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

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

Summary table of the research personnel involved in the project

Partenaire	Nom	Prénom	Position actuelle	Rôle et responsabilité dans le projet	Implication
INRIA Saclay	Scherer	Gabriel	CR	PI	48
INRIA Saclay	Miller	Dale	DR	"focusing"	4-6
INRIA Saclay	Straßburger	Lutz	CR	"combinatorial proofs"	4-6
Université de Nantes	Jaber	Guilhem	MCF	"equivalence and state"	4-6
UCSD (US)	Polikarpova	Nadia	Assistant professor	"program synthesis"	4-6

Changes between the pre-proposal and the detailed proposal

We corrected our financial data: the personnel costs are slightly lower (€162,000 instead of €165,000 in the pre-proposal), but the addition of environmental costs gives a slightly higher final number (€231,000 instead of €227,000 in the pre-proposal).

I Proposal context, positioning and objectives

a Objectives and research hypotheses

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.

To study general-purpose languages, programming language research works with λ-calculi, a family of very idealized formal languages, from the very simple to the very complex, that form 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: what is the most expressive λ-calculus for which we understand the canonical forms?

The best result in this direction comes from the Principal Investigator, with a description in Scherer (2017a) of canonical forms for the simply-typed λ-calculus with functions, finite products and finite sums. This calculus represents some core aspects of programming (functions and values, composite structures and disjoint choices), but it lacks any support for arbitrary-length data structures (inductive or recursive types) and for datatype abstraction (polymorphism).

We believe that it is possible to push further towards more powerful calculi supporting these key elements of real-world programming languages. Promising applications 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 the following objectives for a four-year period:

Richer type theories Studying canonical representations in richer λ-calculi, to bridge the gap with functional languages and proof assistants used today.

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, and thus make them more usable for type-rich programming.

This is a deep, high-risk theoretical challenge.

New connections to logic Uncovering connections with other proof-theoretic approaches to the structure of proofs, such as *deep inference* and *combinatorial proof*.

Proof nets and combinatorial proofs are an ongoing research programme proposing new representations of proof in classical and intuitionistic logics, aiming at canonicity (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.

This is a medium-risk challenge that could connect those areas to programming language applications that were not on the radar before.

Practical equivalence checking Developing practical tools for program equivalence checking in functional programming languages.

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.

This is a low-risk problem with a large potential for applications and user adoption.

Program synthesis Using canonical representations, in particular our *saturated normal forms*, to reduce the search space in *program synthesis* problems.

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.

This is a medium-risk challenge.

Interestingly, in richer type systems, the structure of canonical representations becomes harder to understand, but their fine-grained understanding has *more* direct applications. Indeed, richer types let programmers express finer-grained specifications as types: the more precise the type, the less inhabitants (programs breaking the specification are excluded), and thus the more precise structure of canonical forms. For example, type-directed program synthesis becomes useful more often, and refactoring changes more often preserve program equivalence – when the type of the two programs can rule out, for example, certain observation effects, or make some inputs abstract.

b Positioning wrt. the state of the art

Decision algorithms for program equivalence strongly depends on the forms of computations available in the programming language (the λ-calculus) 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, 2017a).

We believe that we are only at the beginning of proof-theoretical approaches to canonical representations of programs (Scherer, 2017b). 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.

c Methodology and risk management

1 Richer type theories

Studying canonicity in richer type theories 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 accommodate partial procedures: they remain useful if they sometimes answer “I don’t know” or time out.

The richer type theories we need to study in the long term are: polymorphism (System F), inductive types (inspired by logics with fixpoints), and eventually dependent type. For this project we would like to focus on polymorphism, for which we have a specific approach in mind.

Approach When a variable of polymorphic type $\forall\alpha.T(\alpha)$ occurs while exploring the space of canonical programs at a given type, the problem is to determine how to instantiate this variable – for which A are we interested in the instantiated types $T(A)$ to use in the rest of the proof. There is an infinite space of potential candidates, including $\forall\alpha.T(\alpha)$ itself (we are studying *impredicative* polymorphism).

Technically, we say that derivations with polymorphic types/formulas in negative position do not have the *subformula property*, which holds in propositional systems (no polymorphism), and guarantees that decomposing judgments during proof search will only result in subgoals with simpler types/formulas in them. Here $T(A)$ may be more complex than $\forall\alpha.T(\alpha)$, and search termination is not guaranteed anymore.

To control the infinite space of possible instantiations $T(A)$, our approach is to study the structure of $\forall\alpha.T(\alpha)$ (by performing canonical proof search at this type) to suggest a (hopefully finite) set of instantiation candidates.

Let us go into the technical details with an example. If you consider the type $\forall\alpha.(A \rightarrow$

$\alpha) \to \alpha$, any canonical derivation of this type in a context Γ must be of the form:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A \qquad \dfrac{}{\Gamma, \alpha \vdash \alpha}}{\Gamma, A \to \alpha \vdash \alpha}}{\Gamma \vdash (A \to \alpha) \to \alpha} \qquad (\alpha \text{ fresh in } \Gamma, A)}{\Gamma \vdash \forall \alpha.(A \to \alpha) \to \alpha}$$

Going from the root of the judgment to the leaves, the two first reasoning steps (introducing $\forall\alpha.$ and decomposing the function $(A \to \alpha) \to \alpha)$ are imposed by the discipline of focusing, they are automatic steps that our simply-typed algorithm already implements. The third step, claiming that there is a unique/canonical way to build α by applying the function $A \to \alpha$, required human reasoning; in particular, in the left premise we have $\Gamma \vdash A$, instead of $\Gamma, A \to \alpha \vdash A$, and this simplification (strengthening) relies on meta-the meta-level reasoning: since α is a fresh type variable that cannot occur in A, it is not useful to build an argument of type A. We have to teach this more advanced reasoning to an automated search program.

This partial derivation shows that, without loss of generality, any proof of $\Gamma \vdash \forall\alpha.(A \to \alpha) \to \alpha$ contains a derivation of $\Gamma \vdash A$ hidden inside it. Our proposed approach is to consider that, dually, to deconstruct a $\forall\alpha.(A \to \alpha) \to \alpha$ is to deconstruct the proof of A inside it.

Instead of the general left-introduction rule for polymorphic types/formulas on the left, we would therefore propose, for *this specific type*, to use the specialized rule on the right:

$$\dfrac{\Gamma, \forall\alpha.T(\alpha), T(A) \vdash B}{\Gamma, \forall\alpha.T(\alpha) \vdash B} \qquad\qquad \dfrac{\Gamma, A \vdash B}{\Gamma, \forall\alpha.(A \to \alpha) \to \alpha \vdash B}$$

Notice in particular the assumption with the polymorphic type is gone: not only did we extract an A from the polymorphic type, but we decided to not consider it in the rest of the proof search anymore: an A is all that there is to this polymorphic formula.

Generalizing this idea, this suggests trying to define a judgment of the form $\Delta \Vdash T$ to express the idea that "from any proof of T we can exactly extract proofs of the formulas in Δ", and the following generic rule to eliminate left polymorphism, inspired by the "higher-order rule" of Zeilberger (Zeilberger, 2009):

$$\dfrac{\forall\Delta, \quad \Delta \Vdash \forall\alpha.T(\alpha) \implies \Gamma, \Delta \vdash B}{\Gamma, \forall\alpha.T(\alpha) \vdash B}$$

Studying partial canonical proofs for other concrete types shows that this approach could work very well for some polymorphic types of interest. In particular, for $P := \forall\alpha.(A \to B \to \alpha) \to \alpha$ and $S := \forall\alpha.(A \to \alpha) \to (B \to \alpha) \to \alpha$, which are well-known encodings using polymorphism of the *product* and *sum* of A and B, performing the same search process by hand suggests that $A, B \Vdash P$ and $A \Vdash S$ and $B \Vdash S$ hold, so that the polymorphism elimination rule behaves exactly like the usual rules for built-in product and sum connectives.

On the negative side, we already know of types where this process should not work, or we know that it would be difficult to make it work. This should not work for $(A \to 0) \to 0$: we can show that each canonical inhabitant at this type contains a proof term of type A, but we cannot (in an intuitionistic/pure setting) extract this proof term. This would be difficult for $\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$, where a partial proof search show the set of canonical proofs has a *recursive* structure (isomorphic, of course, to the set of natural numbers). This suggests that the generic rule we propose, for some polymorphic types, would have to eliminate polymorphism by producing recursive types $\mu\alpha.T(\alpha)$ instead; it may be possible to obtain interesting proof search result in this case, but this requires studying the situation of fixpoint / inductive types.

To summarize, our proposed approach is to deal with the left-introduction rule (or "usage rule") of *some* polymorphic types $\forall\alpha.T(\alpha)$ by computing a set of important/principal instances by searching the space of canonical inhabitants for $\forall\alpha.T(\alpha)$. Due to the barrier of undecidability, we know that this will not work for all polymorphic types; but our manual experiments suggest that it could work for interesting classes of polymorphic types, in particular for the impredicative encodings of data structures. We need to find how to formalize this idea as a fully generic/automatic approach, to understand the structure of the subsets of System F for which it works, and to experiment with various use-cases to understand how restrictive this subset is.

We discovered a connection with a different problem in the meta-theory of System F, namely the expressivity of atomic polymorphism (Fernando Ferreira, 2013). In particular, very recent work of Paolo Pistone (Pistone, 2018a,b) suggests an approach inspired from linear logic, and defines a fragment of System F called the "Yoneda fragment" (by a technically-rigorous analogy with the Yoneda Lemma of category theory). We believe that this Yoneda Fragment is part of the subset of System F that our approach can handle, but that it can be extended beyond it. Paolo Pistone is currently a post-doctoral student in Tübingen, and we would be interested in offering him the post-doctoral position funded by the REPRO project.

Deliverable WP1: a canonical representation of a well-defined fragment of System F, containing at least the impredicative translations of sum and product datatypes.

Risk management Meeting this objective fully is a high-risk, high-reward challenge. The case of simple types which we previously studied is much simpler than the richer type theories we propose to study, and yet they required new technical approaches and allowed, as unexpected side-results, to solve long-standing open problems. One could predict that working with System F, fixpoint types or dependent types will also require new technical ideas, and hope that they will also provide unexpected results – but there is high risk that such a satisfying solution would not be found during the project lifetime.

Our solution to mitigate risk is to consider fragments of the system (such as the Yoneda fragment we mentioned). We find it very important to be able to precisely describe subsets on which the partial solutions work reliably, and provide theoretical guarantees in this case. We are confident that interesting fragments can be proposed during the project lifetime; a part of our work will be to study practical use-cases to study their practical value and limitations.

Another approach would be to experiment with "heuristics" to instantiate polymorphic formulas, which could work well in practice for examples of interest. We will not follow this approach for two reasons. First, we have already experimented with simple/natural heuristics (typically, instantiating with the subformulas of the judgment) on top of the software prototype we built for the simply-typed case, and found the result to be very disappointing. Second, heuristics whose application domain are not well-understood limit the opportunities to build further research on the work.

2 New connections to logic

In principle, any new approach in proof theory could potentially be transposed to our work on canonical program representations; but they may be unsuitable if they do not satisfy computational completeness, a stronger form of completeness than what logicians typically study. We propose to study the new brand of work on "combinatorial proofs", establish a computational completeness result, and study its usefulness for our study of canonical forms or for practical implementations.

Typically, when logicians interested in canonicity propose a new form of proof structure $\Gamma \vdash_{\text{new}} A$ for an existing logic with a standard proof representation $\Gamma \vdash A$, they validate the new structure by establishing *provability completeness*: whenever $\Gamma \vdash A$ holds, then $\Gamma \vdash_{\text{new}} A$ also holds. The converse direction is also proven but is generally easier: a new "tighter" proof π can be read back as a traditional proof $\lfloor \pi \rfloor$, the difficulty is to take an arbitrary traditional proof and coerce it into the new, more structured presentation.)

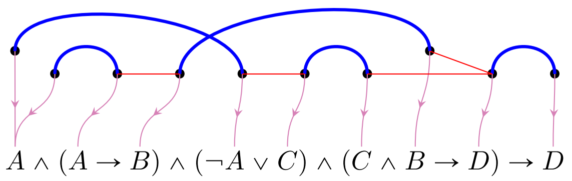
Through the Curry-Howard isomorphism, this is a result between two type systems: if there is a t (in the old type system) such as $\Gamma \vdash t : A$ holds, then there exists a u (in the new type system) such that $\Gamma \vdash_{\text{new}} u : A$ holds. Again, the converse direction typically holds: u can be read back as a term $\lfloor u \rfloor$ in the old type system.

Computational completeness is the following stronger statement: whenever $\Gamma \vdash t : A$ holds, then there exists a u such that $\Gamma \vdash_{\text{new}} u : A$ *and* the program equivalence $\Gamma \vdash t \simeq \lfloor u \rfloor : A$ holds. In other words, not only can we find a new term u from any old term t, but we can find a u with the same computational content as t. The new system ($\vdash_{\text{new}}$) proves the same formulas, but it also contains the same programs.

In our work on using *focusing* for canonical representations of programs, we had to establish computational completeness of focusing. Interestingly, the proof is the same as the usual proof of provability completeness, we just exposed that it establishes a more precise statement. On the other hand, we also tried to use *contraction-free logics* (Dyckhoff, 1992) for canonical program search, and those logics do not satisfy computational completeness – as a type system, they prevent calling most functions more than once.

In this objective, we propose to look at *combinatorial proofs* (Acclavio and Straßburger, 2018), a new brand of work in proof theory which summarizes proofs in classical and intuitionistic logic by graph representations inspired from the proof nets of linear logic, and enjoy interesting canonicity properties. Generally speaking, the idea of combinatorial proofs is to abstract away from particular proof rules and represent the flow of values in the proof by finite graphs satisfying certain combinatorial properties; different proofs can be represented by the same graph.

For illustration, an example of combinatorial proof is given below:


$A \wedge (A \to B) \wedge (\neg A \vee C) \wedge (C \wedge B \to D) \to D$

The first question is whether combinatorial proofs satisfy computational completeness: we expect this to be the case, but it has to be formalized. The second question is whether they can provide an improvement over focusing-based representations to provide canonical representations of programs; an already-canonical representation cannot be made "more canonical", but a new structure could give a simpler description of this representation, which could simplify and accelerate implementations. Finally, it is possible that the new perspective arising from using combinatorial proofs as program representations would result in new insights for logicians – typically, suggest refinements to the notions of proof equivalence they are using, for example in presence of units.

Deliverables

WP2.1 A computational completeness result for intuitionistic combinatorial proofs.

WP2.2 A canonical term enumeration procedure based on enumerating combinatorial proof graphs, evaluated against our existing propositional enumeration procedure.

WP2.3 A precise description of the combinatorial-proof equivalence induced by contextual equivalence through the Curry-Howard isomorphism.

Risks The main difficulty for this objective is that the PI is not currently familiar with combinatorial proofs – a recent subject. Fortunately, one of the main researchers on combinatorial proofs, Lutz Straßburger, is part of the same INRIA team and would provide guidance and help to study the connections with our program representations.

There is a low risk that combinatorial proof fail to satisfy computational completeness, and a medium risk that their use does not improve over existing focusing-based program representations – or that they present a small improvement that does not offset the cost of using a less-familiar representations when presenting our work to the programming-languages community.

3 Practical equivalence checking

A first suggestion for equivalence checking would be to provide a public implementation of an equivalence algorithm for the largest known-decidable fragments (simply-typed lambda-calculus with sums and the empty type). While each academic work in this direction has provided prototype implementations, they are not maintained anymore and are difficult to use for study and demonstration purpose. Providing a well-maintained demonstration implementation (in particular usable as a website) would ease the diffusion of these ideas in the programming language community. It is also an excellent onboarding task for a new student or intern.

Our flagship application for a practical equivalence checker is checking refactoring patches: validating that a restructuring code change does not modify the program's observable behavior, as the programmer intends it. Roughly, the idea is to type-check the versions of the program before and after the modification, abstract away the parts of the two programs that are identical, and run an equivalence algorithm on the parts that were modified.

We propose to apply this idea to the OCaml programming language. Doing this requires work along several different axes:

Engineering The first difficulty is to go from a textual diff/patch (as typically produced by version-management tools) to a pair of well-typed programs (the input to our equivalence algorithm); this requires understanding enough of the program and its build artifacts to know which software libraries are being used (to resolve references to third-party libraries) and generally how to build and type-check the textual representation of the program.

The Trustworthy Refactoring group at Kent works on a tool that has a different purpose (automatically applying specific program transformations), but requires this same technical infrastructure. We hope to reduce engineering efforts by collaborating on a common (free software) implementation.

Theory Functional programming languages contain many more program-construction features than those described in the formal programming models we have studied so far. It is possible to abstract away some of these features as blackboxes – two programs are only equivalent if they are used in the exact same place in the exact same way – but some refactoring transforms will require a finer-grained understanding of some features.

The two main points of friction we expect are the module system, which decides the shape of the naming environment used at any program point, and the interaction of higher-order functions and mutable state.

The module system is involved in many refactoring transforms that move the definition of a program object from one part of the program to another. While we do not foresee major blockers, OCaml has a non-trivial module system that can be difficult to support comprehensively.

The interactions of higher-order functions and mutable states are known to be extremely delicate, both for program reasoning and program equivalence checking. Our project collaborator Guilhem Jaber has strong expertise on the interaction of polymorphism, higher-order functions and mutable state, using theoretical tools from game semantics – a flexible theoretical approach to describe the behavior of program fragments by systematically describing their possible interactions with their surrounding environment.

Ergonomics When the answer is “yes, the change exactly preserves equivalence”, the user has exactly the information they desire. However, the other situations require careful thinking about user interface.

There are cases where two programs should be expected to be “almost equivalent”, up to some differences that the user could want to explicitly ignore. For example, a change to a library (rather than a complete program) could include making a new function visible to client code, or removing an existing function. Those are observable behavior changes, but the library author may ask to check that users not using these functions do not observe any behavior change; this relaxed version of equivalence requires a good interface to indicate this to the tool. Another example is a modification that reorder the invocation of two functions, which is incorrect in general if the functions may perform side-effects, but where users assumes that those functions are observationally pure; again, how to specify this this to the tool so that it can confirm that equivalence holds under the purity assumption?

Finally, in the cases where the code change did affect the program behavior in observable ways, what explanation should we give to the user? Theory suggests that dis-equivalence is justified by providing an example of external environment (around the program fragment) that will distinguish the two versions; but can this be explained to a novice user?

Our general approach to ergonomics issues is to use the tool ourselves to the largest possible extent, to get experience in various situations and use-cases.

Finally, we propose to also study more specialized equivalence-checking algorithms in the context of compiler validation. When modifying an existing compiler to add a new optimization or to change an existing compilation pass, it is easy to introduce regressions that may silently miscompile certain programs. Equivalence-guided compiler validation proceeds by defining an equivalence algorithm, and building a compiler that will try both the old and new compilation pass, checking both results for equivalence. This validating compiler can then be run on all publicly-available codebases in the language, and would be able to find miscompilation bugs without running the resulting binaries – greatly improving trust in the compiler modification. This approach has been used, for example, to validate a rewrite of the HHVM PHP compiler (Benton, 2018) at Facebook by checking equivalence of low-level bytecodes; we would like to apply it to higher-level intermediate languages as found in implementations of functional programming languages.

A concrete case study would be the pattern-matching compilation pass of the OCaml compiler. Compiling pattern-matching into decision trees is a delicate, performance-sensitive part of ML-family functional languages, containing subtle optimizations. The pattern-matching compilation in the OCaml pattern-matching compiler is an old codebase that is difficult to maintain and evolve, but it has not benefited from a cleanup or rewrite in the last two decades due to the fear of introducing miscompilation bugs.

Deliverables

WP3.1 A demonstration version of the equivalence algorithm for simple types using proof-theoretic techniques, available online.

WP3.2 A prototype of equivalence checker for OCaml refactoring changes, built in collaboration with Guilhem Jaber and the Trustworthy Refactoring group at Kent.

WP3.3 An equivalence checker for the output of the OCaml pattern-matching compiler, used to validate a re-engineering of the pattern-matching compiler.

Risk Equivalence checking for behavior-preserving changes is a low-risk objective in the sense that, if some refactoring changes prove too difficult to reason about, it is always possible to declare them out of scope of the – necessarily partial – tool, and cover simpler transforms that are already used in practice. In other words, scientific difficulties can degrade the performance of our proposed tools, but not their feasibility.

4 Saturation for synthesis

Our proof-theoretic approach to canonical program representations, based on focusing and our notion of *saturation*, reduces the search space of a program-enumeration procedure, as used in type-directed program synthesis tools. We propose to experiment with concrete implementations of saturation-based program synthesis, in the hope of obtaining performance gains compared to current approaches.

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: search space reduction could bring large performance improvements to existing program synthesis systems, there is a also medium risk that it would turn out to be unusable for synthesis.

One reason to be optimistic is that implementors of type-directed program synthesis tools (Osera and Zdancewic, 2015; Frankle, Osera, Walker, and Zdancewic, 2016; Polikarpova, Kuraj, and Solar-Lezama, 2016) have already moved in this direction by introducing *focusing* techniques to restrict the search space of their tool. In fact, they started by re-inventing the ideas of focusing, without knowledge of their proof-theory background, before they realized the connection between their search heuristics and this proof search technique.

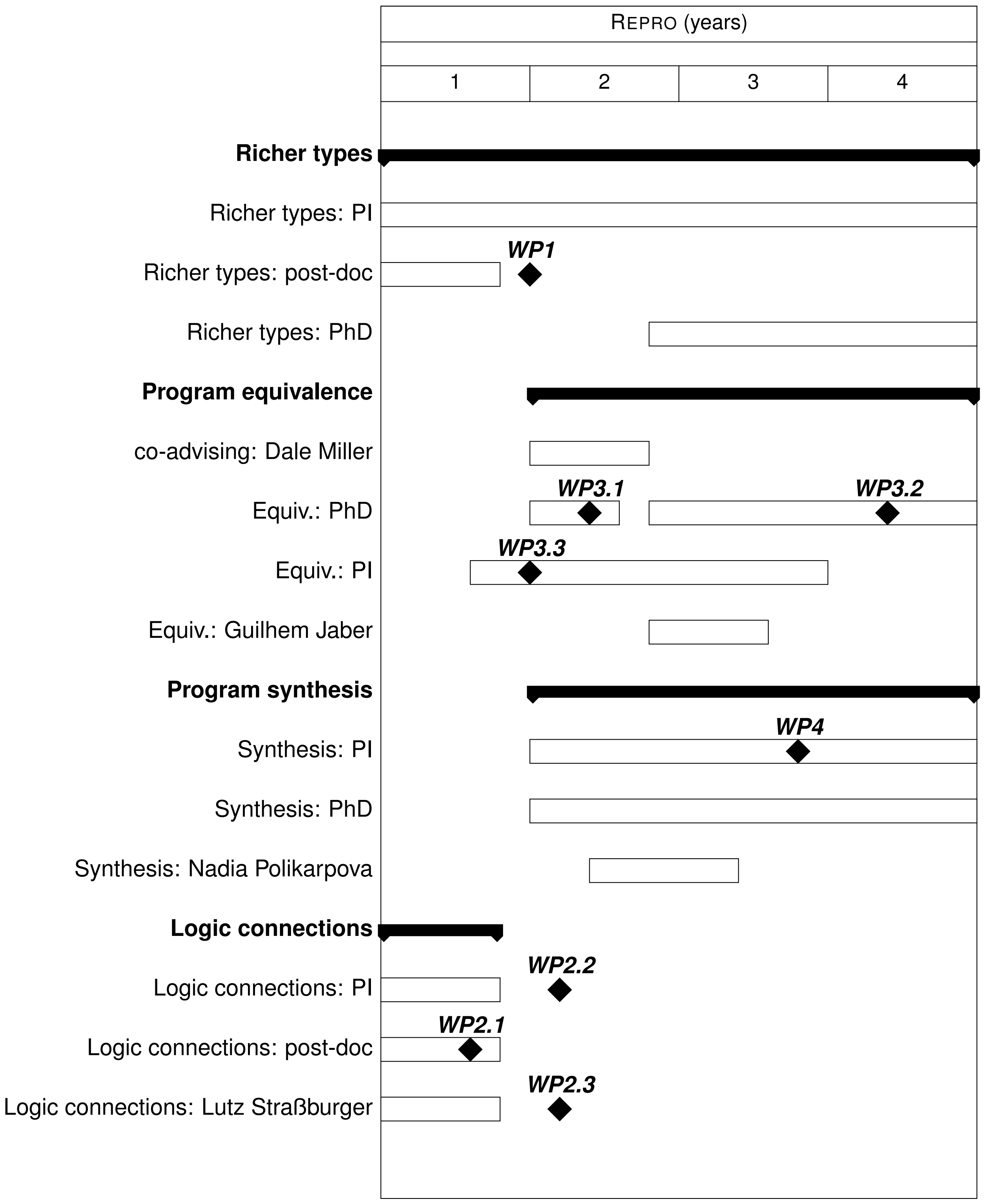
The search space reduction introduced by focusing in program-synthesis engines is dramatic: it does not only bring a speedup, but it turns entire subproblems for "infeasible" to "feasible". Even though saturation is more costly to implement, we hope that similar search-space reduction would offset the book-keeping costs associated with saturated forms.

This objective is naturally linked with our first objective of moving towards richer type theories: the richer theories we can handle, the richer theories can be accelerated for program synthesis. That said, richer type theories always contain the simply-typed lambda-calculus at their core, so even if we have not made progress on the first objective we can obtain synthesis speedup for realistic richly-typed programs.

Deliverable **WP4** a prototype for type-directed program synthesis using our canonical form structure to guide synthesis, evaluated against existing program-enumeration approaches.

Risk There is a medium risk that the cost of larger normal form offsets the search-space reduction brought by saturation. To fairly measure this, one first difficulty is to be able to implement a competitive program-synthesis engine; collaboration with Nadia Polikarpova, who is one of the leading expert on type-directed synthesis for richly-typed languages, will be a key enabler for this objective.

II Project organization and realization



Data plan, Open Access and Intellectual Property None of our objectives involve building research databases or handling personal data.

We plan to follow and encourage the best Open Access practices by publishing our results in Open Access venues that do not charge authors more than fair publishing costs ($\leq$ €200).

Any software produced during this project will be Free Software, using least restrictive licenses (MIT or BSD) for widest possible dissemination.

a Scientific coordinator and their consortium/team

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 lightly involved in the project. One of them (depending on the PhD student's main focus) would also co-supervise the PhD student[1]. Project funding would also help securing collaborations external to INRIA Saclay, with Guilhem Jaber (INRIA Rennes), the Trustworthy Refactoring group in Kent, and Nadia Polikarpova (University of California, San Diego).

1 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. In particular, he is one of the most active maintainers of the OCaml programming language implementation, which is widely used in programming-language and formal methods research communities (Coq, Why3, HOL-Light, CIL, F*, etc.), and also has a solid userbase in industry (Jane Street, Facebook, Docker, first implementations of Rust, WebAssembly, etc.). This makes OCaml a natural testbed for his programming language research, with potential for practical impact beyond research prototypes.

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 provide an opportunity to fund and supervise research on his own research goals. In particular, Gabriel Scherer has supervised several internships (bachelor and master interns), but needs external funding to supervise a PhD student. This is a necessary step in scientific growth, including to obtain his Habilitation Thesis, and to demonstrate personnel management skills in preparation of an ERC submission.

[1] An administrative requirement in France, as the PI has not yet passed its Habilitation Thesis.

Finally, this JCJC project would also provide appropriate funding and opportunities to develop a new expertise in type-directed program synthesis, an emerging topic within general program synthesis research that is, for now, mostly specific to the United States (Osera and Zdancewic, 2015; Frankle, Osera, Walker, and Zdancewic, 2016; Polikarpova, Kuraj, and Solar-Lezama, 2016).

2 Collaborators Our direct colleague Dale Miller is one of the leading experts on focusing in proof theory, and is a 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 is a natural collaborator for our second objective, "New connection to logics".

Our natural collaborators for "Practical equivalence checking" are 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 polymorphism, non-termination and general references. Guilhem Jaber would bring his expertise on game semantics in the project, and would be a natural host for a visit of a PhD student working on program equivalence.

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. There is no expert on type-directed program synthesis in France today, and it would be hard to fairly compare our work with existing state-of-the-art implementations without an effective transfer of knowledge on program-synthesis engines. The ANR funding could help invite Nadia Polikarpova in Paris (possibly with a student) for a few weeks, allowing for effective exchanges to bootstrap an effective collaboration. We are interested in other problem domains related to type-directed program synthesis, outside the scope of the present project, so we hope that such a collaboration would endure in the long term.

Involvement of the coordinator in other ongoing projects

The PI is not currently involved in any other national or international project.
They participated to NSF grant NSF CCF-1618732 until July 2017.

b Requested funding to reach the scientific objectives

We request €231,000 to fund one PhD scholarship (€110,000), a one-year postdoc (€62,000), a travel and equipment budget (€5,000 per person-year (8)), and budget to visit or host collaborators (€3,000 per project-year (4)). The direct total is €224,000, plus €17,000 (8%) in environmental costs.

Human resources In addition to 70% of the PI time, and about 10% of time of each collaborator, we propose to assign this project to one post-doc and one PhD student.

For the post-doc, we already have a candidate in mind: Paolo Pistone, who defended his PhD in 2015, and is an expert on the notion of program equivalence in presence of polymorphism, with cross-disciplinary ties in mathematical logic and philosophy.

We would like to hire Paolo Pistone as soon as the project start, and to quickly get working on **WP1** (canonical representations on a well-defined fragment of F). In parallel we could work on the **WP2.*** deliverables, in collaboration with Lutz Straßburger; Paolo Pistone also has expertise in linear logic proof nets, which would easily transfer to studying combinatorial proofs.

For the PhD student, we do not yet have a candidate in mind. Having a project over 4 years, with a PhD duration of only 3 years, gives us an extra year to select and recruit an excellent candidate. A PhD student also needs a gentler onboarding, starting with easy tasks that let them develop their own autonomy. This suggests starting with our deliverables **WP3.1** (a demonstration implementation of the existing saturation-based equivalence checker for sums), to learn about program equivalence checking and move towards other **WP3.*** deliverables, and **WP4** (a program synthesis prototype) at their own rhythm.

After some time learning the fundamentals of the topic, the PhD student could hopefully also contribute to **WP1**, extending our notions of canonical forms towards richer type theories; this will also depend on the personal inclination of the student, to move towards more theoretical or more applied topics; the PI and a post-doc should be enough to seriously attack the objectives 1 and 2, should the PhD student prefer to concentrate on 3 and 4, guided by the PI and supported by collaborations with Guilhem Jaber and Nadia Polikarpova's group.

Basic funding Our research domain is highly international, centered around top-profile conferences (rather than journals) such as POPL, ICFP and LICS that typically occur in Europe, America and Asia every three years. As a result, it is easy to commit to two international trips per year, eating most of the €5,000 allocated to travel and equipment. On the other hand, we do not require heavy equipment purchases; over four years, each project participant can expect to buy at most one work laptop, which easily fits in the remainder of the basic funding.

We requested extra funding (€3,000 per year) to fund visits to and from our collaborators. Guilhem Jaber from Rennes and the Trustworthy Refactoring group in Kent are easy to reach by train, and would not be high sources of expenses for visit either way. Most of the visit budget would be consumed by inviting Nadia Polikarpova, and possibly a student, to visit us in Paris for starting a (remote) collaboration – taking advantage of the attractiveness of Paris as a scientific and visit location.

Summary table of requested funding

	INRIA
Frais de personnel	€ 162,000
Coûts des instruments et du matériel (dont consommables scientifiques)	
Coûts des bâtiments et des terrains	
Prestation de service et droits de propriété intellectuelle	
Frais généraux additionels: Missions	€ 52,000
Frais généraux additionels: Frais d'environnement**	€ 17,000
Sous-total	€ 231,000
Aide demandée	€ 231,000

III Project impact

Our project being relatively theoretical in nature, one should not expect direct economic impact. On the other hand, we plan to build the principles for an emerging series of tools (programming languages, equivalence checkers, program synthesis) that may participate in changing the way people write computer programs, by lowering the cost of using richer type systems to write correct programs.

Publications We plan to disseminate our results through international publications in the best venues of our field, such as ICFP, POPL – which fortunately allow Diamond Open Access as publication model.

Teaching Our general approach of using proof-theory tools to attack programming-language problems remains fairly new, and is not taught in research-oriented master courses. Over the duration of this project, we hope to be able to give master-level lectures, in universities and during thematic summer schools, to better publicize the technique and form new generation of students to this cross-disciplinary approach.

Software Another source of impact is the publication of robust software prototypes. Most software produced through the project would remain at the stage of experimental prototype, but we will nonetheless ensure that good software engineering principles are followed to favor reuse by ourselves and others, and ensure reproducibility of our research results.

The refactoring-checker tool that we plan to build has potential for wider adoption than most software prototypes, as it would solve a real-world need for which no tool currently exists. Again, we hope that good software engineering principles could ensure a promising future for the codebase we built, living longer than the project itself; possibly as part of a collaboration with the Kent group, or as a collaboration with program tooling editors. We do not expect real end-user adoption during the duration of the project, given the amount of scientific and engineering challenges to solve before releasing usable versions, but this could be an option to encourage in future developments – for example as part of a follow-up ERC project.

Scientific mediation The PI has experience with research blogging during his Phd at INRIA Rocquencourt (where the PI started the Gallium blog) and his post-doc at Northeastern University (where the PI started the PRL blog). We would like to similarly start a collective blog for technical writing related to the project, hosting posts from the PI, students and collaborators.

We also have experience participating to mediation events (Fête de la Science), where our team should be present, and writing on technical websites for French technology enthusiasts – see [Zeste de Savoir] La recherche en langages de programmation au quotien – which could provide popularization outlets for this ANR project.

IV Bibliography

Matteo Acclavio and Lutz Straßburger. From Syntactic Proofs to Combinatorial Proofs. In *IJCAR*, 2018.

Arbob Ahmad, Daniel R. Licata, and Robert Harper. Deciding Coproduct Equality with Focusing. Unpublished, 2010.

Nick Benton. Semantic equivalence checking for HHVM bytecode. In *PPDP*, 2018.

Corrado Böhm. Alcune proprieta delle forme normali nel k-calcolo. *IAC Pubbl*, 696119, 1968.

Kaustuv Chaudhuri, Dale Miller, and Alexis Saurin. Canonical Sequent Proofs via Multi-Focusing. In *IFIP TCS*, 2008.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *PLDI*, 2016.

Gabriel Scherer. Deciding equivalence with sums and the empty type. In *POPL*, 2017a.

Gabriel Scherer. Search for program structure. In *SNAPL*, 2017b.

Gabriel Scherer and Didier Rémy. Which simple types have a unique inhabitant? In *ICFP*, 2015.

Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 1992.

Gilda Ferreira Fernando Ferreira. Atomic Polymorphism. *Journal of Symbolic Logic*, 2013.

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *POPL*, 2016.

Neil Ghani. Beta-Eta Equality for Coproducts. In *TLCA*, 1995.

Dominic J. D. Hughes. Unification nets: canonical proof net quantifiers. In *LICS*, 2018.

Danko Ilik. The Exp-Log Normal Form of Types. In *POPL*, 2017.

Peter-Michael Osera and Steve Zdancewic. Type-and-Example-Directed Program Synthesis. In *PLDI*, 2015.

Paolo Pistone. Proof nets and the instantiation overflow property. *CoRR (arXiv)*, 2018a.

Paolo Pistone. Proof nets, coends and the Yoneda isomorphism. *CoRR (arXiv)*, 2018b.

Max Schäfer, Torbjörn Ekman, and Oege de Moor. Challenge Proposal: Verification of Refactorings. In *PLPV*, 2008.

Noam Zeilberger. *The Logical Basis of Evaluation Order and Pattern-Matching*. PhD thesis, Carnegie Mellon University, 2009.