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.

Contents

Workshop Programme – Monday, February 3rd, 2014

- 9:00 – 9:30 Opening and welcome of OOPSLE'14; overview of OOPSLE'13

- 9:30 – 10:30 (academic keynote) Prof. Dr. Sibylle Schupp (Technische Universität Hamburg-Harburg): Type-Checking the Cyber-Physical World

- 11:00 – 11:30 (paper) Federico Tomassetti, Cristhian Figueroa (Politecnico di Torino), Dr. Daniel Ratiu (fortiss): Tool-automation for Supporting the DSL Learning Process

- 11:30 – 12:00 (paper) Tero Hasu (University of Bergen): Managing Language Variability in Source-to-Source Compilers

- 12:00 – 12:30 (paper) Dr. Raphael Poss (Universiteit van Amsterdam): People-Specific Languages

- 13:30 – 14:30 (industrial keynote) Dr. Ira Baxter (Semantic Designs): "I double-dog dare you..."

- 14:30 – 15:00 (paper) Changyun Huang, Prof. Dr. Naoyasu Ubayashi, Dr. Yasutaka Kamei (Kyushu University): Towards Language-Oriented Software Development

- 15:30 – 16:00 (paper) Dr. Mark Hills (East Carolina University): Capturing Programmer Intent with Extensible Annotation Systems

- 16:00 – 16:30 (paper) Merijn Verstraaten (Universiteit van Amsterdam): Orthogonal and Extensible Type Systems: The Birth of Domain Specific Type Systems?

- 16:30 – 17:30 (discussion)

Introduction – A Workshop on Open and Original Problems in Software Language Engineering

Anya Helene Bagge and Vadim Zaytsev

Abstract: The OOPSLE workshop is a discussion-oriented and collaborative forum for formulating and addressing with open, unsolved and unsolvable problems in software language engineering (SLE), which is a research domain of systematic, disciplined and measurable approaches of development, evolution and maintenance of artificial languages used in software development. OOPSLE aims to serve as a think tank in selecting candidates for the open problem list, as well as other kinds of unconventional questions and definitions that do not necessarily have clear answers or solutions, thus facilitating the exposure of dark data. We also plan to formulate promising language-related challenges to organise in the future.

1 Description

The second international workshop on Open and Original Problems in Software Language Engineering (OOPSLE'14) will follow the first one held at WCRE 2013 in Koblenz. The main focus of the workshop lies in identifying and formulating challenges in the software language engineering field — these challenges could be addressed later at venues like SLE, MODELS, CSMR, WCRE, ICSM and others.

The field covered by the workshop, revolves around "software languages" — all kinds of artificial languages used in software development: for programming, markup, pretty-printing, modelling, data description, formal specification, evolution, etc. Software language engineering is a relatively new research domain of systematic, disciplined and measurable approaches of development, evolution and maintenance of such languages. Many concerns of software language engineering are acknowledged by reverse engineers as well as by forward software engineers: robust parsing of language cocktails, fact extraction from heterogeneous codebases, tool interfaces and interoperability, renovation of legacy systems, static and dynamic code analysis, language feature usage analysis, mining repositories and chrestomathies, library versioning and wrapping, etc.

Some research fields have a list of acknowledged open problems that are being slowly addressed by the community: as examples of such lists, we can recall the Hilbert's problems [Hil02], the POPLmark Challenge [PSWZ08] and a list of open problems in Boolean grammars [Okh13]. However, the field of software language engineering has not yet produced one. This workshop is meant to expose hidden expertise in coping with unsolvable or unsolved problems which commonly remain unexposed in academic publications. OOPSLE aims to serve as a think tank in selecting candidates for the open problem list, as well as other kinds of unconventional questions and definitions that do not necessarily have clear answers or solutions, thus facilitating the exposure of dark data [Goe07]. We also plan to formulate promising language-related challenges to organise in the future. Beside the abovementioned POPLmark Challenge which can also be seen as a collection of benchmarks, there have been many more contests, challenges and competitions related to software language engineering: LDTA Tool Challenge held at the LDTA workshop

in 2011 [LDT11], CodeGeneration-affiliated Language Workbench Challenge [ESV+13] held yearly since 2011, Transformation Tool Contest held six times since 2007 [RV10], Rewrite Engines Competition held three times in 2006, 2008 and 2010 at WRLA [DRB+10], PLT Games held monthly since December 2012 [McK12].

An extensive list of topics encouraged for investigation for workshop participants, can be found at http://oopsle.github.io and the corresponding description of the previous instance [BZ13].

2 Topics

We acknowledge the following list as non-exhaustive collection of examples of topics of interests of the workshop:

- Defining an unsolved problem by establishing both its provenance in prior research and the lack of a fully satisfactory solution.

- Identifying new problem areas in software language engineering that have not been previously studied due to lack of understanding, techniques, practical interest or scalability issues.

- Engaging in technological space travel by identifying similar problems in various sectors of software language engineering (i.e., grammarware, modelware, ontoware, XMLware, databases, spreadsheets, etc).

- Generalising and reformulating of several well-known problems into several sides of one open challenge (e.g., parsing and pretty-printing).

- Proposing systematic methods of assessment and comparison of existing and emerging solutions to a problem that is not or cannot be fully solved (e.g., choosing between parsing techniques, metaprogramming methodologies, software language workbenches).

- Presenting unconventional crossovers of popular research topics and software language engineering concerns (e.g., green IDEs and energy consumption considerations for parsing algorithms).

- Making an overview of major hindrances hindering solution of a standing problem (e.g., tool interoperability and reuse, tackling language and metalanguage diversity and versatility, consistency management).

- Describing novel or unconventional ideas that are promising but are not yet fully validated, or where validation itself may be a challenge.

- Constructing future community experiments and considering topics our community expects to see addressed by such a competition, if one decides to run it in the future.

- Critically reassessing a problem that is widely assumed to be solved but the solution is either underwhelming or could be "considered harmful" (model-driven engineering, domain-specific languages, test-based development).

3 Workshop Format

3.1 Before the Workshop

OOPSLE participants are encouraged to submit position papers up to 4 pages in length, sketching an open or original problem, idea or challenge. The submissions are screened by the workshop chairs, who will select papers based on potential for discussion and interest to the community, as well as the clarity of presentation and motivation — *OOPSLE is not a mini-conference*, and therefore it is not necessary for the work to be conclusive yet. The papers will be posted online prior to the workshop, so the participants have the opportunity to read them in advance.

3.2 At the Workshop

A keynote lecture will be given at the start of the workshop to provide an overview of some existing problems in the field of language engineering and to serve as a starting point for discussion during the rest of the day.

Each accepted position paper is presented at the workshop as a brief summary of its main idea and a set of open questions to be discussed with the audience. Presenters will ask for input on how to proceed with experiments, validation or refinement of their ideas, collect opinions on the presented definitions, share similar experience. We expect the participants to be friendly but inquisitive, and ask hard questions back that may lead to deepening the initially presented insights. The workshop is planned to have short presentations and long discussions to stimulate direct collaboration afterwards.

3.3 After the Workshop

All workshop participants will be invited to submit a full paper to a special issue of the Electronic Communications of the EASST, an open access peer-reviewed journal. Journal submissions will undergo peer review by the members of the program committee consisting of researchers in software language engineering and reverse engineering.

4 Examples

Cossette and Walker [CW12] investigate API migration and conclude that quite often none of known recommender techniques provide any useful advice, and even when they do, the recommendations are correct in only about 20 % of cases.

In 2011, a Tool Challenge was proposed at the LDTA workshop [LDT11], for which participants needed to implement a range of features of the Oberon-0 [Wir96] programming language. Submissions to the challenge were very versatile and gave insights into how the technologies behind them were related to one another.

Klint et al. [KLV05, §7] list 15 research challenges for the domain of grammarware such as having a framework for grammar transformations, comprehensive grammarware testing, modular grammarware development, etc. Since 2005, most of them have been to some extent addressed by papers and tutorials of venues such as MODELS, SLE, WCRE, ICSM, CSMR, etc.

Lämmel [Läm13] has a methodology for identifying several fundamental papers that can be used to define the language engineering field and explain its concepts to outsiders.

Bibliography

[BZ13] A. H. Bagge, V. Zaytsev. Workshop on Open and Original Problems in Software Language Engineering. Oct. 2013.

[CW12] B. E. Cossette, R. J. Walker. Seeking the Ground Truth: a Retroactive Study on the Evolution and Migration of Software Libraries. In *FSE '12*. Pp. 55:1–55:11. ACM, 2012.

[DRB+10] F. Durán, M. Roldán, J.-C. Bach, E. Balland, M. van den Brand, J. R. Cordy, S. Eker, L. Engelen, M. de Jonge, K. T. Kalleberg. The Third Rewrite Engines Competition. In Ölveczky (ed.), *WRLA'10*. LNCS 6381, pp. 243–261. Springer, 2010.

[ESV+13] S. Erdweg, T. van der Storm, M. Völter, M. Boersma, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, J. van der Woning. The State of the Art in Language Workbenches. Conclusions from the Language Workbench Challenge. In Erwig et al. (eds.), *SLE'13*. LNCS 8225. Springer, Oct. 2013.

[Goe07] T. Goetz. Freeing the Dark Data of Failed Scientific Experiments. *Wired Magazine* 15(10), 2007.

[Hil02] D. Hilbert. Mathematical Problems. *Bulletin of the American Mathematical Society* 33(4):433–479, 1902.

[KLV05] P. Klint, R. Lämmel, C. Verhoef. Toward an Engineering Discipline for Grammarware. *ACM TOSEM* 14(3):331–380, 2005.

[Läm13] R. Lämmel. An Annotated and Illustrated Bibliography on Software Language Engineering. Oct. 2013.
http://softlang.uni-koblenz.de/yabib.pdf

[LDT11] LDTA 2011. 11th International Workshop on Language Descriptions, Tools and Applications. Tool Challenge. 2011.
http://ldta.info/tool.html

[McK12] B. McKenna. The Programming Language Theory Games: a monthly programming language competition. Dec. 2012.
http://www.pltgames.com

[Okh13] A. Okhotin. Conjunctive and Boolean Grammars: The True General Case of the Context-Free Grammars. *Computer Science Review* 9:27–59, 2013.

[PSWZ08] B. C. Pierce, P. Sewell, S. Weirich, S. Zdancewic. It Is Time to Mechanize Programming Language Metatheory. In Meyer and Woodcock (eds.), *Verified Software: Theories, Tools, Experiments*. LNCS 4171, pp. 26–30. Springer, 2008.

[RV10]   A. Rensink, P. Van Gorp. Graph Transformation Tool Contest 2008. *International Journal on Software Tools for Technology Transfer (STTT)* 12(3–4):171–181, 2010.

[Wir96] N. Wirth. *Compiler Construction*. International computer science series. Addison-Wesley, 1996.

Academic Keynote: Type-Checking the Cyber-Physical World

Sibylle Schupp

Tool-automation for supporting the DSL learning process

Federico Tomassetti[1]**, Cristhian Figueroa**[1]**, Daniel Ratiu**[2]

[1]Politecnico di Torino,[2]fortiss gGmbH

Abstract: Recent technologies advances reduced significantly the effort needed to develop Domain Specific Languages (DSLs), enabling the transition to language oriented software development. In this scenario new DSLs are developed and evolve at fast-pace, to be used by a small user-base. This impose a large effort on users to learn the DSLs, while DSL designers can use little feedback to guide successive evolutions, usually just based on anecdotal considerations.

We advocate that a central challenge with the proliferation of DSLs is to help users to learn the DSL and providing useful analyses to the language designers, to understand what is working and what is not in the developed DSL.

In this position paper we sketch possible directions for tool-automation to support the learning processes associated with DSL adoption and to permit faster evolution cycles of the DSLs.

Keywords: Domain Specific Languages, DSL Learning, AST, Graph similarity, Language Oriented Programming

1 Motivation

Domain Specific Languages (DSL) are a very efficient mean to express business domain level functionality: the more specific they are, the more tailored and therefore efficient they will be to develop narrow aspects of the business domain.

The language oriented programming paradigm [Dmi04] is made today possible by several technical advances. Firstly, using Language Workbenches[1] like Xtext, MetaEdit+ or Jetbrains MPS, building an entire DSL with its environment, i.e., syntax, code generator or a model interpreter, and advanced editors support can now be done in a matter of days. Secondly, in case of language workbenches with advanced support for language modularization, it is easy to realize and adopt new extensions for existing languages. Developers can program their system in a host language and use domain specific language extensions whenever they are needed – e.g. in the mbeddr project, the host language is C implemented in JetBrains MPS, different domain specific extensions were built to support variability, requirements [VRT13] or analyses [RSVK12]. Thirdly, DSLs can evolve as the domain is better understood or new requirements arise [HRW10], therefore new constructs have to be learned, and new idioms emerge as the codebase of the DSL grows.

As consequence of the reduced cost of building, evolving and adapting DSLs, company-specific or even project-specific DSLs are now being developed in an iterative and more agile manner. *As entire DSL or single extensions are rapidly implemented, evolved and adapted, the*

---

[1] See http://www.languageworkbenches.net/

ability of practitioners to learn them becomes the bottleneck. The learning process involve both DSL users (developers writing code with the DSL) and DSL designers (developers designing and implementing the DSL). Recent work shows that often not all of the constructs of a DSL are used [TC13]. One of the reasons of this phenomenon is that users are not aware of their existence, hence they do not know completely the language. Another reason could be that DSL users consider some constructs not useful, hence the DSL designer did not address their real necessities.

Modern Language Workbenches emerged only recently, therefore DSL design is a practice not yet established. To help DSL design to be attested as a mature practice appropriate guidelines and metrics are needed. We suggest they could be derived from the analysis of the learning process.

The learning curve sums up with an already existing problem: the reluctance of practitioners on investing in learning new DSLs; or the frustration of having to do with different new constructs, or even having them used in a wrong manner. *Adequate support to reduce the investment required from developers to learn new DSLs is essential for the proliferation of DSLs.*

This paper aims to foresee a possible approach to support the DSL learning process. We propose to use code recommenders to support DSL learning and to provide feedback to DSL designers.

2 Proposal

Our goal is to collect the knowledge and wisdom about the DSL usage from available sources, in order to provide useful insights, to different actors, as they need them.

First, we want to guide beginner users to write higher quality DSL statements. It could be done through auto-completion and suggestions of complete examples:

- **auto-completion** would be based not only on syntax rules but also on semantic constraints. Semantic constraints are validation rules expressed in the language definition. Among the allowed constructs, we would suggest the most typical patterns of usage.

- **complete examples** and associated documentation would be shown when relevant to the current context.

Second, we plan to provide feedback to the DSL designers about the real **usage** of the DSL. As part of the feedback, statistical information about the most used and unused constructs could help the designer to identify parts of the DSL which are considered useless or confusing. Additionally, more elaborate processing of the usage information could lead to identify **emerging patterns**, which can be the hint indicating a missing abstraction for which explicit support could be added to the language.

As shown in Figure 1, we plan to reuse the grammar and the constraints defined by the language designer while realizing the DSL and the related tooling. We would also reuse examples provided for documentation purposes. Later, the users themselves could contribute by simply writing DSL code: repetitive, consolidate patterns would be extracted and used together with examples to guide users. Through the analysis of the actual usages feedback would be provided to the DSL designer who could iteratively evolve the language. New evolutions of the language
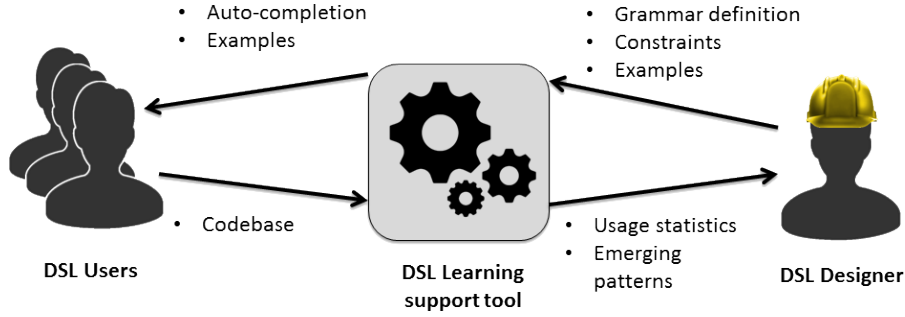
Figure 1: Schema of the DSL learning cycle

would be suggested to DSL users, who could keep in pace with the DSL evolution without consulting continuously the documentation.

3   Foreseeable design

To provide **autocompletion** we would transform the code being currently written by the DSL user, derive the corresponding AST and build possible extensions of that AST on the base of examples and common usages. Serializing those extended AST we would build a list of terms to be proposed as auto-completion items. This process is represented in Figure 2. The extended AST would have to satisfy the syntactic and semantic rules specified by the language.

To identify the appropriate **examples** we would compare the AST of the code being currently written with the AST of the examples, with graph similarities techniques.
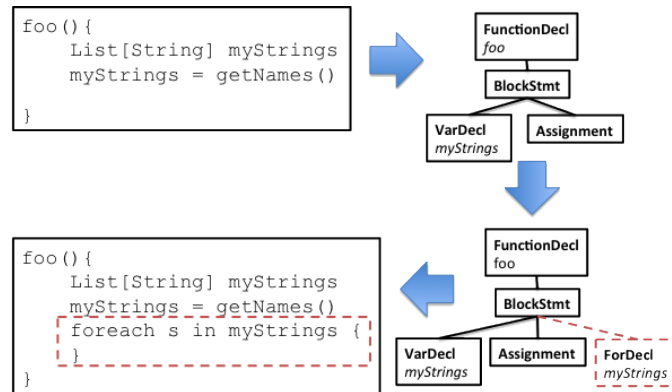

Figure 2: How we could implement autocompletion

Usage statistics could be easily calculated from the ASTs derived from the codebase written by DSL users.

To identify **emerging patterns** we should look for most frequent structures which appear in the codebase and containing redundancy.

9

4 Discussion

Through this automatic tooling we plan to decrease the cost of learning DSLs by supporting users and help them to achieve their immediate goals, reducing their reluctance in learning the language. Moreover DSL designers could benefit from the experience of the whole userbase, in a more systematic way, rather than relying on anecdotal indications, at best.

We think DSL design and deployment is a collaborative effort which involve both users and designers. The learning process, which goes through a number of iterations, need to be supported as a whole: the definitive goal is to base DSL design on facts and to help DSL users in learning how to learn languages.

Future work should be done in different directions: first we should study existing literature to learn from the experience matured on API recommenders and the general lesson learned on language learning. As second point we need to study technological solutions, reusing existing graph similarities algorithms.

There are still many open questions about future steps. We currently do not know how we could validate our approach: how we could measure the effort spent on learning and the reduction hopefully granted by our approach? How we could measure the understanding of a whole language by its users? Should we ask the opinions of users, measure their productivity, calculate how much time is required to each extension to be absorbed? Which are the measures most useful to language designers? How exactly should we identify emerging patterns?

Bibliography

[Dmi04]   S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard* 1(2), 2004.

[HRW10]   M. Herrmannsdoerfer, D. Ratiu, G. Wachsmuth. Language Evolution in Practice: The History of GMF. In Brand et al. (eds.), *Software Language Engineering*. LNCS 5969, pp. 3–22. Springer, 2010.

[RSVK12]   D. Ratiu, B. Schaetz, M. Voelter, B. Kolb. Language engineering as an enabler for incrementally defined formal analyses. In *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*. Pp. 9–15. 2012.

[TC13]   R. Tairas, J. Cabot. Corpus-based analysis of domain-specific languages. *Software and Systems Modeling*, pp. 1–16, 2013.

[VRT13]   M. Voelter, D. Ratiu, F. Tomassetti. Requirements as First-Class Citizens: Integrating Requirements closely with Implementation Artifacts. In *6th Int. Workshop on Model Based Architecting and Construction of Embedded Systems*. 2013.

Managing Language Variability in Source-to-Source Compilers by Transforming Illusionary Syntax

Tero Hasu[*]

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
tero@ii.uib.no

Abstract:

A programming language source-to-source compiler with human-readable output likely operates on a somewhat source and target language specific program object model. A lot of implementation investment may be tied to transformation code written against the specific model. Yet, both the source and target languages typically evolve over time, and compilers may additionally support user-specified, domain-specific language customization. Language workbenches commonly support generating object model implementations based on grammar or data type definitions, and programming of traversals in generic ways. Could more be done to declaratively specify abstractions for insulating the more language-semantic transformations against changes to source and target language syntax? For example, the idea of views enables pattern matching and abstract data types to coexist—could similar abstractions be made pervasive in a generated program object model?

Keywords: Language adaptation, program representation, Racket, transcompilers

1 Introduction

A programming language implemented as a compiler generating source code allows for reuse of existing infrastructure for the target language. Such a language can also enable abstraction over target language versions, implementations, and idioms (such cross-cutting concerns can be particularly pressing in a cross-platform setting). If the source-code generating compiler furthermore produces human-readable, high-abstraction-level output, then it also has a low adoption barrier in the sense that it can be regarded merely as tools assistance for programming in the target language. We use the term *source-to-source compiler* (or *transcompiler* for short) for such language implementations.[1]

A transcompiler typically translates its source language into its target language through successive program transformation steps. Each transformation step is programmed against a *program object model* (POM), which includes at least a data structure used to represent a program,

---

[*] This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

[1] An established definition for the term "source-to-source compiler" encompasses any compiler that produces its output *in* a high-level language, even when the output itself is low-level enough to read like assembly. For lack of a dedicated term for our more narrow definition, we simply use "source-to-source compiler" in the more narrow sense.

and a programming *interface* (or API) for manipulating the data. The POM (or POMs) used should be able to represent both source and target language programs, and any in-between language. It is common to define an intermediate language (or a *core language*) with a simpler, somewhat language agnostic syntax, which is something in between "desugared" source language and "ensugared" target language.

A compiler codebase mostly transforming core language may have fewer conditional cases (due to having fewer language constructs to manipulate) and less dependency on the *subject language* (i.e., source language, target language, or both [Kal07]). Still, as a source-to-source compiler's output should retain a high level of abstraction, sufficient semantic information must be carried through the compilation pipeline to allow high-level constructs to be preserved or recreated. This requirement for a "wider communication path" between the compiler front and back ends makes it hard to avoid language specificity. Having a large part of a compiler's codebase tied to language specifics may make it costly to maintain as languages evolve.

Languages may also change not due to design changes in the course of their evolution, but due to being designed to be extensible or otherwise adaptable to domain-specific purposes. It may be desirable to extend a transcompiler's source language with relevant abstractions for programming application or platform specific components. Likewise, it may be desirable to customize the output for different program configurations, perhaps to pick a supported or customary error handling mechanism [Has12], for instance. Where the extension mechanism is merely capable of mapping extended source language to unextended source language (e.g., traditional Lisp macro systems), an extension has no impact on the compiler. However, there also are extension mechanisms powerful enough to enable domain-specific optimizations [RSL+11, TSC+11], for example, and indeed there might be several extension points within a compilation pipeline [Bag10].

While there are promising approaches (e.g., based on attribute grammars [SKV13] and modern object-oriented language features [HORM08]) to enable modular language extension, it is as yet unclear just what kind of language adaptation is possible without having to manually adjust the compiler codebase. For now, to help achieve some degree of insulation against language variability in transcompiler engineering, we advocate working on tools support that encourages the use of abstraction (over specific language constructs) in programming code transformations. Although there are solutions for adapting existing POMs for existing interfaces (e.g., Kalleberg's POM adapter technique [Kal07]) and extending existing transformations to meet new requirements (e.g., through *aspects* [Kal07]), there is more work to be done in the area of providing a richer variety of APIs to program transformations against in the first place.

Structural abstraction is already widely supported in the form of generic traversals of some kind (e.g., term rewriting strategies, visitor design pattern). Stratego, for example, provides primitive traversal operators (**one**, **some**, and **all** generic functions) for all abstract syntax tree node types, as well as combinators for specifying more complex traversals. We are now looking for better support for language-semantic abstraction. This might entail allowing the programmer to declaratively express commonalities and relationships between syntactic constructs, and have the language development toolkit then readily offer support for writing transformations that avoid needless syntax specifics (where only a more general characteristic is of interest).

There are a number of existing tools (e.g., GOM [Rei07] and ApiGen) capable of generating a program model from a language grammar description (or similar), but the generated API invariably just follows the structure of the grammar. Thus, while it may be possible to declare some

```
(struct DeclVar (id t)) ;; variable binding (declaration)
(struct Var (id)) ;; variable reference (value expression)
(struct NameT (id)) ;; type name reference (type expression)
```

Figure 1: Grammatically unrelated syntax objects types (defined as Racket structures) with a commonality: each of them contains an identifier. An interface capturing the commonality might include a predicate, an id symbol accessor and mutator, etc. A global renaming implemented in terms of such an abstraction should be fairly decoupled from the underlying grammar.

```
(struct DeclVar (id t))        (struct DefVar (id t v))        (struct DeclVar (id t))
(struct DefVar (id t v))       (struct Undefined ())           (struct DefVar DeclVar (v))
```

Figure 2: Alternative declarations of syntax object types for the same abstract syntax: (1) unrelated types; (2) DeclVar is DefVar with Undefined initializer expression; and (3) with subtyping, variable definition being a special case of variable declaration.

commonalities between constructs deriving from the same non-terminal, other commonalities (such as show in Figure 1) typically require ad hoc, handwritten code to capture. We might want to augment a POM generator to implement interfaces that are independent of both subject language grammar and POM data representation. Such interfaces might span multiple (or no) object types, only provide access to certain portions of available information (possibly in a variety of formats), and yet appear similar to actual syntax objects' interfaces. It is unclear as to what kind of commonalities could conveniently be declared for purposes of code generation, and how.

An abstraction-friendly POM generator might benefit a compiler engineer by: making it less effort to implement multiple interfaces to choose from on a case-by-case basis (i.e., whichever seems most convenient for a given transformation); limiting breakage of abstraction-using transformation code upon subject language modification; and providing some insulation against incidental, implementation-specific grammar or program representation choices (such as shown in Figure 2). The possibility of effortlessly exposing various interfaces should make picking the "best" concrete choice less crucial in the first place. Even the distinction between object fields and *annotations* (i.e., open-ended collections of secondary, "non-structural" information in syntax objects) should become less prominent.

Pattern matching is commonly used in program transformations. Programming against abstract interfaces need not always mean the loss of pattern matching. The Tom system, for example, abstracts over concrete data structures by allowing rewriting based on algebraic terms that map to actual data structures [Rei07]. For further abstraction one might—following the philosophy of *views* [Wad87]—(seemingly) expose as many "data representations" per program object as desired. Some languages have sufficient "hooks" to enable "abstract algebraic views" to be implemented for purposes of pattern matching (e.g., as demonstrated in Figure 3).

A drawback of the kind of abstraction we have sketched is that it precludes the use of concrete syntax in patterns and templates, as supported by some language workbenches (e.g., Rascal and Spoofax). The illusion of semantic commonality capturing interfaces being like those of actual syntax objects can also be incomplete in respect to other abstraction mechanisms. Suppose

```
(define−match−expander DeVar
  (syntax−rules ()
    [(_ id t) (or (DeclVar id t) (DefVar id t _))]))
```

```
(match (DefVar 'x 'T (Literal 5))
  [(DeVar id t) (list id t)])
;;=> (x T)
```

Figure 3: Defining and using a custom pattern matching form DeVar in Racket, for matching both DeclVar and DefVar objects and their relevant fields. This implementation of DeVar works for representations (1) and (3) in Figure 2.

a particular view includes a specific annotation (thus making it look structural), but a generic traversal does *not* traverse any annotations; now there is a discrepancy, but it is not clear how the behavior of traversals should be affected by having multiple interfaces per object that may be used to query for traversable sub-objects.

Bibliography

[Bag10]   A. H. Bagge. Yet Another Language Extension Scheme. In Brand et al. (eds.), *SLE '09: Proceedings of the Second International Conference on Software Language Engineering*. LNCS 5969, pp. 123–132. Springer-Verlag, Mar. 2010.

[Has12]   T. Hasu. Concrete Error Handling Mechanisms Should Be Configurable. In *Proceedings of the 5th International Workshop on Exception Handling (WEH'12)*. Pp. 46–48. IEEE, June 2012.

[HORM08] C. Hofer, K. Ostermann, T. Rendel, A. Moors. Polymorphic Embedding of DSLs. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08, pp. 137–148. 2008.

[Kal07]   K. T. Kalleberg. *Abstractions for Language-Independent Program Transformations*. PhD thesis, University of Bergen, Norway, Postboks 7800, 5020 Bergen, Norway, June 2007. ISBN 978-82-308-0441-4.

[Rei07]   A. Reilles. Canonical Abstract Syntax Trees. *Electron. Notes Theor. Comput. Sci.* 176(4):165–179, July 2007.

[RSL+11]   T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, K. Olukotun. Building-Blocks for Performance Oriented DSLs. *ArXiv e-prints*, Sept. 2011.

[SKV13]   A. M. Sloane, L. C. L. Kats, E. Visser. A Pure Embedding of Attribute Grammars. *Sci. Comput. Program.* 78(10):1752–1769, Oct. 2013.

[TSC+11]   S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, M. Felleisen. Languages as libraries. *SIGPLAN Not.* 47(6):132–141, June 2011.

[Wad87]   P. Wadler. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '87, pp. 307–313. 1987.

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 inter-disciplinary 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

Industrial keynote: "I double-dog dare you..."

Ira Baxter

Abstract. Tools for processing code promise the ability to carry complex analyses and code changes not possible with manual approaches. SD's commercial tool, DMS, attempts to provide adequate support for building sophicated custom tools. This talk will sketch a few examples of what DMS has been used for successfully. It will then discuss a variety of issues encountered or expected which prevent more effective use. As CTO, my job is to find working technology gems to integrate into DMS to make it ever more capable. As CEO, my job for a commercial company is to avoid doing anything resembling research. (I fail at this more often than I'd like). As a company, we are dependent on the research community to figure out how to solve the hard problems. A double-dog dare is a bet made that the darer will do something crazy if the daree does it first. So, this is a set of challenges for the research community to do first.

Towards Language-Oriented Software Development

Changyun Huang[1], Naoyasu Ubayashi[2], Yasutaka Kamei[3]

[1] huang@posl.ait.kyushu-u.ac.jp, [2] ubayashi@acm.org, [3] kamei@ait.kyushu-u.ac.jp
Kyushu University, Japan

Abstract: LOP (Language-Oriented Programming) is a programming paradigum in which a programmer constructs one or more DSLs and develops an application using these DSLs. LOP opens the door towards a modern modularity vision. However, it is not clear how to integrate domain analysis and DSL implementaion. To deal with this problem, we propose *DSL-Line Engineering* (DLE), an automated DSL construction method. This paper introduces DLE and raises a discussion towards LOSD (Language-Oriented Software Development).

Keywords: DSL, Language-Oriented Programming, Software Product Line

1 Introduction

Abstraction is crucial in software engineering. DSL (Domain-Specific Language) is one of the promising approaches to dealing with software abstraction. An application can be developed at a high abstraction level by using a DSL that encapsulates the details of domain knowledge and hides the usage of a specific software platform. Recently, extensible programming languages such as Scala have become popular and programmers can easily develop DSLs for their own purpose. LOP (Language-Oriented Programming) [War] is a programming paradigm in which a programmer constructs one or more DSLs and develops an application using these DSLs. That is, application development process is integrated with DSL construction. From the viewpoint of software modularity, it is helpful for a programmer to be able to define each DSL feature as a software module for expressing domain knowledge and create an application by configuring these DSL features. Although LOP opens the door towards a modern modularity vision, it is not clear how to integrate domain analysis and DSL implementation.

To deal with this problem, we propose *DSL-Line Engineering* (DLE), an automated DSL construction method based on SPL (Software Product Line). In our approach, a feature in a product-line analysis model can be regarded as a component for generating a variety of DSLs. The contribution of this paper is a proposal of a design and implementation method for realizing the concept of LOP. This paper introduces DLE and raises a discussion towards LOSD (Language-Oriented Software Development) whose philosophy is *"Software development is language development."*

2 DSL-Line Engineering

DLE consists of the following steps: 1) a feature model extracting DSL features is created using FODA (Feature-Oriented Domain Analysis) [Kan02] and is encoded into logical formula; 2) the syntax of a DSL is generated from the feature model by using a SAT solver as a model finder; and 3) a DSL is automatically implemented using an extensible language. Currently, Alloy
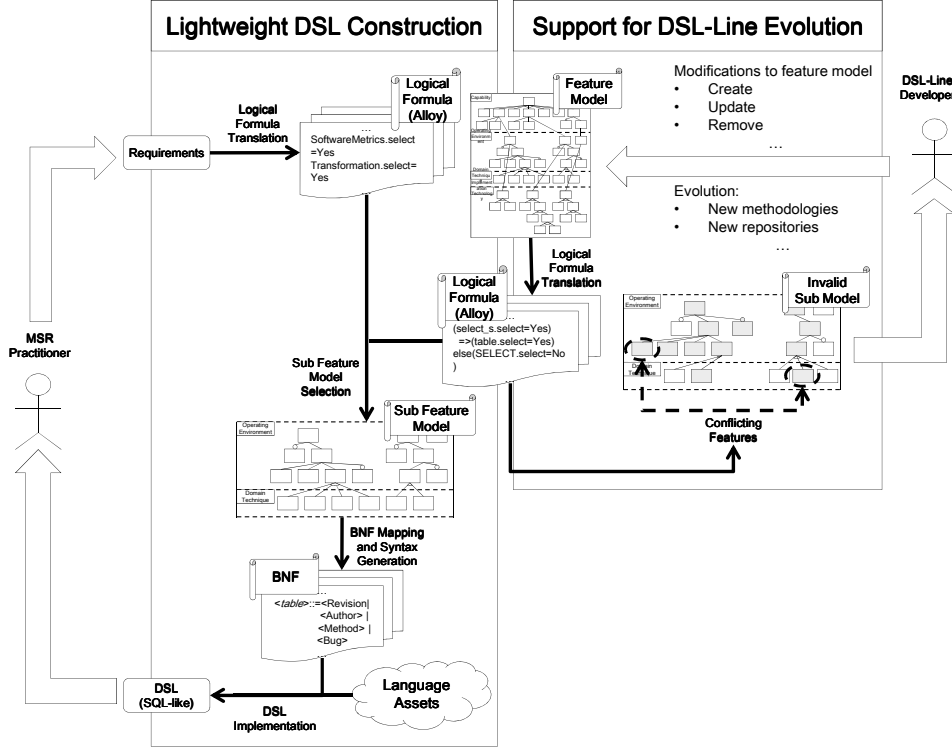
Figure 1: DSL-Line Engineering

[Jac06], a formal specification language based on first-order relational logic, is used in DLE. Alloy Analyser can find a model satisfying specified constraints by using an external SAT solver. In step 2, a BNF (Backus-Naur Form) definition is generated as a sub-feature model satisfying DSL requirements. Figure 1 shows the overview of DLE.

As a case study, we applied DLE to develop ArgyleJ [HYK+13], a Java-based DSL for supporting MSR (Mining Software Repositories) that analyzes big data stored in OSS (Open Source Software) repositories, bug reports, and mailing lists. High performance computing is needed in MSR, because the size of these repositories is huge. Our DSL adopted SQL-like syntax that enables MSR practitioners to mine software repositories as if they retrieve information from relational databases. The platform usage of high performance computing such as GPGPU and Hadoop is encapsulated in a SQL-like DSL. We used ProteaJ [IC11] as a Java-based extensible language. ProteaJ provides a mechanism to define new n-ary operators. The SQL-like statement in ArgyleJ is introduced using this mechanism. Using ArgyleJ, a MSR practitioner can easily develop a Java application that calculates a variety of software metrics such as code complexity and probability of defects without taking into account the existence of GPGPU and Hadoop.

3 DLE as LOP Mechanism

From the viewpoint of LOP, DSL layers can be obtained using DLE. In case of ArgyleJ, ProteaJ is located at the base layer. Each language feature specific to MSR is implemented as an internal DSL of ProteaJ. ArgyleJ is implemented as the following DSL layers: Layer 0) ProteaJ; Layer

1) Basic SQL statement for MSR (e.g. `SELECT` *metrics* FROM *table*); Layer 2) Specification of metrics granularity such as package-level, file-level, and method-level (e.g. `SELECT` *metrics* FROM *table* EACH {*PACKAGE | FILE | METHOD*}); Layer 3) Specification of analysis period using *between*, *before*, and *after* (e.g. `SELECT` *metrics* FROM *table* BETWEEN *date1* and *date2*); and Layer 4) Application-specific DSL constructs. We can define other layers when we want to add new DSL features. These features correspond to the functions required in MSR applications. We can consider that DSL layers correspond to the modular structure of application framework that provides a set of common functions. In other words, application development in LOP can be regarded as designing and implementing DSL layers.

Using DLE, we can develop and maintain an application by adding new DSL layers or re-configuring the existing DSL layers. DLE provides a yet another modularity vision to software development. Although LOP is a modern programming paradigm, systematic methodologies for designing and implementing an application based on LOP have not been proposed. DLE can support not only LOP but also other development phases including analysis and design. In this paper, a software development style inspired by LOP is called LOSD. DLE automates the process of LOSD using SPL practices such as FODA and formal methods.

4   Research Challenges towards LOSD

In this section, we raise several research challenges towards LOSD and provide the direction of possible solutions. The integrated usage of FODA and formal method is applied in many SPL projects and the approach of DLE can be considered reasonable in terms of SPL. However, we have to deal with difficult problems when we apply SPL to DSL construction. The problems shown in this section were recognized when we developed ArgyleJ.

4.1   DSL-aware Feature Modeling

It is not clear what kind of feature model we have to create to generate an input to an extensible language. Moreover, it is not clear how to specify DSL requirements. A feature model in FODA consists of four layers: *Capability*, *Operating Environment*, *Domain Technology*, and *Implementation Technology*. In ArgyleJ, we assigned MSR tasks, ArgyleJ DSL constructs, ArgyleJ data types, and implementation libraries for GPGPU and Hadoop to the capability layer, the operating environment layer, the domain technology layer, and the implementation technology layer, respectively. DSL requirements were specified by selecting the target MSR tasks supported in ArgyleJ (capability layer) and the execution environment of ArgyleJ (implementation technology layer). Using Alloy, DSL constructs and data types in ArgyleJ are automatically selected from the DSL requirements. Alloy as a model finder can generate the DSL syntax composing of operations and data types that satisfy the DSL requirements.

Our approach taken in ArgyleJ is not complete, because the granularity of DSL features remains unclear. We assigned a DSL feature corresponding to each operation such as `SELECT`, `FROM`, and `EACH`. We adopted this approach because of automated syntax generation. However, the granularity is so small in terms of SPL practices and it is difficult to manage DSL features. We have to resolve this problem. One possible solution is to define a DSL feature as a syntax template. A DSL is obtained by composing related templates. Even in this solution, we have to resolve the conflicts among DSL templates. For example, the `BETWEEN` template (`BETWEEN`

date1 `AND` *date2*) cannot be composed with the `AFTER` template (`AFTER` *date*). However, the `BEFORE` template (`BEFORE` *date*) can be composed with the `AFTER` template. In this case, the `AND` operator or the `OR` operator is needed (`BEFORE` *date1* `AND AFTER` *date2*). We have to analyse what kinds of constraints are needed between DSL templates. This problem is not easy.

DSL-Line evolution is a crucial problem to guarantee the compatibility between DSL versions. We have to check whether or not a modification of a feature model is consistent with the existing model. It is unclear what kinds of constraints should be preserved between DSL versions.

4.2 Mapping to Extensible Languages

This problem is crucial in terms of automated DSL construction. In traditional SPL, a software product can be configured by binding a set of software components. The relation between these components can be represented by *caller-callee*. However, the configuration of DSL features cannot be dealt with *caller-callee* relations. In DLE, the configuration is performed by translating a feature model into a BNF description. The functionality of extensible languages is also a main concern in order to bridge a BNF description and DSL implementation. BNF-based external DSL construction frameworks such as Xtext is suitable to DLE. However, extensible languages based on reflection is not suitable to internal DSL construction, because a BNF description cannot be directly mapped to a reflective description. We need an extensible language that provides a BNF-like language extension mechanism. ProteaJ can add a new DSL statement to Java by defining a n-ary operator. Although ProteaJ-like languages open the door towards automated language construction, we have to integrate a BNF description with its implementation whose components are defined in the implementation technology layer. This is an open issue.

5 Conclusion

We proposed the concept of DLE. Although we have to resolve the issues figured out in this paper, we consider that DLE opens the door towards LOSD.

Bibliography

[HYK+13] C. Huang, K. Yamashita, Y. Kamei, K. Hisazumi, N. Ubayashi. Domain Analysis for Mining Software Repositories -Towards Feature-based DSL Construction-. In *4th International Workshop on Product LinE Approaches in Software Engineering (PLEASE 2013) (Workshop at ICSE 2013)*. Pp. 41–44. 2013.

[IC11] K. Ichikawa, S. Chiba. Constructing Internal Dsls in a Statically Typed Language by User-defined Operators. In *25th European Conference on Object-Oriented Programming (ECOOP 2011), poster*. 2011.

[Jac06] D. Jackson. *Software Abstractions*. The MIT Press, 2006.

[Kan02] K. C. Kang. Feature-Oriented Product Line Engineering. *IEEE Software* 19(4):58–65, 2002.

[War] M. Ward. Language Oriented Programming. http://www.cse.dmu.ac.uk/~mward

Capturing Programmer Intent with Extensible Annotation Systems

Mark Hills

Abstract. Many programs make implicit assumptions about data, often captured in comments and variable naming conventions. Common examples include whether a variable has been initialized at a certain program point, whether a reference or pointer is (or must always be) non-null, and whether a program value can escape from the current context. Domain-specific examples are also common, with many scientific programs manipulating values with implicit units of measurement. Most widely-used languages, including C, provide no language facility for representing these assumptions, making violations of these implicit program policies challenging to detect.

To solve this problem, policy frameworks were created to provide a programmer-friendly way to make these assumptions explicit, using function contracts and statement annotations to capture programmer intent. Policy frameworks were also designed to improve reuse in program analysis tools, allowing extensible annotation systems to be defined and used in conjunction with an analysis-specific language semantics. So far, this technique has been used to define multiple policies for the C and SILF languages, including a units of measurement analysis policy for C that is competitive with existing tools.

Current work on policy frameworks is following two tracks. First, since defining a policy can require a significant level of knowledge about the internals of the semantic definition, we are looking at using both domain-specific languages and reflective semantic definitions to guide the policy creation process. Second, we are looking for opportunities to apply these techniques to new languages and new problem domains, including the use of units for empirical software engineering.

Orthogonal and Extensible Type Systems:
The Birth of Domain Specific Type Systems?

Merijn Verstraaten

m.e.verstraaten@uva.nl
Instituut voor Informatica
Universiteit van Amsterdam

Abstract: Type systems are commonly used to help ensure program correctness by automatically verifying invariants. Invariants can usually be divided into several orthogonal classes and encoding these different classes into a single type system leads to needlessly complex types. In this article, I propose splitting the encoding of these orthogonal invariant classes into separate type systems. This lets programmers worry about one class of invariants at a time. Additionally, I observe that if we are designing a programming language to have multiple type systems, we may as well design it to be extensible. This would allow programmers to specify additional classes of invariants and typing rules for these, opening these invariant classes up to automatic verification by the compiler. I outline these opportunities and suggest a followup research agenda.

Keywords: type systems, formal verification, extensibility

1 Introduction

While it is notoriously hard to get programmers and researchers to agree on anything, most non-contrarians will at least agree that catching software problems as early as possible is a Good Thing[TM]. Even so, the holy war between static and dynamic typing continues undiminished. Past attempts at a cease-fire[MD04] have failed, and the rest of this paper will unapologetically argue that more static typing is better.

It is common folk wisdom[1] that using invariants is one of the most tractable ways to reason about program correctness. Invariants are so pervasive they are even catered to by most tools, such as documentation generators (e.g. javadoc). Type systems provide a way to encode invariants in the program text and have the compiler check them. Simple type systems only deal with trivial invariants, like function arity and their types. More expressive type systems allow more complex invariants to be encoded; such as whether a function uses I/O, takes optional arguments (Haskell's `Maybe` and ML's `option`), or whether an operation can fail (Haskell's `Either`).

Many invariants that cannot be checked by the compiler are written down in documentation or comments, their verification is left to the programmer. The problem with relying on programmers, or alternatively the advantage of machine checking invariants, is that programmers are only human[2]. They are prone to laziness and forgetfulness; if the programmer forgets to check

---

[1] I.e., I couldn't find any empirical evidence for this, but I am going to blindly assume it anyway.

[2] This might be an unnecessarily specie-ist assumption.

whether a function indeed upholds the invariants he/she is assuming, the value of reasoning using said invariants is immediately lost. By having the compiler verify invariants they cannot be forgotten. Additionally, some programmers may not bother to fix their programs unless they encounter a bug in production *or* the compiler refuses to produce an executable.

A significant source of interest in dependently types languages comes from the ability to encode more complex invariants in their type systems; such as execution cost [BH06] or resource initialisation and lifetime [FB13]. Even in languages that do not support dependent types the type system is frequently stretched[3] in attempts to encode operational invariants in them [McB].

In the Haskell community it is not uncommon for people to perform elaborate type level acrobatics so that the compiler can verify operational invariants. However, as the complexity and number of invariants increases, so does the clutter forced upon the poor type system. This results in a trade-off between the readability of the types and the possibility to have the compiler automatically verify your invariants.

The realisation that invariants are a significant help is not restricted to functional languages like Haskell and Idriss, they apply equally well to languages like Java. A pluggable type system for Java is described and evaluated in [DDE[+]11, PAC[+]08]. They let programmers specify their own typing invariants, these can then be annotated in type signatures and machine be machine checked. Testing showed that these type systems were not particularly difficult for novices to use and case studies showed a significant number of bugs could be identified this way.

In the above examples we covered some examples of trying to encode invariants with the limits of an existing type system and retrofitting existing type systems with additional invariants. However, why are we even trying to find a one-size-fits-all (or, more accurately, one-size-does-not-quite-fit-anyone) solution? In this position piece I argue that we should consider implementing multiple orthogonal type systems in an extensible way, thus paving the way for the birth of domain-specific type systems.

2   Applications of Domain-Specific Type Systems

I had the privilege this year to work with peers at Heriot-Watt University who are working on the language Single Assignment C (SAC) [GS]. SAC is a purely functional language focussed on high level, high performance array programming. One issue that they run into with their optimising compiler is that many performance optimisations (e.g., vectorisation, optimal use of GPU memory hierarchy) are highly dependent on the memory layout of the arrays.

In other words, they have multiple representations of data that are not visible at the language level, but are important during compilation. This problem is, of course, not restricted to matrices. This problem can be generalised to other data types too, consider graphs. The optimal representation of a graph depends on its size, density and use. In these cases the compiler needs to be aware that two optimised functions may not be able to be trivially combined, as it is possible that they use differing layouts.

In SAC a separate type system was introduced that tracks the layout of data. Not all code is layout sensitive, for example a function that only passes its array arguments on may be layout polymorphic. Such polymorphic code can work with data of any layout, but code with a fixed

---

[3] Past their breaking point, some would argue.

layout can only be combined with data of the right layout, unless an explicit layout transformation is done.

This layout type system is completely orthogonal to the traditional value type system that is used and the typing is not part of the value types. As such, their work is effectively a proof-of-concept illustrating a use case for multiple orthogonal type systems and showing that they are feasible to implement.

In a different direction, with the rise of GPU computing and many core chips, we are increasingly living in a NUMA world. Above we discussed how a layout type system can helps us enforce memory layout invariants. In a concurrent world there are memory issues beyond layout.

Synchronisation between multiple execution units is an expensive operation. Even so, many garbage collectors still rely on a stop-the-world phase, forcing all executing threads to synchronise. The reason for this is the reliance on a single shared heap for all memory allocation, as opposed to a per-core heap. Introducing per-thread heaps is a simple task, but this means data migration between cores can no longer happen transparently, as this now requires a garbage collector hand-off. Additionally, one wants costly operations like this to be visible to the programmer, to allow reasoning about execution costs.

In effect, we want to have multiple garbage collection *domains* and have values be tied to a domain. For example, one domain per core. One important invariant is that cores do not hold references to values that are outside their domain. Exactly the sort of invariant we would like our compiler to verify for us.

We can associate a domain type with every value, indicating the garbage collection domain where it was allocated. Most code should have a polymorphic domain, but we would like threading primitives (like Haskell's forkIO) and communication primitives (e.g. queues, channels) to have an explicit domain type. This notion is very similar to the to the phantom state parameter used in Haskell's lazy functional state threads [LP94] and serves much the same purpose. Mixing domain types would be a type error and conversion between domain types would only be allowed using built-in communication primitives that take care of the garbage collection hand-off.

Yet another potential use case is the age old functional programming fight whether lazy-by-default or strict-by-default is the right choice. We can have a separate typing layer that indicates whether code is lazy, strict or evaluation polymorphic (i.e. can be used both strictly and lazily). This can be used to indicate mark infinite data structures as lazy, thus avoiding the application of a function expecting a strict data structure and ending in an infinite loop.

This is just a small selection of program properties we would like the compiler to verify. Most of these are not entirely new (and some are entirely not new) suggestions. However, most past suggestions have mostly focussed on how to integrate these types into the value type system. The main objection against this is that it clutters and complicates the value type system. This holds especially for scenarios where we would like most of our code to be polymorphic in the additional typing axis.

That is, most code should be polymorphic in layout, evaluation strategy or domain. However, we would still like to expose primitives to programmers that are not polymorphic in these aspects. This would let the compiler verify that the use of these primitives does not violate important invariants, like mixing values from different domains.

3 Extensible Type Systems

Once we take the step of adding one or more orthogonal type systems in addition to a traditional value type system, the question arises: Why restrict ourselves to a fixed set of predefined type systems? A well established software engineering design principle is the Zero-One-Infinity Rule [Jar03]. This rule states: "Allow none of *foo*, allow one of *foo*, or any number of *foo*."

In other words, if we're going to design a programming language that caters to multiple orthogonal type systems, why not immediately generalise our design to be extensible. That is, allow programmers to implement their own additional type systems for encoding variants, similar to the work done in [DDE+11, PAC+08]. Especially since this work has already shown that allowing users to encode their own invariants in the type system is a viable approach for detecting bugs and errors.

4 Conclusion

In short, I hope that this short introduction provides an interesting angle for thinking about the interplay between type system and programming language design. There are many interesting properties of computation that we would like to have the compiler track, without trying to put all of them in a single type system. Some other examples not covered here are contract checking or approximate computing.

Additionally, I hope that the idea of making the type system extensible by programmers helps us reconsider and reexamine the interaction between programmer, compiler and language. As well as functioning as a trojan horse for selling dependent types to the masses.

Bibliography

[BH06]   E. Brady, K. Hammond. A dependently typed framework for static analysis of program execution costs. In *Implementation and Application of Functional Languages*. Pp. 74–90. Springer, 2006.

[DDE+11]   W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*. Pp. 681–690. Waikiki, Hawaii, USA, May 25–27, 2011.

[FB13]   S. Fowler, E. Brady. Practical Dependently-Typed Programming for the Web. In *Proceedings of 25th Symposium on Implementation and Application of Functional Languages (to appear)*. ACM, 2013.

[GS]   C. Grelck, S. bodo Scholz. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming*, p. 2006.

[Jar03]   The Jargon File (version 4.4.7). Online, 2003.
            http://www.catb.org/jargon/html/Z/Zero-One-Infinity-Rule.html

[LP94]   J. Launchbury, S. L. Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*. Volume 29(6), pp. 24–35. 1994.

[McB]   C. McBride. Functional pearl: Kleisli arrows of outrageous fortune. *Journal of Functional Programming (to appear)*.

[MD04]   E. Meijer, P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. 2004.

[PAC+08]   M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*. Pp. 201–212. Seattle, WA, USA, July 22–24, 2008.