Science of Computer Programming 96 (2014) 211–229

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Attribute grammar macros

Marcos Viera [a,*], S. Doaitse Swierstra [b]

[a] Instituto de Computación, Universidad de la República, Montevideo, Uruguay
[b] Department of Computer Science, Utrecht University, Utrecht, The Netherlands

HIGHLIGHTS

- We describe a Haskell library of strongly typed first-class attribute grammars.
- We extend the library with a set of combinators to define semantic macros.
- We show how to define language semantics by re-using and adapting existing semantics.
- We show how to redefine attributes.
- We describe an example implemented using the techniques introduced in this paper.

ARTICLE INFO

Article history:
Received 1 April 2013
Received in revised form 27 September 2013
Accepted 17 January 2014
Available online 31 January 2014

Keywords:
Attribute grammar macro
Extensible language
First-class attribute grammar
Attribute redefinition
Haskell

ABSTRACT

Having extensible languages is appealing, but raises the question of how to construct extensible compilers and how to compose compilers out of a collection of pre-compiled components. We show how having attribute grammar fragments as first-class values can be put into good use to answer this question; the approach leads naturally to a plug-in architecture, in which a core compiler can be constructed out of a (collection of) pre-compiled component(s), to which extra components can safely be added as the need arises. We extend `AspectAG`, our Haskell library for building strongly typed first-class attribute grammars, with a set of combinators that make it easy to describe new semantics in terms of already existing semantics, just as syntax macros extend the syntax of a language. We especially show how semantics thus defined can be redefined, thus adapting some aspects of the behavior as defined by the macro system only.

1. Introduction

Ever since the introduction of the very first programming languages and the invention of grammatical formalisms for describing them, people have investigated how an initial language definition can be extended by someone else than the original language designer, preferably by providing separate language-definition fragments and without having to update an already existing code.

The simplest approach starts from the *text* which describes a compiler for the base language. Just before the compiler is compiled, several extra ingredients may be added textually. In this way we get great flexibility and there is virtually no limit to the things we may add. The Utrecht Haskell Compiler [1,2] has shown the effectiveness of this approach by composing a large number of attribute grammar fragments textually into a complete compiler description. This approach however is not very practical when defining relatively small language extensions; we do not want each individual user to have to generate a completely new compiler because he is defining a small extension. Another problematic aspect of this approach is that, by making the complete text of the compiler available for modification and extension, we lose important safety guarantees, as e.g. provided by the type system; we definitely do not want everyone to mess around with the delicate internals of a compiler for a complex language.

* Corresponding author.

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2014.01.014

So the question arises of how we can reach the effect of textual composition, but without opening up the whole compiler source. A commonly found approach is to use so-called *syntax macros* [3], which enable the programmer to add *syntactic sugar* to a language by defining new notation *in terms of already existing syntax*.

In this paper we focus on how to provide such mechanisms also *at the semantic level* [4]. As a running example we take a minimal expression language described by the grammar:

$$\begin{aligned} \mathit{expr} &\rightarrow \texttt{"let"}\ \mathit{var}\ \texttt{"="}\ \mathit{expr}\ \texttt{"in"}\ \mathit{expr} \mid \mathit{term}\ \texttt{"+"}\ \mathit{expr} \mid \mathit{term} \\ \mathit{term} &\rightarrow \mathit{factor}\ \texttt{"*"}\ \mathit{term} \mid \mathit{factor} \\ \mathit{factor} &\rightarrow \mathit{int} \mid \mathit{var} \end{aligned}$$

with the following abstract syntax (as a Haskell data type):

```
data Root = Root {expr :: Expr}
data Expr = Cst  {cv   :: Int}
          | Var  {vnm  :: String}
          | Mul  {me1  :: Expr,   me2  :: Expr}
          | Add  {ae1  :: Expr,   ae2  :: Expr}
          | Let  {lnm  :: String, val  :: Expr, body :: Expr}
```

Suppose we want to extend the language with an extra production for defining the square of a value. A syntax macro aware compiler might accept definitions of the form $\mathit{square}\ (\mathit{se} :: \mathit{Expr}) \Rightarrow \mathit{Mul}\ \mathit{se}\ \mathit{se}$, translating the new syntax into the existing abstract syntax.

Although this approach may be very effective and seems attractive at the first sight, such transformational programming [5] has its shortcomings. As a consequence of mapping the new constructs onto existing constructs and performing any further processing on this simpler, but often more detailed program representation, feedback from later stages of the compiler is given in terms of the intermediate program representations in which the original program structure may be hard to recognize. For example, if we do not change the pretty printing phase of the compiler, the expression *square* 2 will be printed as $2 * 2$, and worse, type-checking the expression *square True* will lead to more than a single error message. Hence the implementation details shine through, and the produced error messages can be confusing or even incomprehensible. Similar problems show up when defining embedded domain specific languages: the error messages from the type system are typically given in terms of the underlying representation [6].

In a previous paper [7] we introduced `AspectAG`: a Haskell library for defining first-class attribute grammars, which can be used to implement a language semantics and its extensions in a safe way, i.e. by constructing a core compiler as a (collection of) pre-compiled component(s), to which extra components can safely be added as need arises. In this paper we show how we can define new semantics by re-using and adapting existing semantics, in the form of *attribute grammar macros*. We also show how the new semantics thus defined can be *adapted* at the places where it makes a difference, e.g. in pretty printing and presenting error messages.

The functionality provided by the combination of attribute grammar macros and redefinition is similar to the *forwarding attributes* [8] technique for higher-order attribute grammars, implemented in the Silver AG system [9]. We however implement our proposal as a Haskell library in such a way that the consistency of a composite system is checked by the Haskell type checker. With an attribute grammar being *consistent* we mean that:

- every attribute referred to has a defining occurrence,
- no attribute is defined more than once (*closure test*) and
- all defining expressions are of the correct type.

Since the underlying evaluation mechanism is based on lazy evaluation we do not (have to) perform a *circularity test*, and even allow the use of circular attribute grammars like the one underlying the online pretty-printing algorithm [10].

In Section 2 we give a top-level overview of our approach. In order to make the paper self-contained we describe our library for the first-class attribute grammars in Section 3. The library is extended in Section 4 with a set of combinators for defining semantic macros. In Section 5 we introduce combinators that provide access to the attributes of the input of a macro, and we show how to redefine attributes in Section 6. An example implemented using the techniques introduced in this paper is introduced in Section 7. Finally, in Section 8 we discuss related work, and we close in Section 9 by presenting our conclusions and the future work.

2. Attribute grammar combinators

Before delving into the technical details, we show in this section how we implement the semantics of our running example language and some simple extensions. We have chosen to keep our example very simple, and to focus on the introduction of the underlying techniques. For a more involved example we refer to Section 7.

The semantics are defined by two aspects: pretty printing, realized by a synthesized attribute *spp* of type *PP_Doc*, which holds a pretty printed document, and expression evaluation, realized by two attributes: a synthesized *sval* of type *Int*, which