# Zipper-Based Attribute Grammars and Their Extensions*

Pedro Martins[1], João Paulo Fernandes[1,2], and João Saraiva[1]

[1] High-Assurance Software Laboratory (HASLAB/INESC TEC), Universidade do Minho, Portugal
[2] Reliable and Secure Computation Group ((Rel)ease), Universidade da Beira Interior, Portugal
{prmartins,jpaulo,jas}@di.uminho.pt

**Abstract.** 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. Recently, Attribute Grammars have been extended with mechanisms such as references and high-order and circular attributes. Such extensions provide a powerful modular mechanism and allow the specification of complex fix-point computations. This paper defines 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. Several real examples of language specification and processing programs have been implemented in this setting.

## 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, like for example Haskell [2], but also to specify powerful pretty printing algorithms [3], deforestation techniques [4] and powerful type systems [5], for example.

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 in 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 different traversal functions. 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

* This work is funded 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 projects FCOMP-01-0124-FEDER-020532 and FCOMP-01-0124-FEDER-022701.

A. Rauber Du Bois and P. Trinder (Eds.): SBLP 2013, LNCS 8129, pp. 135–149, 2013.

be defined and constructed. In an AG setting, the programmer does not have to concern himself on scheduling traversals, nor on defining gluing data structures.

Recent research in attribute grammars is working in two main directions. Firstly, AG-based systems are supporting new extensions to the standard AG formalism that improve the AG expressiveness. Higher-order AGs (HOAGs) [6, 7] provide a modular extension to AGs. Reference AGs (RAGs) [8] allow the definition of references to remote parts of the tree, and, thus, extending the traditional tree-based algorithms to graphs. Finally, Circular AGs (CAGs) allow the definition of fix-point based algorithms. AG systems like Silver [9], JastAdd [10], and Kiama [11] all support such extensions. Secondly, attribute grammars are embedded in regular programming languages and AG fragments are first-class citizens: they can be analyzed, reused and compiled independently.

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, and ii) semantic-based modularity, while traditional AG specifications use a (restrict) syntax modular approach. Moreover, by using an embedding approach there is no need to construct a large AG (software) system to process, analyse and execute AG specifications: first class AGs reuse for free the mechanisms provided by the host language as much as possible, while increasing abstraction on 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.

This paper presents a novel technique combining these two AG advances.

First, we propose a concise embedding of AGs in `Haskell`. This embedding relies on the extremely simple mechanism of functional zippers. Zippers were originally conceived by Huet [12] to 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`. Thus, our embedding can be straightforwardly re-used in any other functional environment.

Second, we extend our embedding with the main AG extensions proposed to the AG formalism. In fact, we present the first embedding of HOAGs, RAGs and CAGs as first class attribute grammars. By this we are able to express powerful algorithms as the composition of AG reusable components. An approach that we have been using, e.g., in developing techniques for a language processor to implement bidirectional AG specifications and to construct a software portal.

## 2 Motivation

In this section we introduce the Desk language, that was proposed in [13], and that we will use as our running example throughout the paper. This language is small enough to be completely defined here while still holding central