AGENCE NATIONALE DE LA RECHERCHE
ANR
AAPG-ANR-2019 | REPRO | Programme JCJC
Cordonné par: Gabriel Scherer | 4 years, € 227,000
CES 48 - Fondements du numérique : informatique, automatique, traitement du signal

REPRO: Searching for Canonical **Re**presentations of **Pro**grams

Principal Investigator (PI): Gabriel Scherer, *Chargé de recherche* at INRIA

1 Scientific context, positioning and objectives

The REPRO project aims to

1. deepen our understanding of the structure of computer programs by discovering *canonical representations* for fundamental programming languages, and to
2. explore the application of canonical representations to the problems of *program equivalence checking* and *program synthesis*.

Arithmetic (sum and product) expressions in one variable x, such as $(1+2*x)*(1+x)$, *represent* a certain category of functions from real numbers to real numbers. This representation admits redundancies, in the sense that distinct expressions, such as $(a+b)*x$ and $a*x+b*x$, may represent the same function. On the other hand, *polynomials*, which can be described formally as either 0 or a non-empty sum of the form $\sum_{0 \leqslant k \leqslant d} c_k x^k$ with $c_d <> 0$, are a canonical representation of product/sum functions in one variable: if two polynomials have distinct coefficients, they represent distinct functions.

A canonical representation has more structure: its definition encodes more of the meaning of the objects being represented. For example, it is non-obvious whether an arithmetic expression is constant (for all values of x), while this question can be easily decided for polynomials: it is constant when it is 0 or its degree is 0. In general, polynomials are much more convenient to manipulate for virtually any application. They are the ubiquitous choice of representation when studying these objects.

Little is known about the canonical representation of computer programs. Representing programs by their source code admits many redundancies: the sources may be different in inessential ways (choice of variables, order of computations, but also important changes in program structure for example), which make no observable difference in the behavior of the executing program.

Program equivalence has been studied for formal, idealized programming languages known as λ-calculi. It strongly depends on the forms of computations available in the programming language studied, in particular the datastructures it can manipulate. In presence of only functions and finite products (pairs of values), canonical representations (called $\beta\eta$-normal forms) are known (Böhm, 1968). But adding finite sums (disjoint union datatypes, in particular booleans) gives a richer notion of program equivalence. Non-empty sums have been studied since the nineties (Ghani, 1995), providing partial proposals for canonical forms, and the case of general finite sums (including empty sums) was proposed as an important open problem to the programming language community in 1995.

In the 2010s, several groups (Ahmad, Licata, and Harper, 2010; Scherer and Rémy, 2015; Ilik, 2017) simultaneously developed the idea of using ideas of *proof theory*, which had made progress on the quest for canonical representations of proofs. A notion of particular interest is *focusing*, a discipline to reduce non-determinism during proof search by imposing more structure on proofs, providing canonical representations in some logics (Chaudhuri, Miller, and Saurin, 2008). By the Curry-Howard isomorphism between proofs and programs, ideas from logic can be transferred into the programming language community, and programming language problems can motivate new developments in proof theory. In Scherer and Rémy (2015) we extended focusing with a new notion of *saturation*, giving new canonical representations which had the additional property of being effectively searchable/enumerable at a given type. While we did not expect this as a side-result, those

enumerable canonical representations finally solved the open problem of deciding the equivalence in presence of general finite sums (Scherer, 2017).

These theoretical results only cover very idealized formal languages, but those are the basis of our mathematical descriptions of most real-world programming languages and systems. In particular, they are closest to richly typed, functional programming languages (Haskell, SML and OCaml for example), as well as proof assistants – languages for mathematical proofs. We will restrict our attention to those families of languages in the rest of this proposal.

We believe that we are only at the beginning of proof-theoretical approaches to canonical representations of programs. Promising applications of this work include automatically filling richly-typed program fragments, and automated equivalence checking. They would make it easier to write correct software, a major objective of programming language research.

We propose to push further, both in theory and in practice – each informing the other – with the following objectives for a four-year period:

1. Studying canonical representations in richer λ-calculi, to bridge the gap with functional languages and proof assistants used today. This is a deep, high-risk theoretical challenge.

2. Uncovering connections with other proof-theoretic approaches to the structure of proofs, such as *deep inference* and *combinatorial proof*. This is a medium-risk challenge that could connect those areas to programming language applications that were not on the radar before.

3. Developing practical tools for program equivalence checking in functional programming languages. This is a low-risk problem with a large potential for applications and user adoption.

4. Using canonical representations, in particular our *saturated normal forms*, to reduce the search space in *program synthesis* problems. This is a medium-risk challenge; our theoretical developments could bring large performance improvements to existing program synthesis systems.

The principal investigator comes from a programming-languages background, but works inside a group with a strong proof-theory expertise. To our knowledge, this general positioning is unique in the area (even at the international level); the approaches we suggest to programming-language applications have shown some early promises, but are not currently pursued in programming-languages groups due to lack of theoretical expertise. Conversely, proof-theory groups usually do not target applications to programming languages; we believe there is a strong potential for practical impact.

Richer type theories. While simply-typed systems capture the essence of basic datatypes, existing programming systems use more advanced types. *Polymorphic types*, which describe genericity and data abstraction, are the next frontier for canonical representations, and would let us describe most of today's functional, statically-typed programming languages such as OCaml, Haskell and Scala. From a proof-theory perspective, they correspond to second-order logic. Going even further, one would be interested in canonical representations of *dependent types*, used in many proof assistants (Coq, Lean, Agda, Idris, Cedille...), which could improve their local proof-search engines.

Studying these richer settings is a major challenge because we already know that equivalence-checking of programs at these types, as well as proof search in the correspond logics, are undecidable. Any proposal for canonical representations must be partial – fail to compute representations for some program. However, many of the applications we have in mind for canonical representations, such as equivalence checking or program synthesis, can accomodate partial procedures: they remain useful if they sometimes answer "I don't know" or time out.

New connections to logic. Proof nets and combinatorial proofs are an ongoing research programme proposing new representations of proof in classical and intuitionistic logics, aiming at canon-

icity (Hughes, 2018; Acclavio and Straßburger, 2018). We believe that comparing them to our approach to term canonicity, based on focusing and saturation, could put new connections to light – and possibly suggest areas of application of these fairly theoretical logics to programming languages. There is a medium risk that the sources of canonicity in combinatorial proofs are totally inapplicable to our programming-language setting, if they relies on irreversibly erasing computational information.

Practical equivalence checking. Some of the changes that programmers manually perform on a large codebase are "refactorings", reorganizations of the source code that should not change the program behavior but make future modifications and maintenance easier. From our experience on canonical forms and equivalence for typed, functional programming languages, we propose to build an automated equivalence checker that can reassure a programmer, at the press a button, that a code change is indeed a valid refactoring.

There is automated tooling support for certain atomic refactoring operations (rename variable, extract method...), and research to verify these atomic operations (Schäfer, Ekman, and de Moor, 2008). But many users refactor by manually editing the source, several related changes at a time, and they exchange their modifications as textual patches. We propose to check the equivalence of the original and patched programs. Equivalence checking is undecidable in real-world languages, but refactoring changes are designed to be simple and reviewable by humans, so they are more likely (than arbitrary instances) to be supported by partial algorithms.

Equivalence checking has many other practical applications (for example, after-the-fact validation of compiler optimizations, or cross-comparison of two compiler versions), so we can hope for some unplanned side-results of building a practical checker.

Saturation for synthesis. Program synthesis is about exploring a large search space of candidate programs, testing them against some sort of specification (input/output example pairs, a logic formula on their behavior, etc.). There is a growing interest in type-directed synthesis for functional programming language, and the experts in the field have (re)discovered the proof-theoretic notion of *focusing*, which reduces the synthesis search space by imposing additional structure on candidate programs (Frankle, Osera, Walker, and Zdancewic, 2016; Polikarpova, Kuraj, and Solar-Lezama, 2016). Their experimental results show that phase-ordering optimizations suggested by focusing can bring order-of-magnitude speedup, solving previously-infeasible synthesis problems.

Our objective is to apply our notion of *saturation*, which has stronger canonicity properties than focusing alone for pure functional programs, to reduce the program search space and improve synthesis performance. There is a real potential for impact with interesting program synthesis applications, including programming tools for non-specialists and partially-automated verified programming.

There is a tension in synthesis between being smart and fast: if a technique reduces the search space but slows down the production of candidate programs, it may not improve synthesis in practice. Saturation reduces backtracking, but leads to larger normal forms: there is a medium risk that it would turn out to be unusable for synthesis.

2 Partenariat

The principal investigator of the project, Gabriel Scherer, is CR at INRIA Saclay, in the Parsifal project-team. The project will span over four years and require one post-doc and one PhD student. Scherer's colleagues Lutz Straßburger and Dale Miller, also in project-team Parsifal, would be involved in the project; one of them (depending on the PhD student's main focus) would also co-supervise the PhD. Project funding would also help securing collaborations external to INRIA Saclay, with Guilhem Jaber (INRIA Rennes) and Nadia Polikarpova (University of California, San Diego).

Principal Investigator. Gabriel Scherer received his PhD in 2016 from the Paris-Diderot University (working at INRIA Rocquencourt), and spent 18 months as a post-doc at Northeastern University (Boston), before joining INRIA Saclay in July 2017. He has published in top conferences (ICFP, LICS, POPL) either alone, with direct colleagues from the same lab, or remote collaborators. His expertise is recognized, as witnessed by invitations to chair research workshops in his community (the OCaml Workshop and the ML Family workshop) and to serve in program committees (ICFP 2017, POPL 2018). Gabriel Scherer has a strong programming language expertise, both in theory and in practice (he is one of the maintainers of the OCaml programming language implementation).

Gabriel Scherer first started applying logical techniques during his PhD, and joined the Parsifal team at INRIA Saclay precisely to work with the strong proof-theory expertise of Parsifal. This JCJC project will be essential in opening collaborations within his team and institution, and provide an opportunity to fund and supervise research on his own research goals.

Collaborators. Our direct colleague Dale Miller is one of the leading experts on focusing in proof theory, and would be a natural collaborator for our first objective, “richer type theories”.

Our direct colleague Lutz Straßburger is one of the leading experts on combinatorial proofs, and would be a natural collaborator for our second objective, “New connection to logics”.

Natural collaborators for “Practical equivalence checking” would be (1) the “Trustworthy refactoring” group at Kent (UK) which builds software that we could reuse and build upon and (2) Guilhem Jaber (Université de Nantes), as an expert on semantic approaches to understand program equivalence in presence of non-termination, control operators and general references.

Finally, we are in contact with Nadia Polikarpova, one of the leading experts on program synthesis in rich type systems, and this would be an excellent opportunity for collaboration.

Funding, Open Access, IP, (no) data. We request €227,000 to fund one PhD scholarship (€110,000), a one-year postdoc (€65,000), a travel and equipment budget (€5,000 per person-year (8)), and budget to visit or host collaborators (€3,000 per project-year (4)).

We plan to distribute our preprints on HAL and release any softare prototype as free software, following common Open Access practices. The project will not involve building research databases or handling personal data of any kind.

3 Bibliographic references

M. Acclavio and L. Straßburger. From Syntactic Proofs to Combinatorial Proofs. In *IJCAR*, 2018.
A. Ahmad, D. R. Licata, and R. Harper. Deciding Coproduct Equality with Focusing. Unpublished, 2010.
C. Böhm. Alcune proprieta delle forme normali nel k-calcolo. *IAC Pubbl*, 696119, 1968.
K. Chaudhuri, D. Miller, and A. Saurin. Canonical Sequent Proofs via Multi-Focusing. In *IFIP TCS*, 2008.
N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.
G. Scherer. Deciding equivalence with sums and the empty type. In *POPL*, 2017.
G. Scherer and D. Rémy. Which simple types have a unique inhabitant? In *ICFP*, 2015.
J. Frankle, P. Osera, D. Walker, and S. Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.
N. Ghani. Beta-Eta Equality for Coproducts. In *TLCA*, 1995.
D. J. D. Hughes. Unification nets: canonical proof net quantifiers. In *LICS*, 2018.
D. Ilik. The Exp-Log Normal Form of Types. In *POPL*, 2017.
M. Schäfer, T. Ekman, and O. de Moor. Challenge Proposal: Verification of Refactorings. In *PLPV*, 2008.