Type refinement in $\lambda\Pi$-calculus modulo

Arnaud Spiwack & Olivier Hermant
arnaud@spiwack.net olivier.hermant@mines-paristech.fr

Mines ParisTech

Dedukti [9] is an experimental language designed to write proof checkers for logics. It is used to implement independent proof checkers for proof assistants such as Coq [11] and Isabelle [8] and trace checkers for automated deduction tools such as Zenon [3] and iprover modulo [4].

The logic underlying Dedukti is called $\lambda\Pi$-modulo [5]: it is an extension of $\lambda\Pi$, the simplest form of dependently typed λ-calculus (called λP in [2]), with user-defined rewriting rules for type equality (a.k.a. conversion).

Goals and directions of the internship

Writing a checker in Dedukti can be split in two phases: developing the theory of the target tool, and writing a translator from the tools syntax to Dedukti's. This internship aims at making the former task, which amounts to writing programs in Dedukti, more tractable.

Dedukti is organised in a way which is reminiscent of proof assistants. It is split into a small kernel, which is tasked with verifying that a term with type annotations is well-typed, and a front-end which takes the program as written by the user and elaborates it into a term which the kernel understands. In principle, the user-level language may differ significantly from the kernel-level language. The front-end is, for instance, usually expected to perform some type inference, whereas it is not the rôle of the kernel. In Dedukti, however, the front-end stops just short of being the identity: the programmer essentially uses the kernel language directly.

To remedy this unfortunate situation, Dedukti needs to be outfitted with a type inference engine. Type inference in dependently typed language fuses with unification and proof search. The main inspiration for such a type inference engine is the proof-search engine used by Coq [10, Chapter 4]. This proof-search engine is meant to be a safe and universal abstraction to manipulate terms with holes. The large amount of pre-existing code in Coq meant, however, that only interactive demonstrations use this abstraction, and specifically not the type inference or unification mechanisms. We would like to take advantage of the experimental status of Dedukti and improve on this design by

actually expressing type inference and unification in terms of the proof-search abstraction.

This line of research contains both theoretical and practical programming aspects. The internship can be geared towards on or the other depending on the student's preferences.

On the programming side, the proof-search abstraction must be integrated with Dedukti's kernel by means of enriching the syntax of terms with so-called existential variables – also known as unification variables or meta-variables. This change must be made with as little disruption as possible on the kernel side. The design should improve on Coq's by providing two kinds of existential variables: the regular ones and existential variables standing for contexts. This new kind of existential variable was introduced by Lengrand [7, Chapter 9] for theoretical reason pertaining to unification and saw other successful application in the Matita proof assistant [1]. Another aspect to consider is that Coq's abstraction was done with the knowledge that unification was handled somewhere else, and hence features no abstraction for unification problems, which will have to be developed as part of this work.

On the theoretical side, the $\lambda\Pi$-calculus modulo needs a good theory of unification expanding on the work of Dowek [6] for pure type systems. Higher-order unification is undecidable in the case of simply typed λ-calculus. It is even more so in the very expressive $\lambda\Pi$-calculus modulo. This prevents us from achieving a decision procedure for unification, but we can devise a set of deduction rules which, provided an oracle, solves unification. The addition of rewriting rules in the conversion the $\lambda\Pi$-calculus modulo changes significantly the shape of normal forms with respect to pure type systems. A good understanding of these normal form must be achieved, in order to derive an appropriate set of rules to describe unification strategies in the $\lambda\Pi$-calculus modulo.

Practical information

The internship will be supervised by Arnaud Spiwack and Olivier Hermant. It will be hosted by the Deducteam project-team at Inria Paris-Rocquencourt and the CRI at Mines ParisTech.

References

[1] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, 8(1):1–50, March 2012.

[2] Henk Barendregt. Lambda calculus with types. *Handbook of logic in computer science*, 1992.

[3] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. *Logic for Programming, Artificial Intelligence, and Reasoning*, 2007.

[4] Guillaume Burel. Embedding Deduction Modulo into a Prover. *Computer Science Logic*, 6247, 2010.

[5] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. *Typed Lambda Calculi and Applications*, 2007.

[6] Gilles Dowek. A Complete Proof Synthesis Method for the Cube of Type Systems. *Journal of Logic and Computation*, 3(3):287–315, 1993.

[7] Stéphane Lengrand. *Normalisation & equivalence in proof theory & type theory*. PhD thesis, Université Paris VII – Denis Diderot & University of St Andrews, 2006.

[8] Larry Paulson, Tobias Nipkow, and Makarius Wenzel. Isabelle.

[9] Ronan Saillard. Towards explicit rewrite rules in the $\lambda\Pi$-calculus modulo. *IWIL-10th International Workshop on the Implementation of Logics*, pages 1–5, 2013.

[10] Arnaud Spiwack. *Verified Computing in Homological Algebra: A Journey Exploring the Power and Limits of Dependent Type Theory*. PhD thesis, École Polytechnique, 2011.

[11] The Coq development team. The Coq Proof Assistant. http://coq.inria.fr/.