The static debugger: classical realizability rescuing the programmer

Lionel RIEG

Software certification aims at proving the correctness of programs but in many cases, the use of external libraries allows only a conditional proof : the correctness of the program depends on the assumption that the libraries meet their specifications. In particular, a bug in these libraries might still impact the certified program. In this case, the difficulty that arises is to isolate the defective library function and provide a counter-example to its specification. We propose one effective solution to this problem using Krivine's classical realizability [Kri09] and classical extraction from Coq.

Formalization of the problem Given the specification V of the program and U of the unsafe external components on which it depends, the certification of the program is a formal proof of $\vdash U \Rightarrow V$. Both formulæ U and V contain abstract predicates denoting the behavior of the external components and so are not arithmetical. The bug report proving the misbehavior of the global program is an experimental counter-example for V, which we denote by $\not\models V$. The problem we address is then summarized in the following pseudo-inference rule :

$$\frac{\vdash U \Rightarrow V \qquad \not\models V}{\not\models U}\ \text{Experimental Modus Tollens}$$

It is not a formal inference rule because it combines objects of two different natures : a formal proof $\vdash U \Rightarrow V$ and a experimental evidence of misbehavior of the program $\not\models V$. Counter-examples only make sense for universal formulæ so we shall restrict U and V to be so.

Without loss of generality, we can restrict this rule to the case where V is the false formula [Miq09b, p.100]. With this restriction, it becomes the problem of finding a explicit counter-example to a contradicting theory. Notice that this problem is more than the completeness theorem because we want an explicit counter-example.

To build a explicit counter-example, we will need the interpretation of the abstract predicates, *i.e.* we will need to test the implementations of the external components the program uses. This is the idea of the solution presented by Miquel in [Miq09b]. Although we cannot completely remove tests because the very counter-example we seek depends on them (they are the only available source of information about the external predicates), we would like to avoid them as much as possible because they can be expensive to perform. One intermediate solution is to abstract these tests and consider all possible interpretation of the external predicates. Instead of a single counter-example, this will lead to a family of counter-examples that can be organized in the form of a Binary Decision Diagram, which is logically nothing more than a Herbrand tree. Their existence is the very statement of Herbrand's theorem :

Theorem :

If U is a contradicting universal theory, then it admits a Herbrand tree.

whose hypothesis can be rephrased using the completeness theorem as "for all interpretation $\mathscr{M}$, $\mathscr{M}$ is not a model of U" : $\forall \mathscr{M}, \neg(\mathscr{M} \models U)$.

Solution : extract Herbrand's theorem The idea to build Herbrand trees is to extract a program from a proof of Herbrand's theorem formalized in Coq. Since its proof is classical, we need to use classical extraction for Coq [Miq09a] and to be able to extract a real witness from the extracted realizer [Miq10].

In order to ease the proof of Herbrand's theorem in Coq, we transform its statement into one suitable for Coq. Instead of working with the full syntax containing abstract predicates, it is simpler to abstract all atomic formulæ in one data type atom. With this representation, quantifier-free formulæ are simply the elements of the Boolean algebra generated over these atoms ; a data type we call compound. We also replace the universal formula U by the abstract data type index, an indexing of its ground instances together with a decoding function Th : index $\to$ compound translating a ground instance of U into a quantifier-free formula. The precise statement of the theorem proved in Coq is the following :

$$\begin{array}{ll} \forall\, \text{atom}, & \\ \forall\, \text{index}, & \forall\, \text{Th} : \text{index} \to \text{compound}, \\ & \Big(\forall val : \text{atom} \to \text{Prop}, \neg(\forall i : \text{index}, \text{eval}\, val\, (\text{Th}\, i))\Big) \\ & \to \exists\, t : \text{tree}, \text{Htree}\ \ \text{Th}\ \ t = \mathit{true} \end{array}$$

where Htree is a decision function for the predicate "t is a Herbrand tree for the theory denoted by Th".

Proof One very important restriction on the formalized proof is that it must be extractable, so we must use only axioms we know how to realize in Krivine's classical realizability. Apart from excluded middle in Prop (for classical logic) realized by Pierce's law, the only axiom we use is proof irrelevance which is realized by the identity.

Witness extraction also requires that the shape of the extracted formula be $\exists t, f(t) = a$ with a a constant and f a decidable function, which justifies the Coq statement of the theorem and the use of decidable (*i.e.* Boolean) equalities and orders in the proof.

The idea of the formalized proof is based on the usual proof of Herbrand's theorem in mathematics but rather than using Kőnig's lemma, we "inline" its proof, by building an infinite branch. The existence of a Herbrand (sub-)tree is reflected in an inductive predicate and, assuming the absence of a Herbrand tree, builds a model of U, thus contradicting the hypothesis.

Classical extraction The classical extraction technique is based on Miquel's extension to CIC_ω [Miq07] of Krivine's classical realizability for PA2 [Kri09]. Since their second order fragments are isomorphic, terms extracted from Coq are valid realizers in the sense of PA2 realizability and we can pretend to work in this simpler setting, hiding the more complicated CIC_ω classical realizability model.

Proof terms are extracted to the λ_c-calculus, an extension of call-by-name λ-calculus with the call/cc control operator realizing Pierce's law [Gri90]. Evaluation is performed within the Krivine's Abstract Machine (KAM).

Realizer optimization The extracted realizers are not always very efficient so we may want to replace some of them with more efficient ones. There is two main options to do so, the first (and heaviest) one is to change the extracted representation of one data type. It has been successfully used to change the Peano's unary integers into binary integers using an arbitrary precision library, which leads to space and time savings.

The second kind of optimization directly uses the classical realizability theory and the simple realizers it provides for true formulæ of a given shape, like for instance equalities or implications between equalities which are realized by the identity. Thus, any theorem of the correct shape can be realized by this very simple realizer. This realizer substitution leads to huge performance increases, at the scale of several complexity orders of magnitude because it can replace recursive realizers with constant-time ones.

Conclusion This work is the first real use of classical realizability to extract inductive data type, trees (in fact, it is the first example not to extract integers). The extracted trees strongly depend on the inconsistency proof of U, thus this work directly embodies the Curry-Howard correspondence extended to classical logic. Nevertheless, the precise computation behavior of this method still remains to be completely understood. On the long run, optimization is a promising direction for classical realizability that needs further investigation.

Références

[Gri90] Timothy Griffin. A formulae-as-types notion of control. In *POPL*, pages 47–58, 1990.

[Kri09] Jean-Louis Krivine. Realizability in classical logic. In *Interactive models of computation and program behaviour*, volume 27 of *Panoramas et synthèses*, pages 197–229. Société Mathématique de France, 2009.

[Miq07] Alexandre Miquel. Classical program extraction in the calculus of constructions. In *Computer Science Logic (CSL)*, pages 313–327, 2007.

[Miq09a] Alexandre Miquel. *The classical extraction module for Coq*, 2009. Available at `http://perso.ens-lyon.fr/alexandre.miquel/kextraction/`.

[Miq09b] Alexandre Miquel. De la formalisation des preuves à l'extraction de programmes. HdR thesis, Université Paris 7, December 2009.

[Miq10] Alexandre Miquel. Existential witness extraction in classical realizability and via a negative translation. In *Logical Methods in Computer Science (LMCS)*, 2010. to appear.