Modules in Type Theory with Generative Definitions

Jacek Chrząszcz

(extended abstract of the doctoral thesis)

The thesis presents an ML-style module system for a proof assistant based on type theory. In the described formalism modules are named entities and therefore well-suited to contain generative definitions such as inductive types and definitions by rewriting. It is shown that adding named modules to a pure type system with generative definitions does not compromise the system's consistency and decidability of type-checking.

The module system presented in the thesis has been successfully implemented by the author in the released version 7.4 of the Coq proof assistant [5]. Coq is based on the Curry-Howard isomorphism, its logical formalism is the calculus of inductive constructions [6] extended with predicative universes hierarchy [7]. It is actively developed by INRIA Rocquencourt (France) in collaboration with several European research groups. Until version 7.3, Coq lacked the possibility to conveniently define parameterized theories or parametrized certified data structures. Modules, borrowed from the ML family of programming languages, perfectly solve this problem. As the theoretical results of the thesis show, the future planned extension of Coq with rewriting will not conflict with modules.

The conservativity of the modular extension is shown under the assumption that consistency of a calculus with generative definitions means that no closed formal development proving `False` exists. The thesis presents a procedure transforming any closed development (with modules) into a closed development without modules, therefore relating consistency of the calculi with and without modules.

The procedure has two phases: first, all module components of the formal development are *evaluated* to a weak-head normal form using the call-by-value evaluation strategy; and second, the resulting evaluated development is *flattened*, i.e. components of modules are promoted to the toplevel, and modules themselves removed.

Since the module language itself forms a typed lambda calculus, termination of the first phase is not immediate. It is shown by translating the module expressions to a weak version of the module calculus of Courant and relating the evaluation steps with reductions between translated expressions. Since the calculus of Courant is terminating, so is the evaluation of named modules.

Proving the correctness of the second phase requires showing that module typing rules (subtyping in particular) are not essential when typing terms. A simplified calculus, called *principal*, is defined for that purpose. It contains only typing rules for terms and rules to extract term information from the environment. It is shown that term judgments derivable in both calculi are the same, a result which can also be interpreted as soundness of module subtyping. In the principal system, showing that module flattening preserves typing is an easy induction on the derivation.

The relative decidability proof is based on the observation that type preservation for flattening works in both directions: a flattened term is typable in a flattened environment if and only if the original term is typable in the original environment. Since the flattening operation itself has polynomial complexity, this gives us a polynomial reduction of the problem of typing terms in an arbitrary environment to the typing problem for the principal system in an environment without modules. And since derivations in the latter system correspond to derivations in a PTS with generative definitions, the desired relative decidability result follows.

It is also proved that the module language has the principal typing property, which implies decidability of the module typing itself as well, again, under the assumption that typing in the base PTS is decidable.

The complications resulting from considering rewriting are due to the fact that in order to assure that a (consistent and decidable) pure type system extended with rewriting is consistent and decidable, it is necessary to restrict the class of rewriting systems acceptable in a correct environment. Therefore the existence of an *acceptance condition* is assumed, which acts as a guard of the environment's correctness: every rewriting system is verified against this condition before being added to the environment and in return the acceptance condition guarantees that type-checking in the correct environment is decidable.

The acceptance condition is not fixed in the thesis. Instead five closure properties are given, which have to be satisfied by such a condition in order to make the proofs work. These properties may be summarized as modularity and substitutivity of names by names and they are compatible with syntactic acceptance conditions studied in the literature [1, 3, 2, 9]. One chapter

of the thesis contains a detailed discussion of the closure properties and a proposition of an acceptance condition satisfying them.

The main practical result of this thesis is the implementation of the module system in the Coq proof assistant. Apart from implementing a module type-checker in the kernel of Coq, a considerable implementation effort has been done towards the modularization of extra-logical features of Coq such as databases for automated tactics, customizable parsing and pretty-printing, etc. Part of this thesis describing implementation of modules in Coq has been accepted for presentation at the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs) in Rome, Italy, 9-12 September 2003 [4].

Modules are included in the version 7.4 of Coq, officially released in February 2003. Since then, several developments using modules have been realized, proving their usefulness in proof development. Pierre Casteran, Jean-Christophe Filliâtre and Pierre Letouzey showed formalizations of different versions of finite binary trees. These data structures, implementing dictionaries and finite sets, are presented as a functor taking an element type equipped with a partial ordering. Another interesting example of using modules is KRAKATOA [8], a Coq front-end to certify JAVA programs.

References

[1] Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.

[2] Frédéric Blanqui. *Théorie des Types et Récriture*. PhD thesis, Université Paris-Sud, 2001.

[3] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.

[4] Jacek Chrząszcz. Implementation of modules in the Coq system. In D. Basin and B. Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286, Rome, Italy, September 2003.

[5] The Coq proof assistant. `http://coq.inria.fr/`.

[6] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[7] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[8] Claude Marché, Christine Paulin, and Xavier Urbain. The Krakatoa tool for JML/Java program certification. Available at `http://krakatoa.lri.fr`, 2003.

[9] Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions*. PhD thesis, Warsaw University and University Paris XI, 2003.