proposition.

In the proof assistant Coq, termination and productivity are enforced by a *guard predicate* on fixpoints and cofixpoints respectively. For fixpoints, recursive calls must be *guarded by destructors*; that is, they must be performed on structurally smaller arguments. For cofixpoints, corecursive calls must be *guarded by constructors*; that is, they must be the structural arguments of a constructor.

The actual implementation of the guard predicate extends beyond the guarded-by-destructors and guarded-by-constructors conditions to accept a larger set of terminating and productive functions. In particular, function calls will be unfolded (i.e. inlined) in the bodies of (co)fixpoints as needed before checking the guard predicate. This has a few disadvantages: firstly, the bodies of these functions are required, which hinders modular design; and secondly, the (co)fixpoint bodies may become very large after unfolding, which can decrease the performance of type-checking. Furthermore, subtle syntactic changes in functions unfolded by (co)fixpoints can cause the guard predicate to wrongly reject the program even if the functions still behave the same. The dependence of guardedness checking on the structure of functions external to a (co)fixpoint can lead to difficulty in debugging, especially for larger programs.

An alternative to guard predicates for termination and productivity enforcement uses *sized types*. In essence, (co)inductive types are annotated with a size annotation, which can either be some size variable or the successor of another size annotation, like an arithmetic with only zero and addition by one. If some object has a type with size s, then the object wrapped in a constructor would have a type with successor size $\hat{s}$.

Termination- and productivity-checking is then simply a type-checking rule that uses size information. For termination, the type of the recursive call must have a smaller size than that of the outer fixpoint; for productivity, the outer cofixpoint must have a larger size than that of the corecursive call. This ensures that fixpoints act on ever-smaller objects until reaching a base case, and that cofixpoints produce ever-larger objects.

With this type-based method, (co)fixpoints only need the type of a function where it formerly would have needed to unfold the function body. Additionally, the syntactic form of the function would have no effect on its type and thus no effect on the overall typeability of the (co)fixpoint. Some (co)fixpoints preserve the size of arguments in ways that aren't syntactically obvious may be typed to be sized-preserving, allowing them to be called inside of (co)recursive calls and expanding the set of terminating and productive functions that can be accepted.

Past work on sized types in the Calculus of (Co)Inductive Constructions (CIC) such as CIC$\widehat{\ }$ [1] and CIC$\widehat{_}$ [2] describe type systems where sizes can be inferred and are never provided by the user in much the same way other type systems might perform type inference. However, they have some practical issues:

- They require nontrivial additions to the language, making existing Coq code incompatible without adjustments that must be made manually. These include annotations that mark the positions of (co)recursive and size-preserved types, and polarity annotations on (co)inductive definitions that describe how subtyping works with respect to their parameters.
- They require the (co)recursive arguments of (co)fixpoints to have literal (co)inductive types. With dependent types, it is possible to write expressions that evaluate to a type and to use these expressions where one would use types. However, while these works present dependently-typed systems, they disallow argument types being expressions that might otherwise evaluate to (co)inductive types.
- They do not specify how global definitions should be handled. Ideally, size inference should be done on each function independently for performance reasons.

In this project, we have designed an extension of CIC$\widehat{\ }$ called CIC$\widehat{*}$ that resolves these issues without requiring any changes to the surface syntax of Coq. We have also designed and implemented a size inference algorithm based on CIC$\widehat{*}$ within Coq's kernel that allows for termination- and productivity-checking of existing Coq code using sized types instead of guardedness checking [1]. In our implementation, we are easily able to accept terminating (co)recursive functions such as `quicksort` which would otherwise require user-provided proofs under guardedness checking only.

References

[1] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. "CIC$\widehat{\ }$: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Miki Hermann and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 257–271. ISBN: 978-3-540-48282-6.

[2] Jorge Luis Sacchini. "On type-based termination and dependent pattern matching in the calculus of inductive constructions". Theses. École Nationale Supérieure des Mines de Paris, June 2011. URL: https://pastel.archives-ouvertes.fr/pastel-00622429.

[1] https://github.com/ionathanch/coq/tree/dev