Journées de la recherche de l'Université Saint-Joseph
Vice-rectorat à la recherche
9-10 Mai 2019
Résumé de l'intervention

Nom de l'auteur	**Youssef El Bakouny**
Titre	**Doctorant**
faculté	**Faculté d'Ingénierie**
Téléphone	**03 853 877**
Email	**Youssef.Bakouny@usj.edu.lb**
Titre de l'intervention	**Scallina : on the Intersection of Gallina and Scala**
Liste des auteurs	**Youssef El Bakouny and Dani Mezher**
Nom du présentateur	**Youssef El Bakouny**

Abstract

The Coq proof assistant allows the implementation of functional programs in the Gallina specification language while providing an environment for the semi-interactive development of machine-checked proofs. Once a Gallina program has been developed, Coq can extract from it an implementation in a Hindley-Milner based functional programming language such as OCaml or Haskell.

The Scala programming language, interoperable with Java, fuses the functional and object-oriented paradigms; providing a type system, based on the calculus of Dependent Object Types (DOT), which significantly differs from that of Hindley-Milner based languages such as OCaml and Haskell. On the one hand, Scala's type system requires the generation of significantly more type information but, on the other hand, can type-check some constructs that are not typable in OCaml and Haskell. Moreover, Scala diverges significantly from Gallina; a functional specification language based on the Calculus of Inductive Constructions (Cic). Therefore, an adequate extraction of Gallina programs to readable Scala code challenges us to reconcile the differences between these two languages.

In response to these challenges, the Scallina project defines a grammar [1] delimiting a common subset of Gallina and Scala along with a translation strategy for programs conforming to the aforementioned grammar. The Scallina translator [2] shows how these contributions can be transferred into a working prototype; enabling the practical Coq-based synthesis of Scala components. A typical application features a user implementing a functional program in Gallina, proving this program's correctness with regards to its specification and making use of Scallina to synthesize readable, traceable and idiomatic Scala code similar to what a programmer would usually write.

This synthesis of readable, traceable and idiomatic Scala components eases their integration into larger Scala and Java applications; opening the door for a wider community of programmers to benefit from the Coq proof assistant. In this context, the formalization of the Scallina grammar facilitates the reasoning about the fragment of Gallina that is translatable to Scala and the corresponding translation strategy paves the way for an adequate support of the Scala programming language in Coq's native extraction mechanism. A notable contribution of our strategy regards its mapping of Gallina records to Scala; leveraging the path-dependent types of this new target output language.

Introduction:

In recent years, software bugs have been causing significant harm to the engineering industry. Since software components are interconnected, a bug in one component is likely to affect others, causing a system-wide failure. Additionally, software bugs have often created vulnerabilities that are exploitable by malicious users. A notable example of significant losses caused by such bugs is OpenSSL's Heartbleed. To prevent further losses, we have recently seen a rise of interesting initiatives that use formal methods, potentially as a complement to software

testing, with the goal of proving a program's correctness with regards to its specification. A remarkable example of such an initiative is a U.S. National Science Foundation (NSF) expedition in computing project called “the Science of Deep Specification (DeepSpec)” [3].

Functional programming languages, thanks to their use of referential transparency, have always been amenable for the conduction of proofs. However, the manual checking of these proofs is impractical or, to say the least, time-consuming. As a result, several proof assistants have been developed to provide machine-checked proofs. They usually enable users to implement a functional program, prove its correctness with regards to its specification and extract a verified implementation expressed in a given functional programming language. The output program can be considered correct as long as the user trusts the implementations of the proof assistant and its corresponding program extraction mechanism. [4]

Coq [5] and Isabelle/HOL [6] are currently two of the leading proof assistants. Coq has been successfully used to implement CompCert, the first formally verified C compiler [7]; whereas Isabelle/HOL and Coq were respectively used to implement the seL4 [8] and the DeepSpec CertiKOS [9] formally verified general-purpose operating system kernels. The languages currently supported by Coq's extraction mechanism are OCaml, Haskell and Scheme [10], while the ones that are currently supported by Isabelle/HOL's extraction mechanism are OCaml, Haskell, SML and Scala [11].

The Scala programming language [12] is considerably adopted in the industry. It is the implementation language of many important frameworks, including Apache Spark, Kafka, and Akka. It also provides the core infrastructure for sites such as Twitter, Coursera and the Guardian. A distinguishing feature of this language is its practical fusion of the functional and object-oriented programming paradigms. Its type system is, in fact, formalized by the calculus of Dependent Object Types (DOT) which is largely based on path-dependent types [13]; a limited form of dependent types where types can depend on variables, but not on general terms.

The Coq proof assistant, on the other hand, is based on the calculus of inductive constructions; a Pure Type System (PTS) which provides fully dependent types, i.e. types depending on general terms [14]. This means that Gallina, the core language of Coq, allows the implementation of programs that are not typable in conventional programming languages. A notable difference with these languages is that Gallina does not exhibit any syntactic distinction between terms and types [5].

To cope with the challenge of extracting programs written in Gallina to languages based on the Hindley-Milner [15] type system such as OCaml and Haskell, Coq's native extraction mechanism implements a theoretical function that identifies and collapses Gallina's logical parts and types; producing untyped λ-terms with inductive constructions that are then translated to the designated target ML-like language, i.e. OCaml or Haskell. During this process, unsafe type casts are inserted where ML type errors are identified [16]. For example, these unsafe type casts are currently inserted when extracting Gallina records with path-dependent types. However, as mentioned in section 3.2 of [10], this specific case can be improved by exploring advanced typing aspects of the target languages. Indeed, if Scala were a target language for Coq's extraction mechanism, a type-safe extraction of such examples could be done by an appropriate use of Scala's path-dependent types.

It is precisely this Scala code extraction feature for Coq that constitutes the primary aim of the Scallina project. Given the advances in both the Scala programming language and the Coq proof assistant, such a feature would prove both interesting and beneficial. A typical application features a user implementing a functional program in Coq, proving this program's correctness with regards to its specification and making use of Scallina to synthesize Scala components which can then be integrated into larger Scala or Java applications. In fact, since Scala is also interoperable with Java, such a feature would open the door for a significantly larger community of programmers to benefit from the Coq proof assistant. Therefore, in accordance with Coq's native extraction mechanism, Scallina also aims to produce readable code; further encouraging its adoption in industrial and Open Source projects.

However, Scala's type system, which is based on DOT, significantly differs from that of OCaml and Haskell. For instance, Scala sacrifices Hindley-Milner type inference for a richer type system with remarkable support for subtyping and path-dependent types [13]. So, on the one hand, Scala's type system requires the generation of significantly more type information but, on the other hand, can type-check some constructs that are not typable in OCaml and Haskell.

Furthermore, as previously mentioned, both Coq's native extraction mechanism and Scallina aim to produce readable code; keeping in mind that confidence in programs also comes via the readability of their sources, as demonstrated by the Open Source community. For this purpose, Coq's extraction sticks, as much as possible, to a straightforward translation and emphasizes the production of readable interfaces with the goal of facilitating the integration of the extracted code into larger developments [17]. Scallina also opts for a straightforward translation strategy; favoring the synthesis of Scala code that is traceable back to the source Gallina code representing its formal specification. This enables potential adaptations of this specification to the needs of the larger application.

Results:
Therefore, in the context of the Scallina project, we define a grammar [1] that delimits a common subset of Gallina and Scala; facilitating the reasoning about the fragment of Gallina that is translatable to Scala using a relatively straightforward translation strategy. This subset is based on an ML-like fragment that includes both inductive types and a polymorphism similar to the one found in Hindley-Milner type systems. This fragment was then augmented by introducing the support of Gallina records, which correspond to first-class modules. In this extended fragment, Gallina dependent types are partially supported through Scala's path-dependent types; enabling types to depend on variables, but not on general terms [13].

Since the objective of the Scallina project is not to reinvent the Coq extraction process but to extend it with readable Scala code generation, it assumes that a prior removal of logical parts and fully dependent types was already performed by Coq's theoretical extraction function and subsequent type-checking phase; catering for a future integration of the Scallina translation strategy into Coq's native extraction mechanism. In this context, Scallina proposes some modification to the latter with regards to the typing of records with path-dependent types. These modifications, explicitly formulated as possible future works in section 3.2 of [10], constitute the main contribution of Scallina.

Furthermore, Scallina's optimized translation strategy, proposed for programs conforming to the aforementioned grammar, aims to generate idiomatic Scala code from equivalent Gallina constructs. For example, it leverages Scala's variance annotations and Nothing bottom type to propose a translation of Algebraic Data Types (ADT) that conforms with best practices implemented by Scala standard library data structures such as List[+A] and Option[+A]. The Scallina prototype [2] shows how our contributions, embodied by this translation strategy, can be successfully transferred into a working software.

Conclusion and discussion:
In conclusion, the Scallina project enables the translation of a significant subset of Gallina to readable, debuggable and traceable Scala code. The Scallina grammar, along with its proposed translation strategy, facilitates the reasoning about the fragment of Gallina that is translatable to functional object-oriented programming languages such as Scala and Swift. This translation strategy embodies several optimizations such as shallow embedding, implicit Nat conversions and the generation of curried Scala code while leveraging the language's variance annotations and Nothing bottom type during the translation of ADTs. Our strategy's most significant contribution resides in its mapping of Gallina records to Scala; leveraging the path-dependent types of this new target output language. The Scallina prototype shows how our contributions can be successfully transferred into a working tool. It also allows the practical Coq-based synthesis of Scala components that can be integrated into larger applications; opening the door for Scala and Java programmers to benefit from the Coq proof assistant.

Future versions of Scallina are expected to be integrated into Coq's native extraction mechanism by re-using the

expertise acquired through the development of the current Scallina prototype. In this context, an experimental patch for the Coq extraction mechanism was implemented in 2012 but has since become incompatible with the latest version of Coq's source code. The implementation of Scallina's translation strategy into Coq's extraction mechanism could potentially benefit from this existing patch; updating it with regards to the current state of the source code. During this process, the external implementation of the Scallina prototype, which relies on Gallina's stable syntax independently from Coq's source code, could be used to guide the aforementioned integration; providing samples of generated Scala code as needed.

References:

[1] Y. El Bakouny and D. Mezher, "The Scallina grammar: Towards a scala extraction for Coq," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018, vol. 11254 LNCS, pp. 90–108.

[2] Y. El Bakouny and D. Mezher, "Scallina: Translating Verified Programs from Coq to Scala," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018, vol. 11275 LNCS, pp. 131–145.

[3] B. C. Pierce, "The science of deep specification (keynote)," in *Companion Proceedings of the 2016 {ACM} {SIGPLAN} International Conference on Systems, Programming, Languages and Applications: Software for Humanity, {SPLASH} 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, 2016, p. 1.

[4] M. O. MYREEN and S. OWENS, "Proof-producing translation of higher-order logic into pure and stateful ML," *J. Funct. Program.*, vol. 24, no. 2–3, pp. 284–315, May 2014.

[5] The Coq development team, "The Coq proof assistant reference manual." 2004.

[6] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - {A} Proof Assistant for Higher-Order Logic*, vol. 2283. Springer, 2015.

[7] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *Proceedings of the 33rd {ACM} {SIGPLAN-SIGACT} Symposium on Principles of Programming Languages, {POPL} 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 2006, pp. 42–54.

[8] G. Klein *et al.*, "seL4: formal verification of an {OS} kernel," in *Proceedings of the 22nd {ACM} Symposium on Operating Systems Principles 2009, {SOSP} 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009, pp. 207–220.

[9] H. Chen, X. Wu, Z. Shao, J. Lockerman, and R. Gu, "Toward Compositional Verification of Interruptible OS Kernels and Device Drivers," *J. Autom. Reason.*, vol. 61, no. 1–4, pp. 141–189, Jun. 2018.

[10] P. Letouzey, "Extraction in Coq: An Overview," in *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, 2008, vol. 5028, pp. 359–369.

[11] F. Haftmann and T. Nipkow, "Code Generation via Higher-Order Rewrite Systems," in *Functional and Logic Programming: 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, 2010, pp. 103–117.

[12] M. Odersky and T. Rompf, "Unifying functional and object-oriented programming with Scala," *Commun. {ACM}*, vol. 57, no. 4, pp. 76–86, Apr. 2014.

[13] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki, "The Essence of Dependent Object Types," in *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, 2016, vol. 9600, pp. 249–272.

[14] N. Guallart, "An Overview of Type Theories," *Axiomathes*, vol. 25, no. 1, pp. 61–77, 2015.

[15] R. Milner, "A theory of type polymorphism in programming," *J. Comput. Syst. Sci.*, vol. 17, no. 3, pp. 348–375, Dec. 1978.

[16] P. Letouzey, *Programmation fonctionnelle certifiée : L'extraction de programmes dans l'assistant Coq*. 2004.

[17] P. Letouzey, "A New Extraction for Coq," in *Types for Proofs and Programs, Second International Workshop, {TYPES} 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, 2002, vol. 2646, pp. 200–219.

Recent Publications:

- Youssef El Bakouny and Dani Mezher. 2018. Scallina: Translating Verified Programs from Coq to Scala. In: *Ryu S. (eds) Programming Languages and Systems. APLAS 2018. Lecture Notes in Computer Science, vol 11275, pp. 131–145. Springer, Cham.*

- Youssef El Bakouny and Dani Mezher. 2018. The Scallina Grammar. In: *Massoni T., Mousavi M. (eds) Formal Methods: Foundations and Applications. SBMF 2018. Lecture Notes in Computer Science, vol 11254, pp. 90–108. Springer, Cham.*

Recent Prizes:

- December 2018: *Third prize at the Student Research Competition (SRC) of APLAS 2018 at Wellington.*

- June 2018: *Best engineering poster award at "9e journées de la recherche à l'Université Saint-Joseph".*

- April 2018: *Selected among the best SBMF 2018 papers for an extended version in Elsevier's Science of Computer Programming (SCP).*

Recent Presentations:

- November 29, 2018: SBMF, Salvador, Brazil.
- December 3, 2018: APLAS, Wellington, New Zealand.
- May 10, 2019: Journées de la Recherche, USJ, Beirut, Lebanon.
- June 7, 2019: Laboratoire de Recherche en Informatique (LRI), Paris, France.
- June 13, 2019: Ecole Normale Supérieure (ENS) Paris-Saclay, Paris, France.