Research Programme

ALLAIS Guillaume

Practical Dependently Typed Programming

My long-term goal as a researcher is to design languages, tooling, and practices that unlock the ability for *more* users to write *safer* and *more concise* programs.

Program verification techniques allow users to root out entire classes of expensive or dangerous bugs in systems of critical importance. In recent years, such techniques have been used to implement a wide variety of systems e.g. an operating system kernel, an optimising compiler for a large subset of C, a reference implementation of the TLS networking standard, a file system, and a self-hosting dependently typed language.

Most of these projects used tools centering after-the-fact verification. They demonstrate that this approach is a viable, if costly, technique to develop critical systems. A highly trained expert programmer attempts to write a bug-free program, tries to demonstrate that it satisfies the specification, uncovers a bug while doing so, patches the bug, and iterates until the proof is accepted. This methodology is costly in development time and expert developer availability.

While after-the-fact verification is feasible for critical systems today, the advent of Artificial Intelligence-driven program generation would force expert users to wade through a large amount of invalid programs. My work develops a new approach that lets the system automatically discard as many faulty programs as possible before they are even presented to an expert human.

I implement and use expressive dependently typed systems that empower programmers to make their goals and assumptions explicit from the very beginning. The machine then assists the user in interactively writing a program that satisfies the goals and does not violate the assumptions. Because these goals and invariants are stated explicitly, the machine has access to a lot of information that can be leveraged to provide better feedback, suggestions, or automation during the interactive development process. So instead of making the programmer rewrite their code to eliminate bugs, the correct-by-construction methodology helps them produce code that does not have these bugs the first time around.

I follow a whole stack approach, and work on the language's theoretic foundations, its implementation, the interactive user experience, and the engineering patterns to take advantage of the unlocked expressivity. After years of contributing to the Agda compiler, I took a lead developer role in the Idris 2 project which is a dependently typed system written in itself. This cemented my expertise of the whole compiler stack. I want to tackle the core barriers to the wider adoption of correct-by-construction software: performance, automation, and user experience.

Performance

With dependently typed languages being used for real world applications, the performance of the compiler itself and the generated code becomes crucial. In the case of Idris 2, both goals are one and the same because it is its own implementation language.

A Concurrent Typechecker State of the art compilers for dependently typed languages such as Agda or Idris 2 are extremely stateful programs, despite being written in purely functional languages. In order to take advantage of increasingly massively multicore processors, we need to reign in this complexity. A clean slate design is necessary to identify what amount of state is needed and how it can be handled in a concurrent setting.

For the past year I have been collaborating on the development of a language of concurrent actors to write typecheckers and elaborators [3]. We already have a functioning prototype demonstrating the key design ideas.

This project's future impact is twofold. First as a domain specific language it is the ideal platform for the formal study of well behaved algorithmic type theories, their structure, and their models.

Second, the innovative design we are exploring can inform the restructuring of existing compilers' internals to effectively take advantage of multicore processors.

Control over Runtime Representations State of the art implementations of dependently typed languages include some optimisations to erase redundant data when compiling invariant-heavy datatypes and programs. These optimisations do not however change the overall structure of the runtime representation of the data. For instance a length-indexed list may be compiled to a plain list but we cannot expect much better than that.

Sometimes users can come up with a cunning low-level encoding of a datatype. The host language should let them seamlessly use the convenient high-level datatype-based presentation while having the guarantee it will get compiled to the efficient low-level runtime representation. I have demonstrated in a recent article [1] that this technique is effective but it currently requires boilerplate code that is tricky to write.

Delivering on this vision will require acting on the type theory, the language implementation, and designing programming patterns that users can reliably reach for in order to achieve maximal efficiency while keeping a high-level invariant-heavy interface.

Automation at Large

In dependently typed languages programmers start by stating their intent. The information thus available can be leveraged to automate away some of the repetitive work. The user can thus focus on the big ideas and let the machine figure out the mundane details.

These details may be filled fully automatically by proof search or interactively by composing powerful abstractions defined generically. The separation between proof search and generic programming is of degree rather than nature. In both cases, the key insight is in isolating a class of problems that can be dealt with once and for all.

Proof Search and Proof Simplification Whenever the formalisation of a problem involves reasoning modulo a theory, proofs can become a time sink for developers. Languages typically provide solvers to (semi-)decide goals that need to be discharged.

In the past, I have had the gruelling experience of formalising a monolithic decision procedure for Presburger arithmetic in Agda. In contrast, over recent years I have had the pleasure of collaborating on Frex [5, 2], a project that uses universal algebraic techniques to build a modular and extensible framework to define algebraic solvers. Much of the machinery can be reused from one solver to the next and generic tools can be defined.

During my work on proof extraction for Frex, it became manifest that some derivations could be drastically simplified. Preliminary results demonstrate that some simplification passes can be implemented generically. I hope to study a general approach to proof simplification taking into account parallelisable derivation steps. All the solvers we define in Frex will then directly benefit from the improvements.

Generic Programming Whenever a pattern arises repeatedly, it is an opportunity for us to abstract over it and provide a small library that captures it precisely. Dependently typed programming systems are particularly well suited to generic programming: thanks to the power of dependent types, it amounts to ordinary programming over a precisely defined universe of discourse.

I am particularly passionate about using advanced features of existing type systems to provide as seamless as possible libraries. See for instance my library for n-ary polymorphic functions which exploits Agda's unification machinery to minimise the required user annotations [7].

Numerous libraries force users to repeat the same information in multiple places. My work on a library for declarative hierarchical command line interfaces [8] demonstrates how we can use dependently typed languages to avoid this repetition. It shows that from a single description of the command line interface, one can produce documentation, support for auto-completion, an inductive representation of parse trees, and the corresponding parser. Using unparsing techniques, we can also take advantage of these first order representations of interfaces to provide a correct-by-construction toolbox for invoking external command-line tools.

The generic programming toolbox can be applied to any area where the end-user has to write a lot of boilerplate code.

Meta-programming Generic programming requires bridging the gap between a user-written type and the internal representation a library may be using.

The approach I embraced in the POPLMark Reloaded case study [6] was to work directly in the encoding and use pattern synonyms to recover a more standard interface.

A more popular alternative is to use meta-programming. The current state of the art in both Agda and Idris 2 involves writing weakly typed programs using a datatype representing one of the language's internal intermediate representations and 'unquoting' the result. This is the approach I adopted to implement the generic programs distributed as part of Idris 2's standard library.

I want to explore the design space to find a better solution that is both easy to use and provides stronger guarantees. For instance by indexing the encoding by its meaning in the host language and using a thin layer of meta-programming to bridge the gap by quoting a term to its representation.

User Experience

Because of the type-first approach to writing programs typical of dependently typed languages, program development is naturally type-driven. This approach turns writing a program into a dialogue with the machine.

Interactive Development During the development or refactoring of an existing codebase, programs are most of the time either unfinished or rejected by the compiler's static analysis. Language support for currently invalid programs has improved dramatically over the past decade. Originally languages such as Agda supported holes (i.e. parts of the program that are missing) but any invalid program would lead to an error message and the editor would lose all interactive features. As part of the Agda development team, I performed a sizeable refactoring of the Agda compiler that turned many errors into mere warnings, thus preserving interactivity.

The current state of the art is however still lacking: parsing or scope errors for instance still lead to hard failures and a complete loss of interactivity. I want compilers to automatically recover even in some of those situations and even sometimes provide users with plausible fixes that they can decide to apply or reject. I have been exploring such opportunities in the Idris 2 compiler.

Content Discoverability As systems and libraries grow, it becomes harder for newcomers to find what they are looking for. I have experienced this first hand as a co-maintainer of the Agda standard library [4], and of Idris 2. Current tools are extremely lacking so much so that even advanced users looking for a definition are reduced to running textual searches on the libraries' source code, or asking maintainers for help.

Mid-to-long term I am planing to build an interdisciplinary research project in collaboration with experts in library studies and human-computer interactions on bringing information retrieval techniques to compilers to help newcomers interactively explore a codebase and find relevant information.

References

[1] Guillaume Allais. “Builtin Types viewed as Inductive Families”. In: *CoRR* abs/2301.02194 (2023). DOI: 10.48550/arXiv.2301.02194. arXiv: 2301.02194. URL: https://doi.org/10.48550/arXiv.2301.02194.

[2] Guillaume Allais, Edwin Brady, Nathan Corbyn, Ohad Kammar, and Jeremy Yallop. “Frex: dependently-typed algebraic simplification”. Conditionally accepted for publication at POPL’23. 2023.

[3] Guillaume Allais, Malin Altenmüller, Conor McBride, Georgi Nakov, Fredrik Nordvall Forsberg, and Craig Roy. “TypOS: An Operating System for Typechecking Actors”. In: *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, Nantes, France.* 2022.

[4] Guillaume Allais Matthew L. Daggitt. “Using Dependent Types at Scale: Maintaining the Agda Standard Library”. In: *Workshop on the Implementation of Type Systems, WITS@POPL 2022, Philadelphia, Pennsylvania, USA, January 22, 2022.* 2022.

[5] Guillaume Allais, Edwin Brady, Ohad Kammar, and Jeremy Yallop. “Frex: indexing modulo equations with free extensions”. In: *The workshop on Type-Driven Development, TyDe 2020, August 23, 2020.* 2020.

[6] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. “POPLMark reloaded: Mechanizing proofs by logical relations”. In: *J. Funct. Program.* 29 (2019), e19. DOI: 10.1017/S0956796819000170. URL: https://doi.org/10.1017/S0956796819000170.

[7] Guillaume Allais. “Generic level polymorphic n-ary functions”. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2019, Berlin, Germany, August 18, 2019.* Ed. by David Darais and Jeremy Gibbons. ACM, 2019, pp. 14–26. ISBN: 978-1-4503-6815-5. DOI: 10.1145/3331554.3342604. URL: https://doi.org/10.1145/3331554.3342604.

[8] Guillaume Allais. “agdARGS – Declarative Hierarchical Command Line Interfaces”. In: *TTT : Type Theory Based Tools, TTT@POPL 2017, Paris, France, January 15, 2017.* 2017.