Typing the Evolution of Variational Software

Luís Afonso Carvalho[1,2], João Costa Seco[1,2], and Jácome Cunha[1,3]

[1] NOVA LINCS
[2] Universidade NOVA de Lisboa
la.carvalho@campus.fct.unl.pt
joao.seco@fct.unl.pt
[3] Universidade do Minho
jacome@di.uminho.pt

Maintaining multiple versions of a software system is a laborious and challenging task, which is many times a strong requirement of the software development process. Such hurdle is justified by needs of backward compatibility with libraries or existence of legacy equipment with particular constraints. It is also an intrinsic requirement of software product lines that target multiple target platforms, service, or licensing levels [7].

A crucial example of a high variability context is an operating system where hundreds of variants need to be maintained to cope with all the different target architectures [1]. We find another important example in mobile applications, where server and client code need to be updated in sync to change structure of the interface or the semantics of webservices. However, it is always the case that older versions of server code must be maintained to support client devices that are not immediately updated. The soundness of a unique and common code corpus demands a high degree of design and programming discipline [8], code versioning, branching and merging tools [12], and sophisticated management methods [11, 9]. For instance, in loosely-coupled service-oriented architectures, where the compatibility guaranties between modules are almost non-existent, special attention is needed to maintain the soundness between multiple versions of service end-points (cf. Twitter API [13]).

Another issue regarding variability is the evolution of software. Arguably, existing language-based analysis tools for service orchestrations do not really account for evolution [14]. Nevertheless, there are other language- and type-based approaches that focus on dynamic reconfiguration and evolution of software [3, 4], hot swapping of code [10], and variability of software [5], that complement the evolution process with tools, and ensure that each version is sound. However, related versions of a software system usually share a significant amount of code, and there are no true guaranties of the sound co-existence of versions and sound transitions between versions at runtime. Such a need is relevant for monolithic software that must provide different versions in the same code base, and it is crucial in the context of service-based architectures. We have presented prior work to check the soundness of service APIs and the runtime transition between versions [2]. However, special hand crafted code was needed to maintain the semantic coherence of the versions of the state. Hence, we believe that the potential impact of a language-based tool supporting variability and a sound co-existence of versions is very high. By checking incremental evolution development it provides gains in safety and increases developer productivity.

Our approach is thus to provide a lightweight formal platform to solve the problem of multiplicity of code versions, while ensuring that the correct state transformations are executed when crossing contexts from one version to another. Our approach is a generalization of the main idea in [2] that keeps all versions well-typed at one given time. We consider one source file containing the code for all versions, and analysed as a whole. Versions and transitions between versions are made explicit in this model, as to represent code evolution steps. This code base is an analogy for a view over the entire history of a versioned code repository. Such a view

can be navigated with the help of a smart development environment that allows a developer to navigate in time, and identify errors in the evolution process.

We extend Featherweight Java (FJ) [6] with a type discipline that ensures that the evolution of state and functionality is captured and analysed. In a versioned FJ program, each element of a class is declared in a specific version context, and each expression is typed and executed with relation to a given version. Special key versions are used to mark state snapshots, where state variables and method types can change. Regular versions allow for the implementation of methods to be changed while maintaining their signature. Class constructors are used to define typed lenses between versions, declaring how an object (state) is legally translated from one version to another. Version contexts are tracked and transitions are only possible if there is a declared state transition. Any illegal version context crossing is dimmed as a typing error.

Such a type-based approach to the problem of maintaining multiple versions of a code base paves the ground for software construction and analysis tools that operate on main-stream languages and supporting runtime environments. Standard subject reduction results ensure that the ecosystem of versions is well formed and that any "view" on the code base is sound.

This work is supported by NOVA LINCS UID/CEC/04516/2013, COST CA15123 - EUTYPES and FC&T Project CLAY - PTDC/EEI-CTP/4293/2014.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 42 variability bugs in the linux kernel: A qualitative analysis. In *International Conference on Automated Software Engineering*, 2014.

[2] João Campinhos, João Costa Seco, and Jácome Cunha. Type-safe evolution of web services. In *Workshop on Variability and Complexity in Software Design, VACE@ICSE 2017*, 2017.

[3] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In *Symposium on Programming Programming Languages and Systems (ESOP 2006)*.

[4] Miguel Domingues and João Costa Seco. Type Safe Evolution of Live Systems. In *Workshop on Reactive and Event-based Languages & Systems (REBLS'15)*, 2015.

[5] Martin Erwig and Eric Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, 2011.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[7] Software Engineering Institute. Software product lines. www.sei.cmu.edu/productlines, 2016-8-16.

[8] Piotr Kaminski, Marin Litoiu, and Hausi Müller. A design technique for evolving web services. In *Conf. of the Center for Advanced Studies on Collaborative Research*.

[9] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-oriented domain analysis feasibility study. Technical Report CMU/SEI-90-TR-021, SE Institute, Carnegie Mellon University, 1990.

[10] Luís Pina, Luís Veiga, and Michael Hicks. Rubah: Dsu for java on a stock jvm. *SIGPLAN Not.*, 49(10):103–119, October 2014.

[11] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[12] Nayan B. Ruparelia. The history of version control. *SIGSOFT Softw. Eng. Notes*, 35(1):5–9, January 2010.

[13] Twitter Inc. Twitter API. https://developer.twitter.com.

[14] Hugo T. Vieira, Luís Caires, and João C. Seco. The conversation calculus: A model of service-oriented computation. In *European Conference on Programming Languages and Systems (ESOP 2008)*.