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

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.

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.

ACM 978-1-4503-2773-2/14/04.
http://dx.doi.org/10.1145/2584469.2584662

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

  Lam(param, e) :
    scopes Variable

  Param(name, t) :
    defines Variable name of type t

  Var(name) :
    refers to Variable name

  Let(name, t, e1, e2) :
    defines Variable name of type t in e2
```

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

gram' consisting of syntax definitions, transformation rules, and editor service configurations.

Syntax definition in Spoofax is based on the declarative syntax definition formalism SDF2 [12, 11], which supports the full class of context-free grammars based on scannerless generalized-LR parsing. From such a syntax definition, syntactic editor services such as syntax checking and syntax highlighting are automatically derived. Automatic generation of syntax error recovery rules ensures a robust editing experience [2].

All semantic operations on the abstract syntax trees produced by SDF generated parsers are expressed as tree transformations in the Stratego transformation language[1, 13]. In Stratego transformations can be expressed concisely using programmable rewriting strategies. Semantic operations include code generation and type checking, but also semantic editor services such as reference resolution and code completion.

Using Spoofax we have developed compilers and IDEs for a number of languages, including the WebDSL web programming language, that are used in daily practice. And the Spoofax languages are bootstrapped, of course.

3.1 Limitations

While Stratego is a natural fit for transformation and code generation, it leads to programmatic encoding for other concerns. In particular, name resolution proved to be a source of complexity in language implementations. The usual syntax directed style of name and type analysis was not a good fit with the requirements of an IDE, which needs access to the definitions of names at any time.

To realize good response times from the editor, analyses should be incremental such that their effort is proportional to the change made by the user. This requirement caused additional complexity for analyses such as name and type analysis. Incrementality also proved to be a source of complexity for compilation and code generation.

Finally, language workbenches do not provide assistance to language designers in the area of reasoning about language properties such as type soundness.

3.2 Radical Separation of Concerns

To address these issues, we are currently investigating a more radical separation of concerns. To that purpose we are designing declarative meta-languages specialized to the various concerns of language definition that can be used as the source of production quality (incremental) semantic operations *and* as a model for reasoning about language properties.

- **SDF3** is a redesign of the SDF2 syntax definition formalism, which adds generation of formatters and syntactic completion rules [14].
- **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 (IMSOS) [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

[1] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *SCP*, 72(1-2):52–70, 2008. 2

[2] M. de Jonge, L. C. L. Kats, E. Visser, and E. Söderberg. Natural and flexible error recovery for generated modular language environments. *TOPLAS*, 34(4):15, 2012. 2

[3] S. Efftinge et al. openarchitectureware user guide. version 4.3. Available from http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/, April 2008. 1

[4] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, 2006. 1

[5] C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 533–544, 2012. 1

[6] M. Felleisen, R. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009. 1

[7] JetBrains. Meta programming system. https://www.jetbrains.com/mps. 1

[8] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463, 2010. 1

[9] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *SLE*, pages 311–331, 2012. 2

[10] P. D. Mosses and M. J. New. Implicit propagation in structural operational semantics. *ENTCS*, 229(4):49–66, 2009. 2

[11] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *CC*, pages 143–158, 2002. 2

[12] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. 2

[13] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *ICFP*, pages 13–26, 1998. 2

[14] T. Vollebregt, L. C. L. Kats, and E. Visser. Declarative specification of template-based textual editors. In *LDTA*, page 8, 2012. 2

[15] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In *SLE*, pages 260–280, 2013. 2