Noname manuscript No.
(will be inserted by the editor)

The Software Language Extension Problem

Manuel Leduc · Thomas Degueule · Eric Van Wyk · Benoit Combemale

Received: date / Accepted: date

Abstract The problem of software language extension and composition drives much of the research in *Software Language Engineering* (SLE). Although various solutions have already been proposed, there is still little understanding of the specific ins and outs of this problem, which hinders the comparison and evaluation of existing solutions. In this SoSyM Expert Voice, we introduce the *Language Extension Problem* as a way to better qualify the scope of the challenges related to language extension and composition. The formulation of the problem is similar to the seminal *Expression Problem* introduced by Wadler in the late nineties, and lift it from the extensibility of single constructs to the extensibility of groups of constructs, i.e., software languages. We provide a comprehensive definition of the actual constraints when considering language extension, and believe the *Language Extension Problem* will drive future research in SLE, the same way the original *Expression Problem* helped to understand the strengths and weaknesses of programming languages and drove much research in programming languages.

Keywords domain-specific language · extension · composition · expression problem

Manuel Leduc
E-mail: manuel.leduc@irisa.fr
Univ Rennes, Inria, CNRS, IRISA, France

Thomas Degueule
E-mail: degueule@cwi.nl
CWI, The Netherlands

Eric Van Wyk
E-mail: evw@umn.edu
University of Minnesota, USA

Benoît Combemale
E-mail: benoit.combemale@irit.fr
University of Toulouse & Inria, France

Introduction

With the advent of language workbenches, the problem of modular language extension has garnered considerable interest from the research community in the past decade. This problem informally refers to the capability of extending the syntax and semantics of an existing language while reusing its specification (e.g., grammars, semantic inference rules) and implementation (e.g., parsers, interpreters). Various authors have attempted to formalize this problem (e.g., [5]) but the lack of a clear definition makes it hard to evaluate and compare the strengths and weaknesses of existing solutions w.r.t. a common, well-defined framework. This paper is an attempt to define language extensibility in the form of a well-defined problem.

From the *Expression Problem* to the *Language Extension Problem*

Philip Wadler coined the term "*Expression Problem*" to name a well-known problem in the programming languages community and this name has been in common use for more than two decades [21]. As Oliveira and Cook put it [15]:

> The "expression problem" (EP) is now a classical problem in programming languages. It refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants.

Over time, the EP has made it possible to structure the discussions around the capabilities of different programming paradigms and languages regarding data types extensibility using a common set of constraints

that candidate solutions should address. There are different variations of the EP, but its canonical definition includes the following constraints [22]:

Extensibility in both dimensions: It should be possible to add new data variants and adapt existing operations accordingly. Furthermore, it should be possible to introduce new operations.

Strong static type safety: It should be impossible to apply an operation to a data variant that the operation cannot handle.

No modification or duplication: Existing code should neither be modified nor duplicated.

Separate compilation: Compiling datatype extensions or adding new operations should not encompass re-type-checking the original datatype or existing operations.

Independent extensibility: It should be possible to combine independently developed extensions so that they can be used jointly.

There is a striking parallel between the problem of modular language extension and the *Expression Problem*. As a matter of fact, most approaches to modular language extension end up discussing and addressing the EP in some way [8,12]. However, in the context of Software Language Engineering (SLE), "data variants" are groups of syntactic categories and their constructors and "operations" over these data variants define their semantics. Due to the ambiguity in the name *Expression Problem*, in which "expression" may refer to a *language* of expressions, one might naively think that the EP and the problem of modular language extension are equivalent.

In this paper, we demonstrate that instantiating the EP in the context of SLE requires to reformulate and refine the existing constraints of the EP and to introduce new ones, leading to a new problem: the *Language Extension Problem* (LEP). **While the EP is merely a programming problem concerning programmers and focusing on the extensibility of a single datatype, the LEP is a Software Language Engineering (SLE) problem concerning language engineers and focusing on the extensibility of languages (i.e., group of datatypes representing the language constructs). The LEP must also account for engineering practices that are specific to software languages such as the use of language workbenches, the duality of language specifications and implementations, and the specificities of syntax definition.** As extending a group of datatypes entails extending the datatypes it is composed of, in many cases solving the LEP (in the large) entails solving the EP (in the small). We identify what is the meaning of the two extension axes in this context and what is the set of constraints that must be used to assess the success of a given solution. Naturally, many partial solutions to the LEP already exist in the literature, scattered from programming language theory (e.g., modular visitor components [14], Revisitors [12], Recaf [1]), to language workbenches (e.g., Rascal [10], Melange [3], Silver [18,9]). We purposely limit ourselves to the definition of the LEP, and leave to future work the positioning of existing solutions w.r.t. the constraints we list.

The Language Expression Problem

The *Expression Problem* has been initially introduced in the context of datatype extension and composition, hence presented in terms of *datatypes* and *functions* over the datatypes. Conversely, the *Language Extension Problem* focuses on language extension and composition, presented in terms of, respectively, *syntax* in the form of multiple syntactic categories and constructors for each category, and *semantics* over that syntax. In the following, we introduce the *Language Extension Problem* by paraphrasing the original definition of the *Expression Problem* by Wadler [21], but lifting the vocabulary from datatypes to languages.

> The *Language Extension Problem* (LEP) is a new name for an old problem. The goal is to define a family of languages, where one can add a new language to the family by adding new syntax (i.e., new constructors for existing syntactic categories as well as new categories) and also new semantics over existing and new syntax, while conforming to constraints similar to those in the *Expression Problem* but specialized to language extension.

As an example, consider a language family regarding state machines which starts from a core language over simple finite state machines with a simple pretty-printing semantics and constructs a new language by adding syntax to specify hierarchical state machines and a new semantics to evaluate state machines given an input sequence.

According to this characterization of the LEP, we now review the constraints initially identified in the EP, and express them in the context of SLE for the LEP:

Extensibility in both dimensions: It should be possible to extend the syntax and adapt existing semantics accordingly. Furthermore, it should be possible to introduce new semantics on top of existing syntax.

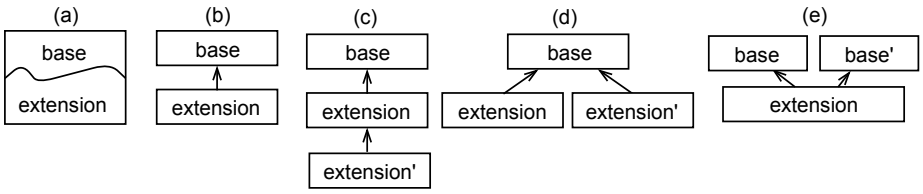

Fig. 1 Approaches for language extension, applicable at the specification and implementation levels. (a) mixes up the extension into the base language, while (b)-(e) keep them separated and use explicit operators (e.g., references, static/dynamic introduction) or glue code.

Strong static type safety: All semantics should be defined for all syntax.

No modification or duplication: Existing language specifications and implementations should neither be modified nor duplicated.

Separate compilation: Compiling a new language (e.g., syntactic extension or new semantics) should not encompass re-compiling the original syntax or semantics.

Independent extensibility: It should be possible to combine and used jointly language extensions (syntax or semantics) independently developed.

Moreover, the distinction between the specification and the implementation of new engineered languages raises a new concern regarding **automatic composition** [8]. Indeed, "glue code", i.e., code dedicated solely to the interconnection of extensions, must be limited or avoided from the user's point of view to compose a collection of language extensions.

The LEP in Practice

Numerous approaches have been explored in the past decade to address specific scenarios of languages extension, either at the specification level (e.g., [2,4,6,9,11,13,17,19,20,22]) or at the implementation level (e.g., [7,16,22,23]). The specification level is based on meta-languages that provide the relevant abstractions, often with limited and domain-specific expressiveness. The specification level is then turned into an implementation thanks to compilation or interpretation, often by targeting general-purpose programming languages and following language implementation patterns specific to each approach.

While all those approaches are heterogeneous and conceptually operate at different levels, they share common extension mechanisms which are summarized in the five approaches depicted in Fig. 1 [5].

Complying exhaustively to the identified constraints is extremely challenging, and trade-offs must be considered for a given context. We present a selection of scenarios illustrating such trade-offs. First, the constraint of separate compilation usually impacts other non-functional properties such as performance, readability, and accidental complexity (e.g., large and complex glue code, unclear modules dependencies). Consequently, it can be worthwhile to relax the separate compilation constraint in order to comply with other non-functional properties. Second, various actors can be responsible for language extension. They each come with very different skills, ranging from SLE experts with a deep understanding of languages and language workbenches internals, to end users with minimal knowledge of software development. While the former is capable of performing composition using complex handwritten glue specifications, the latter will typically require fully automatic composition approaches. Finally, the boundaries of a language family are important to consider. Two statuses can be considered, closed (i.e., all its languages are known) and open (i.e., new languages can be added organically). Indeed, in the context of closed families, the compatibility of the extensions can be checked in advance and conforming to the type-safe constraints is not an issue. On the contrary, in the context of open language families, restrictive type systems can lead to difficulty or impossibility to extend languages in unanticipated contexts.

The constraints of the *Language Extension Problem* define a framework for comparing language extension approaches. It is worth noting that conforming or not to some of the constraints is often the consequence of interesting language design choices, relative to some specific scenarios.

Conclusion

In this column, we describe the *Language Extension Problem*, a lift of the *Expression Problem* at the language level. We lift the constraints drawn from the *Expression Problem* to the context of software language engineering, and introduced an additional constraint specific to this context. Through the *Language Extension Problem*, we hope to provide a framework to reason on language extension and its challenges and help the comparison of existing and future SLE contributions.

References

1. Biboudis, A., Inostroza, P., Storm, T.v.d.: Recaf: Java dialects as libraries. ACM SIGPLAN Notices **52**(3), 2–13 (2017)
2. Butting, A., Eikermann, R., Kautz, O., Rumpe, B., Wortmann, A.: Systematic composition of independent language features. Journal of Systems and Software **152**, 50–69 (2019). DOI 10.1016/j.jss.2019.02.026
3. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: A meta-language for modular and reusable development of dsls. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pp. 25–36. ACM (2015)
4. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Safe Model Polymorphism for Flexible Modeling. Computer Languages, Systems and Structures **49**, 30 (2016). DOI 10.1016/j.cl.2016.09.001
5. Erdweg, S., Giarrusso, P.G., Rendel, T.: Language composition untangled. In: A. Sloane, S. Andova (eds.) International Workshop on Language Descriptions, Tools, and Applications, LDTA '12, Tallinn, Estonia, March 31 - April 1, 2012, p. 7. ACM (2012). DOI 10.1145/2427048.2427055
6. Erdweg, S., Rendel, T., Kastner, C., Ostermann, K.: SugarJ: Library-based syntactic language extensibility. In: Proceedings of ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languags, and Applications (OOPSLA), pp. 391–406. ACM (2011). DOI 10.1145/2048066.2048099
7. Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: A case of visitor versus interpreter pattern. In: J. Bishop, A. Vallecillo (eds.) Objects, Models, Components, Patterns - 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6705, pp. 228–243. Springer (2011). DOI 10.1007/978-3-642-21952-8_17
8. Kaminski, T., Kramer, L., Carlson, T., Van Wyk, E.: Reliable and automatic composition of language extensions to C: the ablec extensible language framework. PACMPL **1**(OOPSLA), 98:1–98:29 (2017). DOI 10.1145/3138224
9. Kaminski, T., Van Wyk, E.: Modular well-definedness analysis for attribute grammars. In: Proceedings of the 5th International Conference on Software Language Engineering (SLE), *Lecture Notes in Computer Science*, vol. 7745, pp. 352–371. Springer Verlag (2012). DOI 10.1007/978-3-642-36089-3_20
10. Klint, P., Van der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on, pp. 168–177. IEEE (2009)
11. Leduc, M., Degueule, T., Combemale, B.: Modular language composition for the masses. In: D. Pearce, T. Mayerhofer, F. Steimann (eds.) Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2018, Boston, MA, USA, November 05-06, 2018, pp. 47–59. ACM (2018). DOI 10.1145/3276604.3276622
12. Leduc, M., Degueule, T., Combemale, B., Van der Storm, T., Barais, O.: Revisiting visitors for modular extension of executable dsmls. In: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 112–122. IEEE (2017)
13. Mernik, M.: An object-oriented approach to language compositions for software language engineering. Journal of Systems and Software **86**(9), 2451–2464 (2013). DOI 10.1016/j.jss.2013.04.087
14. Oliveira, B.C.d.S.: Modular visitor components. In: European Conference on Object-Oriented Programming, pp. 269–293. Springer (2009)
15. d. S. Oliveira, B.C., Cook, W.R.: Extensibility for the masses - practical extensibility with object algebras. In: ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings, pp. 2–27 (2012)
16. Torgersen, M.: The expression problem revisited. In: M. Odersky (ed.) ECOOP 2004 – Object-Oriented Programming, pp. 123–146. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
17. Vacchi, E., Cazzola, W.: Neverlang: A framework for feature-oriented language development. Computer Languages, Systems & Structures **43**, 1–40 (2015). DOI 10.1016/j.cl.2015.02.001
18. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an extensible attribute grammar system. Science of Computer Programming **75**(1–2), 39–54 (2010). DOI 10.1016/j.scico.2009.07.004
19. Voelter, M.: Language and IDE modularization and composition with MPS. In: Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011, Braga, Portugal, July 3-9, 2011. Revised Papers, pp. 383–430 (2011). DOI 10.1007/978-3-642-35992-7_11
20. Wachsmuth, G., Konat, G.D.P., Visser, E.: Language design with the spoofax language workbench. IEEE Software **31**(5), 35–43 (2014). DOI 10.1109/MS.2014.100
21. Wadler, P.: The expression problem (1998). URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Posted on the Java Genericity Mailing List, 12 November 1998
22. Zenger, M., Odersky, M.: Independently extensible solutions to the expression problem. Tech. rep. (2004)
23. Zhang, W., d. S. Oliveira, B.C.: EVF: an extensible and expressive visitor framework for programming language reuse (artifact). DARTS **3**(2), 10:1–10:2 (2017). DOI 10.4230/DARTS.3.2.10