Rodolphe Lepigre | RESEARCH STATEMENT
(+33) 06 95 42 73 30 • rodolphe.lepigre@inria.fr • lepigre.fr

Theoretical and Practical Investigations for an ML-Like Language Supporting Program Proofs

The **safety of computer systems** is one of the major issues of the twenty-first century, as witnessed by countless malfunctions reported in the media. The risks are diverse, ranging from security or privacy concerns (e.g., with autonomous vehicles or connected objects) to economic losses (e.g., with service unavailability). To prevent many such issues, a great number of tools have been developed. They include, for example:

– the certified C compiler CompCert [Leroy, 2009],
– the resource management system of the Rust language,
– the GADTs of OCaml or Haskell [Peyton Jones et al., 2006].

These tools give **safety guarantees by construction**, for example the preservation of the semantics of a program by compilation in the case of CompCert. Nonetheless, software certification (i.e., proving that a software satisfies its specification) requires the use of **formal methods**, which are generally little known in the software industry. They involve the use of a *proof assistant* such as Coq or Isabelle, in which the software and its specification must be modeled, and then their compatibility established. This process is extremely costly, and it requires specialized qualifications, which are not those of a programmer. Moreover, this method is not modular, in the sense that every modification of the software (or of its specification) has important consequences on the certification.

Unifying programming and proving

My main research goal is to provide powerful tools for **certifying programs in a simple and modular way**. I advocate a new approach to program proofs, based on a notion of *program equivalence*, and involving *subtyping* and *Curry-style quantifiers*. I already explored these design choices in my PhD thesis [Lepigre, 2017a], and they proved to be both practical and convenient to use (at least on small scale examples). One of the interests of this approach is that it can be presented as an **extension of an ML-like language** (like OCaml or SML for example). We thus obtain a language that can be used both as a standard, *functional programming language* and as a *proof system*. Such a uniform language allows programmers to choose the level of guarantees that they want to enforce, and it is always possible to give more precise specifications later if the requirements evolve. For instance, a developer can first quickly implement a prototype software, taking advantage of the *type system* and (possibly) *certified libraries*. Then, the developer may decide to invest more time to **progressively obtain more guarantees**, until the software has be partially or completely certified.

Another advantage of having a single language for writing programs and for proving their properties is that this **eliminates intermediate representations**. Indeed, this directly allows the compilation of a program (provided that we implement a compiler), be it certified or not, without having to rely on an extraction mechanism (that would be required in Coq, for example). In other words, programs written in the system are at the same time the *sources of the compilation* to machine code, and the *object of the certification*. Moreover, having a **uniform syntax** yields a system that is relatively easy to use since it only involves a single language.

Previous work: achievements so far

Several first steps toward the goals presented in the previous section have already been achieved. I developed a new **classical realizability** framework, that is compatible with the notion of *observational equivalence* [Lepigre, 2016]. It lays the foundation of the PML_2 language presented in my PhD thesis [Lepigre, 2017a] and in a submitted paper [Lepigre, 2017b]. This work justifies the use of **equational**

reasoning to specify and prove program properties in the context of an effectful language (with a *control operator* similar to Scheme's *call/cc*).

I also developed, in a joint work with Christophe Raffalli, a set of methods for dealing with so-called Curry-style languages [Lepigre and Raffalli, 2017]. Curry-style language such as PML_2 are hard to implement because their type systems is usually not *syntax-directed*. However, we showed that they can be reformulated in a mostly syntax-directed way using our notion of **local subtyping**. Moreover, we showed that this approach is compatible with many usual type formers, including existential type, inductive types and coinductive types. The latter two are handled using a general notion of **well-founded circular proof**, which can also be used to check the termination of recursive programs. Finally, we rely on **symbolic witnesses** to simplify the formulation of our *realizability semantics* and the implementation. Indeed, this amounts to working with only closed terms, and this implies that we only need to use *first-order unification*. This work has been implemented in the SubML language, and also adapted to PML_2 in my thesis [Lepigre, 2017a].

Last but not least, we developed (again with Christophe Raffalli) a set of tools for easing the implementation of languages and proof systems in OCaml. They include the Bindlib library for **variable binding** [Lepigre and Raffalli, 2018], the Earley **parser combinator library**, and the Timed library for imperative state management (all available on Opam). I am using these tools to provide a **new implementation of the Dedukti logical framework** called Lambdapi, which is being developed in the context of my current postdoctoral position at Inria.

Future work and developments

The PML_2 language and its semantics does not yet support the **imperative programming style**, in the sense of the mutation of a global state. This feature is however essential to ML programmers, and it is widely used in OCaml for example. The main problem to extend the language with *references* (mutable variables) is to incorporate them at the level of the realizability model. Indeed, the management of **references in the context of classical realizability** remains an open problem, even if the work of Krivine indicates that *forcing* could be a solution [Krivine, 2011]. First steps in this direction have been made by Brunel [Brunel, 2014] and by Herbelin and Miquey [Herbelin and Miquey, 2018], and I would very much like to pursue this line of work. A possible perspective could be the development of (classical) realizability semantics for ownership type systems.

Another important research direction that I would like to pursue is to apply the techniques that were developed in [Lepigre and Raffalli, 2017] to other systems, and in other contexts. In particular, it should be possible to apply our circular proof framework in the context of the *modal μ-calculus*, or any other **system containing fixed-points**. I would also like to explore the connections between our circular proof framework (based on the *size change principle* [Lee et al., 2001]), and the **infinitary proof systems** found in the work of Baelde, Doumane and Saurin [Baelde et al., 2016]. It would also be interesting to relate the generalized notion of size-change principle of Hyvernat [Hyvernat, 2014] with our circular proof system. Preliminary work seems to indicate that a specific *unfolding* of our circular proofs allows to recover the full power of his extension. However, it is not yet clear how to systematically decide when cycles should be unfolded.

Finally, I would like to **extend the PML_2 language with a compiler**, which is an essential milestone for the system to achieve its main purpose. To develop the compiler, the LLVM pseudo-assembly language seems to be an ideal target to obtain many standard optimizations for free. Nonetheless, the specificities of PML_2 will certainly allow for novel optimization techniques. For example, it should be possible to eliminate *dead code* by proving that it is inaccessible, or to optimize *allocation* in relation to subtyping. Another possible target could be one of the **intermediate representations used in the CompCert project** [Leroy, 2009], which would be a fist step toward a certification of PML_2, before PML_2 can be *bootstrapped* (i.e., developed, and at least partially certified in PML_2).

References

[Baelde et al., 2016] Baelde, D., Doumane, A., and Saurin, A. (2016). Infinitary proof theory: the multiplicative additive case. In *CSL*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Brunel, 2014] Brunel, A. (2014). *The monitoring power of forcing program transformations.* PhD thesis, Paris 13 University, Villetaneuse, Saint-Denis, Bobigny, France.

[Herbelin and Miquey, 2018] Herbelin, H. and Miquey, É. (2018). Realizability interpretation and normalization of typed call-by-need \lambda -calculus with control. In *FoSSaCS*, volume 10803 of *Lecture Notes in Computer Science*, pages 276–292. Springer.

[Hyvernat, 2014] Hyvernat, P. (2014). The size-change termination principle for constructor based languages. *Logical Methods in Computer Science*, 10(1).

[Krivine, 2011] Krivine, J. (2011). Realizability algebras: a program to well order R. *Logical Methods in Computer Science*, 7(3).

[Lee et al., 2001] Lee, C. S., Jones, N. D., and Ben-Amram, A. M. (2001). The size-change principle for program termination. In *POPL Proceedings*, pages 81–92. ACM.

[Lepigre, 2016] Lepigre, R. (2016). A Classical Realizability Model for a Semantical Value Restriction. In Thiemann, P., editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer.
`https://lepigre.fr/files/docs/lepigre2016_svr.pdf`.

[Lepigre, 2017a] Lepigre, R. (2017a). *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes)*. PhD thesis, Université Grenoble Alpes.
`https://lepigre.fr/files/docs/phd.pdf`.

[Lepigre, 2017b] Lepigre, R. (2017b). The PML$_2$ Language: Proving Programs in ML. Submitted for the post-proceedings of the TYPES 2017 conference (awaiting decision on minor revision).
`https://lepigre.fr/files/docs/lepigre2017_pml2.pdf`.

[Lepigre and Raffalli, 2017] Lepigre, R. and Raffalli, C. (2017). Practical Subtyping for Curry-Style Languages. Submitted for the TOPLAS journal (awaiting decision on minor revision).
`https://lepigre.fr/files/docs/lepigre2017_subml.pdf`.

[Lepigre and Raffalli, 2018] Lepigre, R. and Raffalli, C. (2018). Abstract representation of binders in ocaml using the bindlib library. In Blanqui, F. and Reis, G., editors, *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, Oxford, UK, 7th July 2018*, volume 274 of *Electronic Proceedings in Theoretical Computer Science*, pages 42–56. Open Publishing Association.
`https://lepigre.fr/files/docs/bindlib.pdf`.

[Leroy, 2009] Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM (CACM)*, 52(7):107–115.

[Peyton Jones et al., 2006] Peyton Jones, S. L., Vytiniotis, D., Weirich, S., and Washburn, G. (2006). Simple unification-based type inference for gadts. In *ICFP*, pages 50–61. ACM.