4.2 Multidirectionality in Compiler Testing

Vadim Zaytsev (Raincode Labs, BE)

A classical software language processor can be viewed as a chain of transformations, most of them even unidirectional, going through most of the following intermediate artefacts [1]:

- Program text
- Preprocessed program text
- Parse tree as a structural model of a program
- Abstract syntax graph as a conceptual model
- Annotated graph with types and other information
- Code model suitable for optimisations
- Executable code
- Computation result

Each of these artefacts/models conforms to a different metamodel. Examples of bidirectional transformations in this chain, are:

- Error correction facilities [2], where a "later" and more rich artefact can be used to point out errors in an "earlier" and more primitive artefact, such as misplaced punctuation or parenthesis in the text of the program.
- Semantic-driven disambiguation [3], where the structure or a model of a program can only be decisively determined after semantic analysis. The need and necessity for such techniques is caused by having constructs like "`x * y`" in C (which can either mean declaring a variable `y` typed as a pointer to a value of type `x`, or a multiplication of two variables named `x` and `y`), dangling clauses in COBOL (a language where it is not always straightforward to determine where one statement ends and the next one begins), offside rule in languages like SASL, Python or Haskell (where statement affiliation with a block depends on the indentation of a piece of code), as well as various ambiguities in 4GLs caused by bad language design [4].
- Incremental techniques where the change that needs to be propagated in either direction, is several orders of magnitude smaller than the entire model. For example, many legacy systems have flat hierarchies of interlinked and intercommunicating entities spanning over millions of lines of code, but the evolution they undergo on a daily basis, covers small scale bug fixes, rarely even multiline. Implementations of incremental synchronisation techniques usually involve some sort of bx.

At Raincode Labs, which is commonly employed as a team of compiler mercenaries, we are being asked to implement some of these features regularly, so having some bx is a norm rather than something exotic.

A typical compiler test is a tuple, which elements correspond to some of the artefacts listed in the beginning of this section. In the simplest case, it is a tuple with a program text and its expected execution results. However, such simplistic test cases are only useful with mature projects [5]. Compilers under active development require a much more elaborate framework for testing, capable of forming hypotheses, crystallising them as specifications and testing them differentially on available oracles (such as working legacy implementations or remaining living domain experts). It is not uncommon for such a test spec to include all or almost all of the artefacts, allowing for testing whether the parser could recognise the

input as correct, whether it succeeded building a proper parse tree, whether in its turn a corresponding syntax graph was constructed correctly, etc, all the way to the execution of the compiled code and comparing the result with the baseline [4, 5]. In practice it helps enormously to have the capability to locate the exact point of failure.

So, since a test case is an n-tuple, a collection of them (known as a test suite) can be seen as a specification of an n-ary relationship. When it gets broken (by a change in a compiler, or, even more commonly during development, by the customer providing additional information that conflicts with the contemporary understanding of the intended language semantics), it needs to be restored, and that can/should be done by a multx. In general, all connected artefacts are needed as inputs to make a consistency restoration decision, and all of them have a chance to be changed as its result.

Unfortunately, the state of the art is to accomplish this with a combination of manual programming and bespoke proprietary tools. The main intention behind exposing this case study during the seminar as well as in this report, is to provide a somewhat detailed description of an open problem that seems suitable to be solved with multx.

References

1 Vadim Zaytsev, Anya Helene Bagge. *Parsing in a Broad Sense*, Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 8767, pp. 50–67, https://doi.org/10.1007/978-3-319-11653-2_4, Springer, 2014.

2 Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, Emma Söderberg. *Natural and Flexible Error Recovery for Generated Modular Language Environments.* ACM Transactions on Programming Languages and Systems, 34(4): 15:1-15:50, https://doi.org/10.1145/2400676.2400678 2012.

3 Mark van den Brand, Steven Klusener, Leon Moonen, Jurgen J. Vinju. *Generalized Parsing and Term Rewriting: Semantics Driven Disambiguation.* Proceedings of the Third Workshop on Language Descriptions, Tools and Applications (LDTA), ENTCS 82(3), https://doi.org/10.1016/S1571-0661(05)82629-5, 2003.

4 Vadim Zaytsev. *Open Challenges in Incremental Coverage of Legacy Software Languages.* Proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2), pp. 1–6, https://dl.acm.org/citation.cfm?id=3167105, 2017.

5 Vadim Zaytsev. *An Industrial Case Study in Compiler Testing.* Proceedings of the 11th International Conference on Software Language Engineering (SLE), pp. 97–102, https://doi.org/10.1145/3276604.3276619, ACM, 2018.