Science of Computer Programming 97 (2015) 11–16

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Understanding software through linguistic abstraction

Eelco Visser

*Software Engineering Research Group, Department of Software and Computer Technology, Delft University of Technology, The Netherlands*

ARTICLE INFO

*Article history:*
Received 9 October 2013
Accepted 3 December 2013
Available online 31 December 2013

*Keywords:*
Linguistic abstraction
Programming languages
Domain-specific languages
Software understanding

ABSTRACT

In this essay, I argue that linguistic abstraction should be used systematically as a tool to capture our emerging understanding of domains of computation. Moreover, to enable that systematic application, we need to capture our understanding of the domain of linguistic abstraction itself in higher-level meta languages. The argument is illustrated with examples from the SDF, Stratego, Spoofax, and WebDSL projects in which I explore these ideas.

## 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 often still require low-level programmatic encodings of domain concepts. Or as Alan Perlis formulated it in one of his famous epigrams [1]: "A programming language is low level when its programs require attention to the irrelevant". A fixed set of (Turing Complete) programming constructs is sufficient to express all possible computations, but at the expense of considerable encoding that obfuscates the concepts under consideration. This essay argues that linguistic abstraction should be used systematically as a tool to capture our emerging understanding of domains of computation. Moreover, to enable that systematic application, we need to capture our understanding of the domain of linguistic abstraction *itself* in higher-level meta languages. The argument is illustrated with examples from the SDF, Stratego, Spoofax, and WebDSL projects in which I explore these ideas. A thorough investigation of the literature on this topic is beyond the scope of this short essay.

## 2. From design patterns to linguistic abstractions

A design pattern describes an approach (or a family of approaches) to solve a reoccurring problem in software development. A design pattern is a programming recipe that is applied manually by a programmer. When a design pattern is understood well, we can recognize formalizable regularity in the problem pattern and its encodings. A linguistic abstraction can then be used to formalize the design pattern in a language construct. To understand this process, let's examine the classical example of procedural abstraction.

*E-mail address:* visser@acm.org.
*URL:* http://eelcovisser.org.

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.12.001

```
label P
pop r3
pop r2
pop r1
// instructions for P
jump r3 // return
```

```
push v1
push v2
push L
goto P
label L
```

**Fig. 1.** Encoding a procedure in an imaginary assembly language with labels, stack operations, and jumps. The procedure definition (top) pops arguments from the stack and stores the values in registers. The procedure call (bottom) pushes arguments and the return address on the stack and jumps to the procedure code.

### 2.1. Example: procedural abstraction

A procedure in assembly programming amounts to a design pattern for organizing reuse of code (Fig. 1). In its simplest form, a sequence of instructions that is used at multiple places in the program is given a label. When jumping to the label the address of the next instruction after the call is stored, so that the procedure knows where to continue after completion. Passing arguments to a procedure requires storing the arguments on the stack and/or in registers. The particular protocol for doing this depends on the definition of the procedure. In principle, each procedure may require a different protocol. A *calling convention* standardizes the protocol for procedure calls in programs. However, a calling convention is a *convention* and is not enforced; adherence requires programmer discipline. This means that it is possible to deviate and make errors. Detecting such errors typically requires *debugging* rather than static analysis. Furthermore, calling conventions for different platforms may differ, for example, in the order in which arguments are pushed on the stack. Such differences reduce the portability of code.

Procedural abstraction is a linguistic abstraction that formalizes the design pattern of procedure definitions and calls. A procedure is introduced with a procedure definition that is *syntactically* recognizable as such:

```
def P(x, y) {
  // definition of P using parameters x and y
  return; // return control to caller
}
```

The definition introduces the name of the procedure and the names (and possibly types) of the arguments. A procedure call `P(e1, e2)` uses function notation known from mathematics to invoke a function, passing its arguments. Thus, the semantic concept is reified in syntax, allowing developers to directly express design intent ('language shapes thought').

We might now consider procedural abstraction as providing *syntactic sugar* for a particular implementation of procedures with jump and stack instructions. That particular implementation defines the *semantics* of procedures. However, we can go further and define a more abstract semantics that captures the essence of procedures; the fact that they name a parameterized sequence of instructions to which control is passed. Given that view, we can define mappings from the same notation to multiple alternative implementation models. In particular, we can make translations to the instruction sets and calling conventions of other platforms than the one that we originally developed the abstraction for, thus achieving portability of programs. Since these translations are automated we can ensure that the generated code is *correct by construction*, i.e. follows the rules of the design pattern. Alternatively, we can define an interpreter for programs, instead of a translation to a sequence of instructions.

In addition to varying implementation models, the abstraction makes it much easier to perform all sorts of static analyses on the program. Instead of having to identify the pieces of code that make up procedure definitions and calls, that information is now explicit at the syntactic level. For example, we can check that procedure calls are consistent in arity and type of arguments with procedure definitions, ruling out a large source of errors with a simple static analysis, *effectively enforcing consistent application of the design pattern*. Moreover, errors can be reported in the terminology of the abstraction ('procedure call has too few parameters'). In reasoning about the behavior of procedures in such analyses we only need to consider their abstract semantics.

A linguistic abstraction such as procedural abstraction *captures our understanding of a concept in software*. Over time the understanding of the abstraction in terms of the original implementation model erodes. New programmers learn to program with procedures without ever learning the underlying implementation scheme (or the mathematical semantics for that matter). The concept is no longer a convenience, but a first-class concept in thinking about software construction.

### 2.2. Generalization

The process from design pattern to linguistic abstraction has occurred time and again in the history of programming languages. We have linguistic abstractions for *structured control-flow* (if–then–else vs goto), *automatic memory management* (garbage collection vs manual alloc/free), *data abstraction* (abstract data types and objects), and *modules* (inheritance, traits, mixins). Of course, these abstractions are well known to any users of modern programming languages. They are so ingrained and internalized in the conceptual framework of computer science that they are taken for granted. That leads to the perception that there is a mostly fixed and final set of abstractions for programming that is sufficient for all eternity. Underlying this is a paradigm of simplicity that is driven by reducing the complexity of *compilers* rather than the programs that they compile. However, many areas of software development are in need of linguistic abstraction. For example, data persistence, data services, concurrency, distribution, access control, data invariants, and workflow currently require programmatic encodings in general purpose programming languages. As an example, let's consider an example of linguistic abstraction in web programming.

### 2.3. Example: web programming

A typical three-tier web program is a distributed software system with parts running in the browser, the web server, and the database. Each of these tiers runs different programming languages. The browser is programmed with HTML, CSS, and JavaScript; the server is programmed with a general purpose language such as Java or Ruby; and the database is programmed with a query language such as SQL. This entails that a web program is really a jumble of programs in these different languages. As a result, little static checking is or can be done of the consistency between fragments in different languages and failures are detected late in the development life cycle [2]. The WebDSL language addresses this problem by linguistically integrating DSLs for different concerns of web programming [3]. The language integrates sub-languages for persistent data models, user interface templates [4], access control policies [5], and data validation [6]. The integration enables early detection of failures, in particular cross-concern safety violations [2]. In addition to integration of languages, WebDSL introduces linguistic abstractions that formalize design patterns in the domain. Let's consider two examples, page navigation and the model-view pattern.

*Page navigation.* Users navigate between the pages of a web application through links. One expects that links to internal pages are correct, i.e. that they point to defined pages, passing valid parameters. In regular web programming approaches this is hard to check since page addresses are represented as string encoded URLs and processed using string manipulation. The programming languages in which these encodings are defined has no notion of pages and links, and cannot check for consistency as part of its regular static checking. An additional static analysis would have a hard time identifying the parts of programs concerned with URLs.

In a WebDSL program the consistency between page definitions and links can be checked easily, since the concept is captured in a linguistic abstraction. A page definition such as

```
page profile(user: User) { /* markup for page content */ }
```

introduces a page with a name and formal parameters, which can be arbitrary (persistent) object types. A page definition produces the code for routing, i.e. decoding an incoming page request URL, fetching objects from the database to instantiate the page parameters, and composing the response to the request based on the page body.

As page definitions in WebDSL are similar to function definitions, links are similar to function *calls*. A navigate clause such as

```
navigate profile(user) { output(user.name) }
```

generates a link to a page, passing it arguments of the appropriate type. (The argument between curly braces produces the anchor for the link.) By formalizing the notion of page definitions and links, their consistency can be checked at compile-time. That is, the compiler can check that the page in a link is defined and that the arguments passed to the page are of the right type. The implementation takes care of generating a correct URL to the page, with appropriate keys to represent the argument objects.

*Model-view.* Modifying content in a web application is typically organized using the model-view–controller design pattern. An internal data structure (model) is rendered as an HTML form (view) populating input elements with data. On submission of the form, the form request is processed by a function on the server (controller) that decodes the form request, validates data against data invariants, binds values to the data model, and persists the data in the database. When encountering validation errors the controller re-renders the view with error messages. This design pattern is typically re-implemented for each form.

WebDSL avoids the definition of separate controllers by automating the page life cycle. A form in a page such as in Fig. 2 takes care of rendering the form in HTML. However, in addition to generating the view, a page definition doubles as controller. On a `POST` request, the page definition is interpreted to perform all the operations normally performed by a separate controller. The changed data in the form request are bound to the corresponding l-values (e.g. `user.name`) in the form.

```
page editprofile(user: User) {
  action save() { return profile(user); }
  form{
    input(user.name)
    input(user.description)
    submit save() { "Save" }
  }
}
```

**Fig. 2.** Model-view pattern in WebDSL.

The data are validated against the data invariants for the data types. In case of validation conflicts, the page is re-rendered with embedded error messages. If validation succeeds, the changed objects are persisted in the database. In general, what we see here is that a linguistic abstraction can be interpreted for multiple purposes, unlike a programmatic encoding that defines an algorithm serving one specific purpose.

## 3. Meta linguistic abstraction

The examples above illustrate how linguistic abstractions formalize our understanding of concepts in software, thus allowing us to automate consistency checking, hide irrelevant implementation details, and reuse programs for multiple purposes. Using linguistic abstraction as a standard tool in the battle with software complexity requires that we can apply it effectively. A software language is a complex software system in its own right — consisting of syntactic and semantic analyzers, a translator or interpreter, and an interactive development environment (IDE) — and can take significant effort to design and implement. Language workbenches are language development tools aiming to considerably lower the threshold for software engineers to develop DSLs to automate software tasks [7]. The field was pioneered in the 1980s with projects such as the Synthesizer Generator [8] and the ASF + SDF MetaEnvironment [9]. Examples of modern language workbenches include MPS [10], Xtext [11], and Rascal [12]. My group at TU Delft has developed the *Spoofax Language Workbench* [13] based on the SDF and Stratego meta languages.

Despite advances in the field, many aspects of language definition still require programmatic encodings of language designs. As a result, language definitions are often only usable for a single purpose and it is hard to check consistency properties of language definitions. It is an important research challenge for the field of software language engineering to better understand the domain of linguistic abstraction itself, and capture that understanding in high-level linguistic abstractions that allow language designers to create new linguistic abstractions with much less effort. I examine two examples of high-level linguistic abstraction for language definition.

### *3.1. Example: syntax definition*

The syntax of a language concerns the form of its programs. The principal operation associated with syntax is parsing, the recognition of syntactically well-formed program texts and turning those into an abstract syntax tree structure for further processing. It is not uncommon for language engineers to encode the syntax of a language in a hand-written parser. However, many other operations and data structures depend directly on syntax. The schema for abstract syntax trees is directly related to the structure of programs. Pretty-printing, the formatting of an abstract syntax tree as text, is the inverse of parsing. Syntax-aware editors provide editor services such as syntax highlighting, parse error recovery, syntactic code completion, and outline views that are based on aspects of syntax. All these operations need to be implemented in addition to a hand-written parser.

Declarative syntax definition supported by generalized parsing abstracts from the details of parser implementations [14] and allows alternative interpretations in addition to a parser to be derived automatically. The SDF syntax definition formalism provides such an approach. Its first incarnation integrated lexical and context-free syntax based on GLR parsing in combination with a sophisticated lexical analysis scheme [15]. Its second incarnation completed the integration through Scannerless GLR (SGLR) parsing [16]. The example in Fig. 3 illustrates the third generation, SDF3 [17]. The production defines the syntax of function definitions with a name, list of arguments, optional return type, and a list of statements as body. As with previous versions of SDF, SDF3 definitions generate an SGLR parser. In addition, the formalism now explicitly defines the derivation of AST schemas (the `Function` is the constructor of the production). Furthermore, pretty-print rules and syntactic completion schemas can be derived from the template aspect of the production.

While syntax definition has been a successful example of declarative language definition, most other concerns of language definition require programmatic encodings. Recently, we have made progress in developing high-level linguistic abstractions for the specification of name binding rules.

### *3.2. Example: name binding*

Name resolution is a crucial static analysis performed after parsing for identifying which definitions belong to which uses of names in a program. The analysis is typically defined as an algorithm, programmatically encoding the conceptual name

```
Property.Function = <
  function <ID> (<Arg*; separator=",">) <ReturnType?> {
    <Stat*>
  }
>
```

```
Function : ID * List(Arg) * Option(ReturnType) * List(Stat) -> Property
```

**Fig. 3.** SDF3 production (top) and derived signature for AST constructor (bottom).

```
Function(f, _, _, _) :
  defines Function f
  scopes Variable
Param(x, _) :
  defines Variable x
Var(x) :
  refers to Variable x
Call(f, _) :
  refers to Function f
```

**Fig. 4.** NaBL name binding rules.

binding and scope rules of a language. While definitions with attribute grammars may abstract from some programmatic aspects, such as explicit scheduling, they still constitute an encoding of an algorithm. The NaBL name binding language [18] that we have recently developed aims to provide direct, explicit expression of name binding and scoping rules using the concepts of the domain. For example, consider the rules in Fig. 4 that define name binding for functions and variables. The rules define binding clauses that apply to abstract syntax tree patterns (such as `Var(x)`). The first two rules introduce *definitions* of names, for function definitions and function parameters, respectively. The last two rules define *references* of names, i.e. occurrences of names that refer to definitions with the same name. Finally, the rule for `Function` declares a function as a scope for variables. These rules do not define a recursive traversal over abstract syntax trees, or explicitly manipulate symbol tables. Instead, a name resolution algorithm that performs a traversal and stores binding information in a symbol table can be automatically derived from these rules. Moreover, the derived algorithm can be made *incremental*, such that the effort of reanalysis after a change during development is proportional to the size of the change [19].

### *3.3. Understanding other meta linguistic concerns*

Other language definition concerns such as type systems, dynamic semantics, and transformation also require better linguistic abstractions such that language designers can focus on design rather than implementation. By integrating verification techniques from semantics engineering, it will then be possible to automatically detect inconsistencies in language definitions.

## 4. The future of linguistic abstraction

With the increasing expressivity of meta languages and the powerful implementations produced by language workbenches, linguistic abstraction can become a professional tool routinely used by meta software engineers to develop an increasingly rich body of knowledge capturing our understanding of software.

## Acknowledgements

I met Paul Klint as an undergraduate student in his *Programming Environments* course at the University of Amsterdam in 1991. This was not my first encounter with programming languages and compilers. I had already built interpreters in Scheme, constructed a compiler with rewrite rules in ASF (without SDF), and taken a classical compiler construction course at the VU. But Paul put the field in a different perspective. In his characteristic charismatic lecture style, he introduced elegant (meta) language design and tool architecture instead of quintuples and Greek letters. The beauty of declarative syntax definition, the effortless syntactic extensibility of term rewriting, the modularity of language definitions were all equally fascinating. Even though the ASF + SDF MetaEnvironment was virtually unusable on the brand new Sun Sparc workstations with 8 MB of RAM we had in the lab, and despite (or perhaps especially due to) the numerous points for improvement that I started to observe, I was hooked. And the rest, as they say, is history. Thanks, Paul!

## References

[1] A.J. Perlis, Epigrams on programming, SIGPLAN Not. 17 (9) (1982).

[2] Z. Hemel, D.M. Groenewegen, L.C.L. Kats, E. Visser, Static consistency checking of web applications with WebDSL, J. Symb. Comput. 46 (2) (2011) 150–182, http://dx.doi.org/10.1016/j.jsc.2010.08.006.

[3] D.M. Groenewegen, Z. Hemel, E. Visser, Separation of concerns and linguistic integration in WebDSL, IEEE Softw. 27 (5) (2010) 31–37, http://dx.doi.org/10.1109/MS.2010.92.

[4] E. Visser, WebDSL: A case study in domain-specific language engineering, in: R. Laemmel, J. Visser, J. Saraiva (Eds.), Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, in: Lecture Notes in Computer Science, vol. 5235, Springer, Braga, Portugal, 2007, pp. 291–373.

[5] D.M. Groenewegen, E. Visser, Declarative access control for WebDSL: Combining language integration and separation of concerns, in: D. Schwabe, F. Curbera, P. Dantzig (Eds.), Proceedings of the Eighth International Conference on Web Engineering, ICWE 2008, 14–18 July 2008, IEEE, Yorktown Heights, New York, USA, 2008, pp. 175–188.

[6] D.M. Groenewegen, E. Visser, Integration of data validation and user interface concerns in a DSL for web applications, Softw. Syst. Model. 12 (1) (2013) 35–52, http://dx.doi.org/10.1007/s10270-010-0173-9.

[7] M. Fowler, Language workbenches: The killer-app for domain specific languages?, 2005, http://www.martinfowler.com/articles/languageWorkbench.html.

[8] T. Teitelbaum, T.W. Reps, The Cornell program synthesizer: A syntax-directed programming environment, Commun. ACM 24 (9) (1981).

[9] P. Klint, A meta-environment for generating programming environments, ACM Trans. Softw. Eng. Methodol. 2 (2) (1993), http://doi.acm.org/10.1145/151257.151260.

[10] JetBrains, Meta programming system, https://www.jetbrains.com/mps.

[11] S. Efftinge, M. Voelter, oAW xText: A framework for textual DSLs, in: Workshop on Modeling Symposium at Eclipse Summit, 2006.

[12] P. Klint, T. van der Storm, J.J. Vinju, Rascal: A domain specific language for source code analysis and manipulation, in: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20–21, 2009, IEEE Computer Society, 2009, pp. 168–177.

[13] L.C.L. Kats, E. Visser, The Spoofax Language Workbench: rules for declarative specification of languages and IDEs, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, ACM, Reno/Tahoe, Nevada, 2010, pp. 444–463.

[14] L.C.L. Kats, E. Visser, G. Wachsmuth, Pure and declarative syntax definition: paradise lost and regained, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, ACM, Reno/Tahoe, Nevada, 2010, pp. 918–932.

[15] J. Heering, P.R.H. Hendriks, P. Klint, J. Rekers, The syntax definition formalism SDF – reference manual, SIGPLAN Not. 24 (11) (1989), http://doi.acm.org/10.1145/71605.71607.

[16] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.

[17] T. Vollebregt, L.C.L. Kats, E. Visser, Declarative specification of template-based textual editors, in: S. Andova, A.M. Sloane (Eds.), Workshop on Language Descriptions, Tools, and Applications, Proceedings, 2012.

[18] G.D.P. Konat, L.C.L. Kats, G. Wachsmuth, E. Visser, Declarative name binding and scope rules, in: K. Czarnecki, G. Hedin (Eds.), Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26–28, 2012, in: Lecture Notes in Computer Science, vol. 7745, Springer, 2013, pp. 311–331, revised selected papers.

[19] G. Wachsmuth, G.D.P. Konat, V.A. Vergu, D.M. Groenewegen, E. Visser, A language independent task engine for incremental name and type analysis, in: M. Erwig, R.F. Paige, E.V. Wyk (Eds.), 6th International Conference on Software Language Engineering, Proceedings (SLE 2013), in: Lecture Notes in Computer Science, Springer, 2013.