Modern type-safe embedding of attribute grammars

An exercise with functional zippers in Haskell — Extended Abstract —

João Paulo Fernandes[1], Pedro Martins[2], Alberto Pardo[3], João Saraiva[4], Marcos Viera[3], and Tom Westerhout[5]

[1] CISUC – Universidade de Coimbra, Portugal jpf@dei.uc.pt
[2] University of California, Irvine, USA pribeiro@uci.edu
[3] Universidad de la República, Uruguay, {pardo,mviera}@fing.edu.uy
[4] Universidade do Minho, Portugal, saraiva@di.uminho.pt
[5] Radboud University, The Netherlands, t.westerhout@student.ru.nl

Abstract

Attribute grammars are a powerful, declarative formalism to implement and reason about programs which, by design, are conveniently modular. Although a full attribute grammar compiler can be tailored to specific needs, its implementation is highly non-trivial, and its long-term maintenance is a major endeavor. In fact, maintaining a traditional attribute grammar system is such a big effort that most systems that were proposed in the past are no longer active. Our approach to implementing attribute grammars is to write them as first-class citizens of a modern functional programming language. We improve a previous zipper-based attribute grammar embedding making it non-intrusive (i.e. no changes to the user-defined data types are required) and type-safe. On top of that, we achieve clearer syntax by using modern Haskell extensions. We believe that our embedding can be employed in practice to implement elegant, efficient, and modular solutions to real-life programming challenges.

Keywords: Embedded Domain Specific Languages · Zipper data structure · Memoization · Attribute Grammars · Higher-Order Attribute Grammars · Functional Programming

1 Introduction

Attribute Grammars (AGs) are a declarative formalism which was proposed by Knuth [7] in the late 60s and allows one to implement and reason about programs in a modular and convenient way. A concrete AG relies on a context-free grammar to define the syntax of a language, and on attributes associated with the productions of the grammar to define the semantics of that language. AGs

have been used in practice to specify real programming languages, like for example Haskell [2], as well as powerful pretty printing algorithms [16], deforestation techniques [4], and powerful type systems [11].

When programming with AGs, modularity is achieved due to the possibility of defining and using different aspects of computations as separate attributes. Attributes are distinct computation units, typically quite simple and modular, that can be combined into elaborated solutions to complex programming problems. They can also be analyzed, debugged, and maintained independently which eases program development and evolution.

AGs have proven to be particularly useful to specify computations over trees: given one tree, several AG systems such as [14, 3, 8, 17] take specifications of which values or attributes need to be computed and perform these computations. The design and coding efforts put into the creation, improvement, and maintenance of these AG systems, however, is tremendous which is often an obstacle to achieving the success they deserve.

An increasingly popular alternative approach to the use of AGs relies on embedding them as first-class citizens of general purpose programming languages [12, 9, 13, 15, 18, 1]. This avoids the burden of implementing a totally new language and associated system by hosting it in state-of-the-art programming languages. Following this approach one then exploits the modern constructions and infrastructure that are already provided by those languages and focuses on the particularities of the domain specific language being developed.

Functional zipper [6] is a powerful abstraction which greatly simplifies the implementation of traversal algorithms performing a lot of local updates. Functional zippers have successfully been applied to construct an AG embedding in Haskell [9, 10]. Despite its elegance, this solution had a major drawback which prevented its use in real-world applications: attributes were not cached, but rather repeatedly recomputed which severely hurt performance. Recently, this flaw has been eliminated [5] and replaced with a different one: the approach became intrusive, i.e. to benefit from the embedding user-defined data structures have to be adjusted.

In this paper we present an alternative mechanism to cache attributes based on a self-organising infinite grid. This graph is laid on top of the user-defined algebraic data type (ADT) and mirrors its structure. The used-defined data type itself remains untouched. The embedding is then based on two (rather than one) coherent zippers traversing the data structures in parallel. On top of being non-intrusive our solution is completely type-safe. Modern Haskell extensions such as `ConstraintKinds` allow us to propagate constraints down in the ADT completely eliminating run-time type casts present in the previous versions. Another side benefit of using modern Haskell features is a cleaner syntax with less code being generated by means of Template Haskell.

References

1. Balestrieri, F.: The Productivity of Polymorphic Stream Equations and The Composition of Circular Traversals. Ph.D. thesis, University of Nottingham (2015)

2. Dijkstra, A., Fokker, J., Swierstra, S.D.: The architecture of the Utrecht Haskell compiler. In: Haskell Symposium. pp. 93–104 (2009)
3. Dijkstra, A., Swierstra, D.: Typing Haskell with an Attribute Grammar (Part I). Tech. Rep. UU-CS-2004-037, Institute of Information and Computing Sciences, Utrecht University (2004)
4. Fernandes, J.P., Saraiva, J.: Tools and Libraries to Model and Manipulate Circular Programs. In: Symposium on Partial Evaluation and Program Manipulation. pp. 102–111. ACM (2007)
5. Fernandes, J.P., Martins, P., Pardo, A., Saraiva, J., Viera, M.: Memoized zipper-based attribute grammars and their higher order extension. Science of Computer Programming (2018)
6. Huet, G.: The zipper. Journal of functional programming **7**(5), 549–554 (1997)
7. Knuth, D.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (June 1968), *Correction: Mathematical Systems Theory* 5 (1), March 1971.
8. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In: International Conference on Compiler Construction. pp. 298–301. Springer-Verlag (1998)
9. Martins, P., Fernandes, J.P., Saraiva, J.: Zipper-based attribute grammars and their extensions. In: Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings. pp. 135–149 (2013)
10. Martins, P., Fernandes, J.P., Saraiva, J., Wyk, E.V., Sloane, A.: Embedding attribute grammars and their extensions using functional zippers. Science of Computer Programming **132**, 2 – 28 (2016), selected and extended papers from SBLP 2013
11. Middelkoop, A., Dijkstra, A., Swierstra, S.D.: Iterative type inference with attribute grammars. In: International Conference on Generative Programming. pp. 43–52. ACM (2010)
12. de Moor, O., Backhouse, K., Swierstra, D.: First-Class Attribute Grammars. In: 3rd. Workshop on Attribute Grammars and their Applications. pp. 1–20. Ponte de Lima, Portugal (2000)
13. Norell, U., Gerdes, A.: Attribute Grammars in Erlang. In: Workshop on Erlang. pp. 1–12. 2015, ACM (2015)
14. Reps, T., Teitelbaum, T.: The synthesizer generator. SIGPLAN Not. **19**(5), 42–48 (Apr 1984)
15. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. Electronic Notes in Theoretical Computer Science **253**(7), 205–219 (2010)
16. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In: Third Summer School on Advanced Functional Programming. LNCS Tutorial, vol. 1608, pp. 150–206. Springer Verlag (1999)
17. Van Wyk, E., Bodin, D., Gao, J., Krishnan, L.: Silver: an Extensible Attribute Grammar System. Electronic Notes in Theoretical Computer Science **203**(2), 103–116 (2008)
18. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: International Conference on Functional Programming. pp. 245–256. ACM (2009)