

Impact of Software Transformation Systems on Language Workbenches and Domain-Specific Language Tools.

Martin Bravenboer, martin@cs.uu.nl

In 2002, Todd Proebsting gave a talk on “disruptive programming language technologies”. The talk was motivated by the series of questions Richard Hamming used to ask during his lunches at Bell Labs: “What are the important problems of your field?” and “What important problems are you working on?”. These questions inspired Proebsting to analyse the field of programming languages research, whereafter he concluded that research on compiler optimization is least important and improving programmer productivity based on language design is most important. His business plan for researchers was to design languages to solve real problems, and sneak in research ideas as ‘trojan horses’.

Proebsting also mentioned program generation as an option for improving the productivity of programmers. At least at the time, he was skeptical about program generation, because of its narrow applicability. However, in the last few years, the combination of language design and program generation is being proposed by research and industry for increasing productivity. Of course, generative programming in general has been around, but more concretely the combination of language design and program generation is most prominent in Ruby (on Rails), Microsoft Domain Specific Language Tools (Software Factories), JetBrains Meta-Programming System (Language Oriented Programming), the discussion on Language Workbenches (Martin Fowler), Intentional Software, not to mention model-driven engineering. We still have to await how these efforts will work out, but the widespread interest in the direction of language workbenches and program generation is quite impressive.

Impact. The software transformation system community has not been very successful at contributing to this field, either by filling the gaps ourselves, by having influence on the proposed solutions, or by adding interesting trojan horses to generally less interesting solutions. Frankly, it is quite sad that software transformation systems (such as ASF+SDF Meta-Environment, TXL, DMS, and Stratego/XT) are not even mentioned as one of the possible candidates for (components of) languages workbenches. Sometimes, parser generation tools like Yacc are mentioned, and in the most positive case there was a discussion of ANTLR. Usually, the topic of conventional language implementation is quickly abandoned by concluding that this is all pretty complex.

Many transformation systems are not really just about software transformation, but rather provide a versatile system for defining the syntax (and sometimes semantics) of the languages that are subject to transformation. Indeed, software transformation systems are prominent in the research on parsing and declarative specification of the syntax of languages. Hence, software transformation systems should be a number one candidate to facilitate language oriented programming. It would be unfair to blame the ‘movement’ of language oriented programming for their lack of interest in transformation systems. Rather, we should reflect over our work to find out how we can increase the impact of transformation systems. In this talk, I will discuss some of the issues from my point of view. Most of the issues I’d like to raise make creating DSLs with software transformation systems it too expensive. Furthermore, the workshop format will hopefully result in a lively discussion on this topic with new perspectives and maybe even new initiatives.

Commonalities. The first problem is that software transformation systems are not visible to the outside world as a community that has things in common. Where should someone go if he wants to learn about transformation systems? Are there principles and theory that we have in common? After

discovering that there is not much to learn about transformation systems in general, the first thing a new user has to do is to pick a software transformation system. However, the differences between the various systems are very unclear (even to us), so it is difficult to make a well-informed decision. The choice for a particular system is often based on the available support for a specific subject language, which leads to the next problem.

Language Support. Transformation systems have spent substantial effort on making the definition of the syntax of languages as declarative as possible. Yet, this advancement has had very little impact on the practice of grammars and parsing, as is illustrated by the still omnipresent `lex` and `yacc`, which are used to compare the tool-oriented language workbenches to conventional solutions. To improve the practice of language definition, the first thing to do is to share language specific support. Transformation systems have their own (usually limited) set of grammars, even if the used parsing techniques are very similar. Beyond grammars, everything becomes even more system specific, while semantic analysis is often essential for implementing transformations and generators. In practice, this language specific support appears to be one of our most valuable assets. To increase the set of supported languages, to improve the quality, and to make the choice for a particular method of software transformation easier, we need to share and reuse the basic support for languages. An additional advantage is that the methods for implementing this will get more attention. We will probably be able to improve our current methods, but even more important is that it will become acceptable to spend time on learning how to work with them. This will decrease the learning curve and will increase visibility.

Also, the impact of transformation systems can be increased in the area of language specification. For example, `Stratego/XT` is currently being used as the underlying system for defining the semantics (and perhaps the syntax) of the next version of ECMAScript. Also, we can propose better specifications ourselves, which can even be a research contribution, as is illustrated by our AspectJ syntax paper at OOPSLA'06.

Being On The Radar. Being on the radar is not just about marketing, but even more about clearly organizing the solution space. Currently, there is no dominant transformation system. This is a problem, since outsiders cannot have every single system on their radar. By increasing the number of components that we have in common, such as language specific support, we can at least achieve that these components become visible. A single toolset for the basic infrastructure of every software transformation system is most useful. Concerning marketing, several things can be improved as well. There is no book on software transformation, there is no involvement in this the discussion on language workbenches, most of us do not have a blog, and we do not write articles for the mainstream developer.

Research Thrills. The new applications of software transformation also ask for a reconsideration of the topics that need to be researched. While the generative programming community has spent a lot of effort on topics like static guarantees, syntactical embeddings, and hygiene, the current practice of program generation largely ignores these advancements. For example, run-time code generation in Ruby is technically quite poor, yet nobody seems to care. What developers do care about is not losing the productivity they have just gained by using program generation. For example, an IDE that understands the mixture of languages that are involved, learning curves, avoiding lock-in, good error reporting, and the maturity of the systems. These might be mostly engineering aspects of language workbenches, yet research on this will be of the uttermost importance.