Generic Programming and Proving for Programming Language Metatheory

Adam Chlipala
University of California, Berkeley
adamc@cs.berkeley.edu

June 16, 2007

Practical generic programming has been in vogue lately, especially with the realization that common extensions to Haskell facilitate it nicely. The *scrap your boilerplate (SYB)* family of techniques [LJ03] is probably the most well-known example. SYB achieves genericity through a combination of ad-hoc code generation at compile time and "unsafe" type coercions at run time. Unfortunately, these coercions are at odds with formal, type-theoretical theorem-proving, having no ready semantic interpretation that isn't just a front for syntax. In the course of a project where we used Coq to develop a certified compiler for a statically-typed functional language [Chl07], we developed a new tool, AutoSyntax, that is implemented in the Calculus of Inductive Constructions (CIC) and is thus suitable for use with formal theorem proving.

We also found ourselves looking for a new category of functionality: *generic proofs* about our generic programs. Some past work has considered this problem, but every treatment that we have seen deals only with proofs about operations that must work over any representatives of broad classes of inductive types. Here, we are interested in generic functions that only work on a very narrow class of datatypes: those that are GADT-style definitions of the syntax and typing rules of programming languages. We present an approach for not only generic programming of operations like substitution and free variable set calculation over such abstract syntax trees, but also an approach for statically-verified generation of proofs about the interaction of those syntactic operations with arbitrary compositional *denotational semantics* of programs, implemented in CIC.

Our implementation strategy is based on *proof by reflection* [Bou97]. We define a programmatic representation of language definitions in CIC, allowing us to write functions that operate on these definitions. As programs and proofs are unified in one syntactic class in CIC, this same mechanism enables both. Coq's static type system guarantees that every generic function produces well-typed terms; additionally, the dependent types assigned to generic operations guarantee that their outputs are type-compatible with corresponding input languages.

Our implementation is available online with documentation at:

`http://ltamer.sourceforge.net/`

References

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Proc. STACS*, pages 515–529, 1997.

[Chl07] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *Proc. PLDI*, June 2007.

[LJ03] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proc. TLDI*, January 2003.