# Dedicated Support for Analyses and Optimizations in Language Workbenches

Tamás Szabó

itemis, Germany / Delft University of Technology, Netherlands

tamas.szabo@itemis.de

## Abstract

Language workbenches are widely used to implement domain-specific languages (DSLs) and their accompanying integrated development environments (IDEs). They help to define the abstract syntax, concrete syntax(es), type system, and transformations for the languages. However, there are other language aspects, specifically program analyses and optimizations, that are also crucial to a language implementation, but state-of-the-art language workbenches has only limited support for them. The high implementation effort for these language aspects is justifiable for a general-purpose language (GPL), but is not justifiable for DSLs because of their different development economies.

To this end, I conduct research on dedicated support for analyses and optimizations for DSLs in language workbenches. My main goal is to develop declarative meta-languages that help to define static program analyses and that capture and automate patterns and techniques of optimizations. The research directions are directly driven by industrial need, and upon successful completion, the results would be applied in projects centered around DSLs for high-performance computing (HPC), insurance, and concurrent embedded systems.

***Categories and Subject Descriptors*** D.2.6 [*Software Engineering*]: Programming Environments; D.3.2 [*Programming Languages*]: Processors

***Keywords*** Static Analysis; Incremental Computation; Domain-specific Language; Language Workbench

## 1. Introduction

Language workbenches [3] are tools for efficiently implementing DSLs and their IDEs. They offer generic support for specifying the various aspects of a language implementation including, for example, abstract and concrete syntax, type systems, and transformations. This support is either in the form of APIs (e.g. Xtext[1] and Rascal[2] follow this approach), while other language workbenches (such as MPS[3], Spoofax[4]) provide meta-languages to express the language aspects.

Advances in language workbenches allowed to use these tools to build systems of considerable size, such as mbeddr [5] or WebDSL [4]. Compared to early systems, it is not enough anymore to define only a single DSL: sets of integrated DSLs are required. Additionally, in many domains, textual notation for DSLs is only one concrete syntax style among many. There are several areas, such as insurance, HPC, or circuit design, where multiple syntactic forms (math formulas, tables, diagrams, text) are required in an integrated fashion.

Projectional workbenches are particularly good at supporting language composition and notational flexibility [6]. They do not use a parser in the editing process but directly work on the abstract syntax tree (AST) of the program. User edit actions are directly mapped to tree transformations. If independently developed languages are composed, the resulting composite program will never be syntactically ambiguous because there is no parsing step. This is important when composing and extending multiple DSLs. Additionally, users interact with a projection of the AST, which allows the use of multiple notations even for different parts of the same program. This is a perfect enabler of implementing DSLs with non-textual notations.

## 2. Problem Statement

As discussed in the previous section, the advent of language workbenches has led to significant simplifications in the design and implementation of DSLs. However, there are other language aspects that are crucial to a language implementation that were traditionally integrated into the compiler of the language. Typically, these relate to non-local static analyses and optimizations. Examples include data-flow analyses

[1] http://www.eclipse.org/Xtext

[2] http://www.rascal-mpl.org/

[3] http://www.jetbrains.com/mps

[4] http://metaborg.org/en/latest

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

*SPLASH Companion'16*, October 30 – November 4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4437-1/16/10...
http://dx.doi.org/10.1145/2984043.2984050

for program understanding and target-architecture specific generators or optimizations to reduce the overhead involved in managing concurrency. The state-of-the-art in language workbenches currently lacks support for efficiently implementing these advanced language aspects.

The focus on the *efficient* implementation of these advanced aspects is motivated by an important economic difference between GPLs and DSLs. The former have a large user base and evolve slowly. The latter have a small(er) user base and evolve much quicker, based on the evolving requirements of the domain. Thus, a high implementation effort for sophisticated optimizations and fast analyses can be justified for GPLs, but is infeasible for DSLs. To resolve the tension between the increasing need for analyses and optimization and the low(er) available effort for their implementation, language workbenches must provide dedicated support (similar to the support for syntax, transformations, etc.) for efficiently implementing analyses and optimizations to allow them to evolve together with the subject DSL as it evolves iteratively.

**Research problem** My main research problem is to design and develop dedicated support for optimizations and analyses for DSLs in language workbenches, as shown in the innermost circle of Figure 1.

Based on the experience of the language engineering team at itemis (the company I work for), there is significant industry need for these language aspects and meta-languages in the domain of HPC, DSLs for insurance, and in the context of embedded systems.

## 3. Solution Approach

My approach relies on three pillars: subject-language extension, meta-languages and inherent incrementality.

**Subject-language Extension** A major problem with analyses is "understanding" the semantics of the subject language. The subject language is typically not designed for a particular class of analysis, which means that the semantics relevant for the analysis must be derived from the existing abstractions in the subject program. Using language engineering, the subject language can be extended with concepts that *directly encode the particular semantics relevant for a given analysis*. In the extreme case, this may enable the analysis in the first place if the abstractions are semantically rich enough. More often, however, it will make the implementation of the analysis algorithmically simpler. This is important, because, as discussed before, a high implementation effort for analyses is hard to justify for DSLs.

**Meta-languages** The second synergy between language engineering and analyses is that a meta-language can be used to express the analysis. This, in turn, has two advantages. First, the development of the analysis becomes more efficient, because the benefits of DSLs – better abstractions, better notations – are applied to the domain of analyses. Second, the

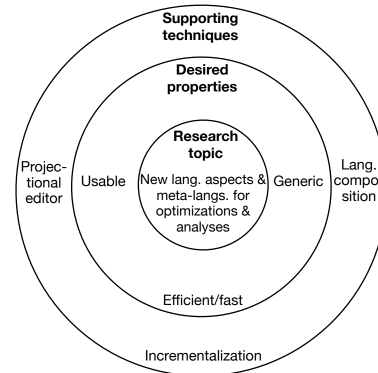


**Figure 1:** The figure positions the research topic. The innermost circle states the research problem. The middle circle lists the desired properties for the new language aspects. The outermost circle shows the techniques that enable the tackling of the research questions.

analysis programs themselves become analyzable, enabling further optimizations which would not be the case if some form of library and APIs would be used for program analyses.

**Incrementality** There are several ways to speed-up program analyses and compilation. For example, with the current hardware resources, parallelization is an immediate option. Additionally, approximations are widely used in program analyses, because often an approximate analysis result (which is cheaper to compute) is still enough for bug finding or code optimizations. However, incrementalization is a perfect aid in achieving considerable speed-ups because it perfectly aligns with how developers modify the subject program in a language workbench. In fact, a projectional workbench is a perfect environment for incremental program analyses because it can identify localized and fine-grained program updates without additional computation overhead. This would not be the case in a textual workbench without an incremental parser in the background.

**Concrete steps** The current state-of-the-art has many solutions for fast and precise program analyses, but these mostly target only a particular subject language and/or a particular class of analyses. As shown in the middle circle of Figure 1, the new language aspects and meta-languages will only be useful if they meet certain properties: they must be generically applicable to all DSLs, they must have good usability both in terms of necessary abstractions and desired notations, and their underlying runtime system must be efficient and fast. I plan to improve on the state-of-the-art with the following concrete steps using the supporting techniques shown in the outermost circle of Figure 1:

- I design DSLs for specifying program analyses that are independent of the subject language and support a wide range of analyses (data-flow, syntactic checks, enforcement of coding standards, etc.).

- I introduce support for variable-precision analyses which trade-off precision for runtime performance. This has the benefit that analyses can be tailored to a particular use case; fast and less precise analyses for real-time feedback, while potentially slower but more precise analyses in compilers. Additionally, the technique of incrementalization has the potential to considerably speed up even the more precise program analyses, thus making them feasible to also run in real-time. This is because instead of the repeated "from-scratch" computations, an incremental analysis updates its result as the program changes and does that by only considering the truly affected program parts.
- I investigate how to use DSLs to capture and automate patterns and techniques of code generation, transformation, and optimization that recur in an application domain. This area is widely regarded as a current prompt topic among programming language and HPC researchers [1].

## 4. Evaluation

To evaluate my approach, I will implement the designed meta-languages in the projectional MPS language workbench. My research work is directly driven by industrial requirements and by customer projects built on top of MPS. In particular, a concrete case study is mbeddr [5] which is a set of integrated languages for embedded software development plus an IDE built with MPS. The newly developed meta-languages and language aspects will be of immediate interest for fast and precise data-flow analyses as well as concurrency related optimizations for embedded software.

**Hypothesis** We (the language engineering team at itemis) expect that these languages will boost developer productivity for the development of analyses and optimizations. Consequently, we expect more, and more interesting analyses to be developed for DSLs as part of customer projects.

**Experiments** There are two aspects of the evaluation of the new meta-languages and language aspects. First, project experience and surveys among language engineers will show the usefulness of the meta-languages. Second, I will measure the efficiency and performance of the program analyses, compilers, and optimizations. To this end, I will use the code bases of varying sizes of existing customer projects built with mbeddr and MPS.

**Use cases** There are several areas where I will apply the research results. I highlight two of them:

mbeddr comes with a core C language implementation and several extension DSLs with concurrency abstraction for writing concurrent embedded software. I will use the analysis meta-language to implement data-flow analyses that drive concurrency-related optimizations, such as lock elision, removal of recursive locks, and narrowing of synchronization statements [2].

Another example is centered around HPC applications. Here, mbeddr comes with DSLs for matrix operations, operations on complex vectors, and stencil computations. I will use the research results to create code generators that can emit target-platform specific code from the same high-level DSL code.

## References


[1] B. Aktemur, Y. Kameyama, O. Kiselyov, and C.-c. Shan. Shonan Challenge for Generative Programming: Short Position Paper. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*, PEPM '13, pages 147–154, New York, NY, USA, 2013. ACM.

[2] J. Aldrich, C. Chambers, E. G. Sirer, and S. Eggers. *Static Analyses for Eliminating Unnecessary Synchronization from Java Programs*, pages 19–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[3] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, 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, and J. van der Woning. Evaluating and comparing language workbenches. *Comput. Lang. Syst. Struct.*, 44(PA):24–47, Dec. 2015.

[4] E. Visser. Generative and Transformational Techniques in Software Engineering II. chapter WebDSL: A Case Study in Domain-Specific Language Engineering, pages 291–373. Springer-Verlag, Berlin, Heidelberg, 2008.

[5] M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using C Language Extensions for Developing Embedded Software: A Case Study. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 655–674, New York, NY, USA, 2015. ACM.

[6] M. Voelter and S. Lisson. Supporting Diverse Notations in MPS' Projectional Editor. In *GEMOC@Models 2014, Valencia, - Spain, September 28, 2014.*, pages 7–16, 2014.