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
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.