Research Statement

Gabriel Radanne

One of the central aspects of computer science is the act of programming: algorithms and languages are ultimately designed to be implemented into a program which will be analyzed, optimized, inspected and eventually run on a given hardware. All this task assume that someone, at some point, is going to write a program.

Unfortunately, this act of implementing correct programs is quite a painful part of the process. There are many difficulties that the programmer must overcome for his implementation. First, they must define what should be written and what are the criteria for their program to be qualified "correct". Determining the specification is, by itself, a difficult process. However, assuming this first hurdle is crossed, the programmer will now need to create an implementation using a given programming language and ensure its correctness.

Ensuring the correctness of already written software has spawned numerous fields in computer science by exploiting mostly two large families of techniques: testing and verification. These two techniques are, of course, complementary and have been employed with great success in numerous cases. These tasks, however, are only "a posteriori": they only try to verify properties of given programs. While they can be used to inspire and inform the writing of such programs, they do not fundamentally improve the act of programming.

Indeed, the main tool of programming is the programming language itself. Unfortunately, most programming languages are completely inappropriate for the many kinds of programs written in them: from concurrent programs written in C (whose memory model, while allowing efficient code, is very difficult to master) from large web applications written in Javascript (who provides neither encapsulation nor abstraction facilities, thus hindering many modularity aspects), programming is a constant struggle to torture the code the programmer wants to write into the code allowed by the programming language. Furthermore, most programming languages do not help in any way in the previous task of ensuring the correctness of programs.

One lead that proved very fruitful in the project of easing this proof of correctness is types. Types allow to verify numerous properties directly while programming. Since their introduction, types have allowed to check properties of increasing complexity and diversity (which probably reached its peak with the Coq programming language). Types, however, are not only useful for correctness: they provide statically checked documentation and make programming languages more ergonomic. For example, types are used

to provide accurate tab-completion for methods in modern editors or to allow "fearless refactoring": the ability to do complex refactoring simply by following the type errors.

My first interest is to develop a language approach to problem solving: by developing Domain-Specific Languages, it is possible to provide easy-to-use, convenient programming models for many specific and complex tasks. I believe this approach, which has been used in many contexts with great success, can be further improved by leveraging special purposed type systems to improve both the correctness and the usability of the DSL. In particular, I'm interested in exploring the association of regular, general-purpose, statically typed languages with foreign elements. This drove me to contribute to Eliom, an extension of the OCaml programming language for Web programming. OCaml is a general-purpose, industrial-strength, statically typed language. Eliom provides new powerful programming constructs on top of the OCaml language which allow to mix client and server code. In the context of my PHD thesis, I developed a novel type system [6] which statically ensures correctness of communications between server and client, but also allows to easily write complex communication patterns that are common in Web programming. I also formalized its semantics and compilation scheme and developed a module system [4] that allows to leverage the powerful abstraction and modularity features provided by ML module systems in the context of web programming.

Domain-specific languages can also be used from a tooling perspective: by providing declarative libraries as part of a generic programming language, we can let programmers customize tools to improve programming. For example, I improved the Mirage tooling by developing Functoria. Mirage is a library operating-system developed in OCaml which allows to build unikernels (applications that bundles the pieces of operating system they need). Functoria [2, 3] is a library allowing programmers to describe the architecture of their unikernels and assemble them according to a number of parameters (such as configure-time instrumentation and files) and targets (Mirage unikernels can run on a large variety of contexts). Functoria can also be used to statically document and describe unikernels in a way that is easy to access for other programmers.

My other interest is to improve the convenience of generic programming languages through the use of types. For example, I am developing a library for research by type signatures [1]. By leveraging the descriptive nature of types, it is possible to use type patterns, ie. types with holes, to search for function that matches this type inside libraries. The challenge here is to find results that are both sound (they do match the type) and useful (we do return results). Of course, this task grows more delicate as the type system is more expressive.

As I am interested in improving the tools for programming, I am also interested in using good tools to create software. This leads me to develop working prototypes and usable libraries in the context of my research. In particular, I implemented my work on Eliom as an extension of the OCaml typechecker and Functoria is now used in production by Mirage users.

This language-based approach can also be used to improve the other aspects of programming. Quickcheck, a library for property testing in Haskell, showed that very powerful testing techniques can be developed by leveraging an expressive language. Similarly, numerous programming languages have been developed to make static verification more

approachable, such as Why3 and Beluga. I believe there are many more domains where this approach is worth exploring in a formal manner.

References

[1] Dowsing. *Dowsing sources*. `http://github.com/Drup/dowsing`, 2017.

[2] Functoria. *Functoria sources*. `https://github.com/mirage/functoria/`, 2017.

[3] IntroFunctoria. *Introducing Functoria*. `https://mirage.io/blog/introducing-functoria`, 2017.

[4] Gabriel Radanne and Jérôme Vouillon. Tierless Modules. Draft, March 2017. URL `https://hal.archives-ouvertes.fr/hal-01485362`.

[5] Gabriel Radanne, Vasilis Papavasileiou, Jérôme Vouillon, and Vincent Balat. Eliom: tierless web programming from the ground up. In Tom Schrijvers, editor, *IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*, pages 8:1–8:12. ACM, 2016. ISBN 978-1-4503-4767-9. doi: 10.1145/3064899.3064901. URL `http://doi.acm.org/10.1145/3064899.3064901`.

[6] Gabriel Radanne, Jérôme Vouillon, and Vincent Balat. Eliom: A core ML language for tierless web programming. In Atsushi Igarashi, editor, *APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 377–397, 2016. ISBN 978-3-319-47957-6. doi: 10.1007/978-3-319-47958-3_20. URL `https://doi.org/10.1007/978-3-319-47958-3_20`.