Incremental Coverage of Legacy Software Languages

Extended Abstract

Vadim Zaytsev
Raincode Labs
Kazernestraat 45
Brussels 1000, Belgium
vadim@grammarware.net

Abstract

Legacy software systems were often written not just in programming languages typically associated with legacy, such as COBOL, JOVIAL and PL/I, but also in decommissioned 4GLs. Writing compilers and other migration and renovation tools for such languages is an active business that requires substantial effort but has proven to be a successful strategy for many cases. However, the process of covering such languages is filled with unconventional requirements and limitations: the lack of useful documentation, large scale of codebases, counter-intuitive language engineering principles, buggy reference implementations, etc.

In this paper, we motivate the incremental nature of software language engineering when it concerns legacy languages in particular, and outline a few related challenges.

Keywords legacy software systems, fourth generation programming languages, compiler construction

ACM Reference format:
Vadim Zaytsev. 2017. Incremental Coverage of Legacy Software Languages. In *Proceedings of the Third Edition of the Programming Experience Workshop, Vancouver, Canada, October 2017 (PX/17.2),* 3 pages.

1 Introduction

There are hundreds of software languages currently in use, and only a few dozens of them are in widespread use. For the rest, it is possible to find systems written in those languages, deployed in ecosystems characteristic for them and being worked on by developers familiar with the fine details of language constructs and context, but it is not realistic to hire new engineers familiar with the language, unless they are switching from one such non-mainstream project to another. Most of these not-widespread languages are prototypical, experimental or domain-specific, which means that they were meant to cater for very peculiar needs of a limited group of people solving specific problems. The remaining ones are what people refer to as *legacy* languages.

Many languages that became legacy, were once called *fourth-generation programming languages* (4GLs), the name signifying the fact that they came after the first generation (machine codes), second generation (assemblers) and third generation (compiled ones). Basically, 4GLs are DSLs (domain-specific languages) designed around the 1980s before powerful language workbenches and other software language engineering technologies made the process easy and (to some extent) error-proof. Programs in a 4GL are typically compiled into a 3GL and then fed into a standard compiler, possibly complemented by code fragments written directly in the 3GLs (and sometimes 2GLs [2]) for performance and expressiveness reasons. Each 4GL is usually known by a small and steadily shrinking group of people and used inside companies that have their entire business logic expressed in that particular 4GL. Eventually every 4GL is decommissioned, and then the companies using it are faced with a choice of migration, re-engineering or bankruptcy.

One of the possible ways out of a decommissioned 4GL is redevelopment of its compiler with modern technologies, and the author of this paper works for a company regularly providing such services to paying customers. A new compiler requires a substantial effort to become operational, since even writing a quality parser for a COBOL-like language takes 2–3 years, according to professionals [16]. However, it provides numerous advantages such as modern deployment platforms (e.g., cloud), modern IDEs (e.g., Eclipse or Visual Studio), as well as the opportunity to evolve the language further in a desired direction—the combination of these offers a threatened company a very bright future.

Besides classic compiler design and development challenges elaborately articulated in many books [6], compiler engineering for legacy languages faces others such as the lack of documentation (which either does not exist, or is outdated beyond usefulness, or may not be used for legal reasons) or a mixture of language rules (imposed by the original compiler) and company conventions (imposed by in-house coding style manuals). In the proposed talk we can discuss some of them, with details sketched below.

The process of incrementally covering a legacy software language goes as follows:

- there is a legacy language, represented by a complete codebase of the customer company and complemented by the knowledge of the in-house experts;

PX/17.2, October 2017, Vancouver, Canada
2017. ACM ISBN 123-4567-24-567/08/06…$15.00

- there is a grammar that covers some of the language and is being continuously worked and improved on;
- one of the failing code samples is considered by the grammar engineer who changes the grammar to accommodate it;
- the cycle is repeated until the entire codebase can be parsed and compiled (or migrated).

The idea of incremental grammar engineering was conceptually proposed by Klint et al. [12], there is a case study reported in detail [1], there are techniques to converge multiple imperfect grammars [15] and many others that can also be useful. These techniques can be either applied directly or used from a language workbench [3, 5] such as Rascal [13], Spoofax [9], ASF+SDF Meta-Environment [11], JastAdd [7], Xtext [4], Intentional [19] or MPS [22]. As a more recent example of a realistic combination of grammar engineering, tool building and static program analysis, we refer to [21]. However, the incremental aspect is not researched well enough to provide tool support for it (which would go beyond conceptual process guidelines). There is research on incremental parsing, which is usually a combination of semiparsing [23] and ad hoc algorithms of selective or lazy parsing. However, they solve a much less painful problem of reducing parse time, as opposed to reducing compiler development time, which has substantially bigger impact on developers' lives. It is possible for powerful parallel parsing techniques in combination with bug prioritisation and error clustering to become somewhat useful at some point, but not in the near future.

2 Research Challenges

We identify at least the following research challenges:

Regression Parsing

Similar to regression testing [18] we would like to be able to quickly and incrementally check if the change in the grammar has had any detrimental effect on the standing coverage of the language. Reparsing the entire codebase is definitely an option, but the usual scale of legacy portfolios is in the tens or hundreds of millions of lines of code, which unnecessarily brings up scalability and performance challenges toward the very beginning of the project. (During later stages of the project this typically takes place on a separate server running the nightly build system).

Test Suite Inference from Codebase

The current practice of regression parsing is to approximate the codebase with a test suite which must be gradually designed by hand and coevolve [14] with each discovered detail of the language. Alternatively, this can be done automatically by a some combination of test prioritisation [20], test suite minimisation [21], as well as machine learning techniques based on the assumption of the naturalness of software [8] which was shown to yield positive results in the past with language identification [10] and works very effectively with position-sensitive pattern languages [24].

Dependency Analysis within the Grammar

Many incremental techniques can be based on the knowledge of dependencies existing within the grammar, since it enables fast and cheap impact analysis of the changes. There is a substantial body of techniques and tools on change impact analysis [17], but to the best of our knowledge, they have not yet been extended to grammars, parsing specifications and other compiler-building models.

Dependency Analysis on the Grammar vs. Samples

Alternatively or complementary, we can investigate and develop algorithms for tracing dependencies between the codebase (or some representation thereof like a repository of parse trees) and the grammar, and updating the parsability metadata incrementally whenever needed or whenever the CPU is idle, to provide the results of regression/impact analysis to the grammar engineer.

Neighbour Analysis

A lot of grammar development cycles are lost on tiny tweaks of features: can you only print a variable or can it be any expression? is assignment target always one identifier or can it be multiple? can you call a method on the result of call of another method? is there an `else` clause to the `if`? how does a default clause of the `caseof` look like? The answers to all these questions in mainstream languages are either straightforward or easily obtained from language manuals. For legacy languages, we must follow the trial and error approach. However, it should be possible, given both the original tested grammar and the changed one, and detailed enough information about the language coverage of both, to identify and show examples of "near misses"—cases that were bluntly rejected by the original grammar and still fail in the new one but "later" or "better" by some definition.

3 Conclusion

We have raised a question of developing a systematic approach of covering legacy software languages incrementally, and outlined a few challenges to be solved in developing, testing, validating and deploying parsers and other form of grammarware. There do not seem to be major roadblocks in this research direction, but engineering and architectural challenges are substantial.

References

[1] Tiago Laureano Alves and Joost Visser. 2008. A Case Study in Grammar Engineering. In *Revised Selected Papers of the First International Conference on Software Language Engineering (SLE) (LNCS)*, Vol. 5452. Springer, 285–304. https://doi.org/10.1007/978-3-642-00434-6_18

[2] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. 2016. Tool Demo: Raincode Assembler Compiler. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*, Tijs van der Storm, Emilie Balland, and Dániel Varró (Eds.). 221–225. https://doi.org/10.1145/2997364.2997387

[3] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches — Conclusions from the Language Workbench Challenge. In *Proceedings of the Sixth International Conference on Software Language Engineering (SLE) (LNCS)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[4] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement Your Language Faster than the Quick and Dirty Way. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 307–309. https://doi.org/10.1145/1869542.1869625

[5] Martin Fowler. 2005. Language Workbenches: The Killer-App for Domain Specific Languages? MartinFowler.com. (June 2005). https://martinfowler.com/articles/languageWorkbench.html

[6] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J. H. Jacobs, and Koen G. Langendoen. 2012. *Modern Compiler Design* (second ed.). Addison-Wesley. https://dickgrune.com/Books/MCD_2nd_Edition/

[7] Görel Hedin and Eva Magnusson. 2003. JastAdd — An Aspect-Oriented Compiler Construction System. *Science of Computer Programming (LDTA'01 Special Issue)* 47, 1 (2003), 37–58. https://doi.org/10.1016/S0167-6423(02)00109-0

[8] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Martin Glinz, Gail C. Murphy, and Mauro Pezzé (Eds.). IEEE, 837–847. https://doi.org/10.1109/ICSE.2012.6227135

[9] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *Proceedings of the 25th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[10] Juriaan Kennedy van Dam and Vadim Zaytsev. 2016. Software Language Identification with Natural Language Classifiers. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering: the Early Research Achievements track (SANER ERA)*, Katsuro Inoue, Yasutaka Kamei, Michele Lanza, and Norihiro Yoshida (Eds.). IEEE, 624–628. https://doi.org/10.1109/SANER.2016.92

[11] Paul Klint. 1993. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 2 (1993), 176–201.

[12] Paul Klint, Ralf Lämmel, and Chris Verhoef. 2005. Toward an Engineering Discipline for Grammarware. *ACM Transactions on Software Engineering Methodology (TOSEM)* 14, 3 (2005), 331–380. https://doi.org/10.1145/1072997.1073000

[13] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[14] Ralf Lämmel. 2016. Coupled Software Transformations—Revisited. In *Proceedings of the Ninth International Conference on Software Language Engineering (SLE)*. ACM, 239–252. https://doi.org/10.1145/2997364.2997366

[15] Ralf Lämmel and Vadim Zaytsev. 2009. An Introduction to Grammar Convergence. In *Proceedings of the Seventh International Conference on Integrated Formal Methods (iFM 2009) (LNCS)*, Michael Leuschel and Heike Wehrheim (Eds.), Vol. 5423. Springer, 246–260. https://doi.org/10.1007/978-3-642-00255-7_17

[16] Vadim Maslov. 1998. Re: An Odd Grammar Question. http://compilers.iecc.com/comparch/article/98-05-108. (May 1998).

[17] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. 2004. Chianti: A Tool for Change Impact Analysis of Java Programs. In *Proceedings of the 19th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 432–448. https://doi.org/10.1145/1028976.1029012

[18] Gregg Rothermel and Mary Jean Harrold. 1997. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 2 (1997), 173–210. https://doi.org/10.1145/248233.248262

[19] Charles Simonyi, Magnus Christerson, and Shane Clifford. 2006. Intentional Software. In *Proceedings of the 21th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Peri L. Tarr and William R. Cook (Eds.). ACM, 451–464. https://doi.org/10.1145/1167473.1167511

[20] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively Prioritizing Tests in Development Environment. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 97–106. https://doi.org/10.1145/566172.566187

[21] Nicole Vavrová and Vadim Zaytsev. 2017. Does Python Smell Like Java? *The Art, Science and Engineering of Programming (‹Programming›)* 1 (April 2017), 11–1–11–29. Issue 2. https://doi.org/10.22152/programming-journal.org/2017/1/11

[22] Markus Völter and Vaclav Pech. 2012. Language Modularity with the MPS Language Workbench. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 1449–1450. https://doi.org/10.1109/ICSE.2012.6227070

[23] Vadim Zaytsev. 2014. Formal Foundations for Semi-parsing. In *Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements Track (CSMR-WCRE 2014 ERA)*, Serge Demeyer, Dave Binkley, and Filippo Ricca (Eds.). IEEE, 313–317. https://doi.org/10.1109/CSMR-WCRE.2014.6747184

[24] Vadim Zaytsev. 2017. Parser Generation by Example for Legacy Pattern Languages. In *Proceedings of the 16th International Conference on Generative Programming: Concepts and Experience (GPCE)*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, 212–218. https://doi.org/10.1145/3136040.3136058