Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Embedding attribute grammars and their extensions using functional zippers

Pedro Martins [a,1], João Paulo Fernandes [a,b], João Saraiva [a], Eric Van Wyk [c,2], Anthony Sloane [d]

[a] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Braga, Portugal
[b] Reliable and Secure Computation Group ((rel)ease), Universidade da Beira Interior, Covilhã, Portugal
[c] Department of Computer Science and Engineering, University of Minnesota, Minneapolis, USA
[d] Department of Computing, Macquarie University, Sydney, Australia

A R T I C L E   I N F O

Article history:
Received 17 April 2014
Received in revised form 20 February 2016
Accepted 15 March 2016
Available online xxxx

Keywords:
Attribute grammars
Functional programming
Functional zippers
Bidirectional transformations

A B S T R A C T

Attribute grammars are a suitable formalism to express complex software language analysis and manipulation algorithms, which rely on multiple traversals of the underlying syntax tree. Attribute grammars have been extended with mechanisms such as reference, higher-order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex computations. This paper studies an elegant and simple, zipper-based embedding of attribute grammars and their extensions as first class citizens. In this setting, language specifications are defined as a set of independent, off-the-shelf components that can easily be composed into a powerful, executable language processor. Techniques to describe automatic bidirectional transformations between grammars in this setting are also described. Several real examples of language specification and processing programs have been implemented.

1. Introduction

Attribute Grammars (AGs) [1] are a well-known and convenient formalism not only for specifying the semantic analysis phase of a compiler but also to model complex multiple traversal algorithms. Indeed, AGs have been used not only to specify real programming languages, for example `Haskell` [2], but also to specify sophisticated pretty printing algorithms [3], deforestation techniques [4,5] and powerful type systems [6].

All these attribute grammars specify complex and large algorithms that rely on multiple traversals over large tree-like data structures. To express these algorithms in regular programming languages is difficult because they rely on complex recursive patterns, and, most importantly, because there are dependencies between values computed in one traversal and used in following ones. In such cases, an explicit data structure has to be used to glue together different traversal functions.

E-mail addresses: prmartins@di.uminho.pt (P. Martins), jpf@di.ubi.pt (J.P. Fernandes), jas@di.uminho.pt (J. Saraiva), evw@cs.umn.edu (E. Van Wyk), anthony.sloane@mq.edu.au (A. Sloane).

[1] This author is supported by ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project ON.2 IC&DT Programa Integrado "BEST CASE – Better Science Through Cooperative Advanced Synergetic Efforts (Ref. BIM-2013_BestCase_RL3.2_UMINHO) and project FATBIT – Foundations, Applications and Tools for Bidirectional Transformation (Ref. FCOMP-01-0124-FEDER-020532).

[2] This author is partially supported by NSF Award #1047961.

http://dx.doi.org/10.1016/j.scico.2016.03.005
0167-6423/© 2016 Elsevier B.V. All rights reserved.

In an imperative setting those values are stored in the tree nodes (which work as a gluing data structure), while in a declarative setting such data structures have to be defined and constructed. In an AG setting, the programmer does not have to concern himself or herself with scheduling traversals, nor on defining gluing data structures.

Recent research in attribute grammars has proceeded primarily in two directions.

Firstly, attribute grammars are embedded in regular programming languages with AG fragments as first-class values in the language: they can be analyzed, reused and compiled independently [7–11]. First class AGs provide:

i) A full component-based approach to AGs where a language is specified/implemented as a set of reusable off-the-shelf components;
ii) Semantic-based modularity, while some traditional AG systems use a (restricted) syntactic approach to modularity.

Moreover, by using an embedding approach there is no need to construct a large AG (software) system to process, analyze and execute AG specifications. First class AGs reuse for free the mechanisms provided by the host language as much as possible, while increasing abstraction in the host language. Although this option may also entail some disadvantages, e.g. error messages relating to complex features of the host language instead of specificities of the embedded language, the fact is that an entire infrastructure, including libraries and language extensions, is readily available at a minimum cost. Also, the support and evolution of such infrastructure is not a concern.

Secondly, AG-based systems have extended the standard AG formalism which improves the expressiveness of AGs. Higher-order AGs (HOAGs) [12,13] provide a modular extension to AGs in which abstract trees can be stored as attribute values. Reference AGs (RAGs) [14,15] allow the definition of references to remote parts of the tree, and, thus, extend the traditional tree-based algorithms to graphs. Finally, Circular AGs (CAGs) allow the definition of fix-point based algorithms. AG systems like Silver [16,17], JastAdd [18], and Kiama [11] all support such extensions.

However, and even considering their modern extensions, attribute grammars only provide support for specifying unidirectional transformations, despite bidirectional transformations being common in AG applications. Bidirectional transformations are especially common between abstract/concrete syntax. For example, when reporting errors discovered on the abstract syntax we want error messages to refer to the original code, not a possible de-sugared version of it. Or when refactoring source code, programmers should be able to evolve the refactored code, and have the change propagated back to the original source code.

In this work, we present the first embedding of HOAGs, RAGs and CAGs as first class attribute grammars, an embedding which is also powerful enough to express bidirectional transformations. Indeed, we revise the zipper-based AG embedding proposed in [9] to extend it with the bidirectional capabilities of [19]. We have used this embedding in a number of applications, e.g., in developing techniques for a language processor to implement bidirectional AG specifications and to construct a software portal.

In the remainder of the paper, we start by revising the concise embedding of AGs in `Haskell` of [9]. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [20] for a purely functional environment and represent a tree together with a subtree that is the focus of attention, where that focus may move within the tree. By providing access to any element of a tree, zippers are very convenient in our setting: attributes may be defined by accessing other attributes in other nodes. Moreover, they do not rely on any advanced feature of `Haskell` such as lazy evaluation or type classes. Thus, our embedding can be straightforwardly re-used in any other functional environment.

Finally, we extend our embedding with the primary AG extensions proposed to the AG formalism and with novel techniques for AG-based bidirectionalization systems.

This paper is organized as follows: in Section 2 we motivate our approach with the introduction of both our running example and AGs. In Section 3 we introduce zippers and explain how they can be used to embed AGs in a functional setting, and implement an AG in our setting.

Section 4 extends our running example and defines an AG implementing the scope rules for the newly defined language, with the aid of AG references. Section 5 describes the embedding of higher-order attributes as an extension to AGs and presents an example of an AG that uses this extension. In Section 6 we describe another AG extension, circularity, showing how it can be implemented with our technique, and give practical examples that build on the previous section.

In Section 7 a technique for defining a bidirectionalization system for AGs is presented, with an example providing automatic transformations between a concrete and an abstract version of our running example.

In Section 8 the reader is presented with works that relate to ours, either by having similar techniques or domains. Section 9 concludes this paper and Section 10 shows possible future research work.

2. Motivation

As a running example throughout this paper, we will describe and use the `LET` language, that could for example be used to define **let** expressions as incorporated in the functional languages `Haskell` [21] or `ML` [22].