People-Specific Languages: a case for automated programming language generation by reverse-engineering programmer minds

Poss, R.

Publication date
2014

Document Version
Final published version

Published in
Extended abstracts of the 2nd International Workshop on Open and Original Problems in Software Language Engineering: OOPSLE 2014: Antwerpen, Belgium, February, 2014

Extended Abstracts

of the

2th International Workshop on Open and Original Problems in Software Language Engineering,

OOPSLE 2014

Anya Helene Bagge & Vadim Zaytsev (eds.)

Antwerpen, Belgium, February, 2014.

People-Specific Languages: A case for automated programming language generation by reverse-engineering programmer minds

Raphael 'kena' Poss

r.poss@uva.nl
Institute for Informatics
University of Amsterdam, The Netherlands

Abstract: The innovation of DSLs was the recognition that each application domain has its few idiomatic patterns of language use, found often in that domain and rarely in others. Capturing these idioms in the language design makes a DSL and yields gains in productivity, reliability and maintainability. Similarly, different groups of programmers have different predominant cognitive quirks. In this article I argue that programmers are attracted to some types of languages that resonate with their quirks and reluctant to use others that grate against them. Hence the question: could we tailor or evolve programming languages to the particular personality of their users? Due to the sheer diversity of personality types, any answer should be combined with automated language generation. The potential benefits include a leap in productivity and more social diversity in software engineering workplaces. The main pitfall is the risk of introducing new language barriers between people and decreased code reuse. However this may be avoidable by combining automated language generation with shared fundamental semantic building blocks.

Keywords: People-Specific Languages, automated language generation, meta-meta-programming

1 Background

In the fall of 2013, HASTAC[1] student Arielle 'Ari' Schelsinger caused a commotion on the Interwebs by announcing her research question: what would a *feminist programming language* look like? [Sch13] Regardless of Ari's particular approach, I found her hypothesis intruiguing: that programming languages are primarily designed by men for men, and that their deep structure may be favorable to the perpetuation of thought patterns that alienate women.

In a situation closer to home, intellectual sparring partner Merijn Verstraaten and I have been arguing loudly in the period 2011-2012 about whether the lack of interest for operational semantics by most functional programmers is a feature (his view) or a bug (mine). It took us nearly one year until we homed on the resolution of our argument: for some *people*, the core challenge of the programming *activity* is overcoming human limitations: how to express oneself clearly, derive a clear specification for a solution, and gaining confidence that this specification answers a particular problem; whereas for some other people, the core challenge of the activity is overcoming

[1] http://www.hastac.org/

machine limitations: how to drive an artefact of the real world to the edge of its technical abilities, and maximize its extra-functional output, eg. regarding performance & efficiency. To simplify, we concluded that there are programmers who mostly need help from their languages to write "what to do", whereas others mostly need help to write "how to do it". This clustering of interests is psychological in nature, and orthogonal to the kind of problems that programmers solve. This model explains partially the diversity of *flavors* of programming languages, even within well-identified clusters like functional languages. For example, the Haskell's semantic purity will "press the right buttons" of the first population, whereas Clojure's and Racket's celebrated ability to invoke unsafe side effects will appeal to the second population. Although there are some versatile individuals, most programmers are able to phrase which flavor they *prefer*, even if they don't position their choice consciously on this particular scale. Conversely, programmers asked to program in a flavor they do not prefer actually work outside of their intellectual comfort zone and are thus under-productive.

2 People-specific languages

This observation drove the following thought experiment: what would happen if we could model the correlation between features of programming languages and the personal traits of programmers who find themselves "in tune" with the language?

One consequence is obvious: whenever a large-scale problem requires both a software engineering effort and a precise selection of non-software-related human skills, such as the specific combination of graphical, musical, screenwriting and marketing abilities needed to deliver a successful video game, we could select the people first, and *then* select a programming language closest to their cluster of personalities profiles according to the model. This would alleviate the need to have separate teams for art&marketing and programming. It would increase productivity. The practical benefits are directly economically measurable.

The other consequence is more subtle and perhaps surprising: a correlation model between personal traits and language features would identify known clusters ("for a known personality trait, these known language features seem most correlated"), but also *reveal inter-cluster space still left unchartered*. For example, when the personality profile of a person who never programmed before is characterized, it could be that the profile does not fit within the known clusters. By intrapolation of extrapolation from the known clusters, we could derive *new programming languages* with the mix of features from the closest clusters in the model, tuned for this person. If successful, this process would produce what I call *"People-Specific Languages"* (PSLs). This process would encompass and supersede Ari Schelsinger's problem, by automating the selection of features most adapted to a "feminist programmer" after profiling him/her. This would simplify the teaching of programming, as the target users of a PSL would *find their language simple to use and understand, by construction*. It would open software engineering to a more diverse population and quickly empower a large part of the workforce without a technical education and currently unemployed.

But how to construct such a correlation model? To answer this, we could start by observing that both personality types and programming language features have been decomposed in their respective fields in more primitive traits which can be quantified. One such decomposition is

Table 1: Hypothetical correlation model between MBTI types and language features.

MBTI personality feature	Correlated programming language features
Extraversion (E)	dynamic typing
Introversion (I)	static typing
Sensing (S)	nominative typing
Intuition (N)	structural/duck typing
Thinking (T)	domain-specific idioms, orthogonal library
Feeling (F)	small general purpose toolbox, multi-paradigm
Judging (J)	extra weight for features from the T/F group
Perception(P)	extra weight for features from the S/N group

the 4-variable MBTI[2] psychometric model. Programming languages can also be classified by e.g. the features of their type system and their core syntax&semantics. After polling a suitable sample of programmers for their MBTI and preferred languages, we could conduct a principal component analysis and derive correlations betwen specific MBTI types and language features. An hypothetical example resulting model is given in table 1. This example would suggest that Python appeals more to ENTP types, while INFJ types may find more comfort with Haskell. Note however that MBTI is considered here for the sake of simplicity but may be just too simplistic for good results[3]. Another model could correlate other traits entirely, for example respect for authority, the ability to follow procedures, or ethnical background. The actual inter-disciplinary exercise of constructing a model empirically validated using real data is left open for future work.

3 Automated programming language generation

A particular problem of language engineering is the difficulty to design, implement and maintain a new language and its tool chain (incl. compilers, editors, debuggers, etc.). DSL creation, evolution and maintenance is still an activity reserved to a select elite of human experts. The cost of staffing DSL designers/implementers/maintainers is actually a known obstacle to the wider adoption of DSLs, since only few industry players have the financial flexibility to invest both in language design for their specific domain and the subsequent software engineering needed to deliver their products. Unadressed, I believe this cost alone would prevent the appearance of PSLs, since the large diversity of human personalities would mandate a combinatorially larger number of PSLs for each considered group of programmers.

The issue may be alleviated if we consider the opportunity to *automate* the creation of PSLs. In general, there are three phases to language design: 1) *designing* core language features, 2) *selecting* which features to incorporate, then 3) *combining* them to a coherent useable whole with libraries and tools. The design of core features is probably difficult to automate, but arguably it has not been often needed so far as many languages share their core features, reused from a few precursors. Combining features towards a language *after they have been selected* has

[2] Myers-Briggs Type Indicator

[3] In particular one person may have different MBTI profiles depending on social context.

been studied already; for example with Racket one can merge, on demand, language features already available as library modules [TSC+11]. Arguably, the largest remaining obstacle to the mass generation of new languages is the selection phase, but the model-based approach outlined above would enable the automation of this, too.

In practice, automatic PSL generation would require to design a meta-meta-programming framework: a set of tools which can be programmatically recombined to create the various facets of programming tools: syntax, mapping from syntax to semantics, type systems, language documentation, pre-imported library APIs, etc. A *PSL generator* would be a program in this framework which inputs a person's profile and a correlation model, and outputs a PSL's programming tools. Assuming this generator is reasonably fast and complete, PSLs could be created on demand for even small scale projects.

4 Open questions

The vision presented in this position piece will likely require more than a decade of interdisciplinary research before convincing results can be obtained. Any concrete effort to deliver PSLs on demand will face at least the following fundamental questions:

How to quantitatively compare the benefits of a PSL delivered to a person who never programmed before, with this person's potential performance using one or more previously existing language? This is a consequence of the "first use bias": programmers are influenced by the first language(s) they learn, and a PSL produced for an already experienced programmer may not differ significantly from the language(s) they already know and use. Overcoming this experimental obstacle will be key to scientifically validate the benefits of PSLs.

Can we make different PSLs interoperable? This will be key to preserve code reuse: while the current economical pressure for *expertise reuse*, an obstacle to the adoption of DSLs, would be diminished by PSLs, *code reuse* stays crucial for productivity ("avoid solving the same problem twice"). However achieving inter-operability between PSLs requires a common semantico-linguistic system between humans, the existence of which is a known open problem in linguistics.

How to design a meta-meta-language for PSL generation? This language's library, if/when it exists, must necessarily feature at least composable grammars, composable type systems, composable semantics, composable documentation, as building blocks, all of which are still currently research topics. Whether this is at all possible is still an open question of computing theory.

Nevertheless, the opportunities highlighted in this modest introduction should be sufficient to motivate further research. The philosophical, linguistic and sociological insights such exercise would deliver would be ample reward on their own regardless of the eventual success of PSLs.

Bibliography

[Sch13] A. Schlesinger. A Feminist && A Programmer. http://www.hastac.org/blogs/ari-schlesinger/2013/12/13/feminist-programmer, December 2013.

[TSC+11] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen. Languages As Libraries. *SIGPLAN Not.* 46(6):132–141, June 2011. doi:10.1145/1993316.1993514