Separation of concerns in language definition

Visser, Eelco

DOI
10.1145/2584469.2584662

Publication date
2014

Document Version
Final published version

Published in
MODULARITY ’14 Proceedings of the companion publication of the 13th international conference on Modularity

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable). Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights. We will remove access to the work immediately and investigate your claim.
Separation of Concerns in Language Definition

Eelco Visser
Delft University of Technology, Delft, The Netherlands
visser@acm.org, http://eelcovisser.org

Abstract
Effectively applying linguistic abstraction to emerging domains of computation requires the ability to rapidly develop software languages. However, a software language is a complex software system in its own right and can take significant effort to design and implement. We are currently investigating a radical separation of concerns in language definition by designing high-level declarative meta-languages specialized to the various concerns of language definition that can be used as the single source of production quality (incremental) semantic operations and as a model for reasoning about language properties.

Categories and Subject Descriptors D.2.4 [Software/Program Verification]: Correctness proofs; D.2.6 [Programming Environments]: Interactive environments; D.3.1 [Formal Definitions and Theory]: Semantics; D.3.4 [Processors]: Translator writing systems and compiler generators

General Terms Languages, Design

Keywords language design; meta-languages; language workbench; Spoofax; SDF3; NaBL; Stratego

1. Introduction
Software systems are the engines of modern information society. Our ability to cope with the increasing complexity of software systems is limited by the programming languages we use to build them. Bridging the gap between domain concepts and the implementation of these concepts in a programming language is one of the core challenges of software engineering. Modern programming languages have considerably reduced this gap, but still require low-level programmatic encodings of domain concepts.

Domain-specific software languages (DSLs) address the complexity problem through linguistic abstraction by providing notation, analysis, verification, and optimization that are specialized to an application domain. The high-level abstractions of a DSL allow developers to directly express design intent (‘language shapes thought’), and allows a compiler to report errors using domain terminology.

Effectively applying linguistic abstraction to emerging domains of computation requires the ability to rapidly develop software languages. However, a software language is a complex software system in its own right and can take significant effort to design and implement.

2. Concerns in Languages Definition
A language implementation includes the following language components. A compiler translates a source program to executable code in a lower-level language. An interpreter or execution engine directly executes a program. An integrated development environment (IDE) supports developing (editing) programs by means of a range of editor services that check programs and help navigating large programs. A language specification that documents its features.

The production of each of these components incorporates several or all of the following language definition concerns. A syntax definition defines the structure of well-formed sentences in the language. Name binding and scope rules determine the relation between definitions and uses of names in programs. A type system defines static constraints on syntactically well-formed programs, avoiding a large class of run-time errors. The dynamic semantics defines the dynamic behaviour (execution) of programs. Transformations define modifications that improve programs in some dimension such as performance or understandability. A code generator defines the translation to code in another, typically lower-level, target language.

Over five decades, programming language developers and designers have produced many different methods for encoding the definition of language concerns. Some are specialized to the implementation of compilers and IDEs, others are specialized to reasoning about the correctness of language definitions. Language workbenches such as MPS [7], Xtext [3, 4], and our own Spoofax [8] provide abstractions for simplifying the implementation of parsers, syntax-aware editors, and code generators. Semantics specification formalisms such as Redex [6] and K [5] focus on modeling the semantics of programming languages at a high level of abstraction. Specifications can be used to simulate execution for the purpose of detecting specification errors. In practice, this means that aspects of a language definition are often implemented several times in different forms.

3. Separation of Concerns in Spoofax
Language workbenches are language development tools that considerably lower the threshold for software engineers to develop DSLs by abstracting over commonality in language implementations. The focus of my research group at TU Delft is to contribute to the development of language workbenches by developing language engineering techniques that simplify the work of language designers. We validate and test these techniques through integration in the Spoofax Language Workbench [8]. The key design principle for Spoofax is that language implementers should not have to be concerned with low-level implementation details. Thus, Spoofax generates a language-specific Eclipse plugin from a Spoofax ‘pro-
binding rules

\[
\begin{align*}
\text{Lam}(\text{param}, \text{e}) : & \quad \text{scopes Variable} \\
\text{Param}(\text{name}, \text{t}) : & \quad \text{defines Variable name of type t} \\
\text{Var}(\text{name}) : & \quad \text{refers to Variable name} \\
\text{Let}(\text{name}, \text{t}, \text{e1}, \text{e2}) : & \quad \text{defines Variable name of type t in e2}
\end{align*}
\]

Figure 1. NaBL name binding and scope rules for lambda terms

NaBL is a language for the specification of the name binding and scope rules of programming languages [9]. Figure 1 gives an example name binding specification for the lambda calculus. Based on such a specification, an incremental name resolution algorithm is automatically generated [15].

**TS** is a language for the specification of type analysis rules that complement the name binding rules of NaBL. Using the same task engine as targeted by NaBL, an incremental type checker is generated.

**DynSem** is a language for specification of the operational semantics of programming languages based on the work of Peter Mosses on implicitly modular operational semantics (IMOS) [10]. The goal is to generate fast interpreters from DynSem specifications.

We are currently working on mappings to Coq from these languages in order to support proving properties of languages.

---

**References**


