Zipper-based Embedding of Modern Attribute Grammar Extensions*

Pedro Martins
Universidade do Minho, Portugal

Abstract. This research abstract describes the research plan for a Ph.D project. We plan to define a powerful and elegant embedding of modern extensions to attribute grammars. Attribute grammars are a suitable formalism to express complex, multiple traversal algorithms. In recent years there has been a lot of work in attribute grammars, namely by defining new extensions to the formalism (forwarding and reference attribute grammars, etc), by proposing new attribute evaluation models (lazy and circular evaluators, etc) and by embedding attribute grammars (like first class attribute grammars). We will study how to design such extensions through a zipper-based embedding and we will study efficient evaluation models for this embedding. Finally, we will express several attribute grammars in our setting and we will analyse the performance of our implementation.

1 Problem Description and Motivation

Attribute grammars (AGs) [1] are a convenient formalism not only for specifying the semantic analysis phase of a compiler but also to model complex multiple traversal algorithms. Traditional AG systems tailor their own syntax for the definition of AGs. Attribute grammars in concrete syntaxes are then automatically transformed into efficient implementations in general purpose programming languages such as Haskell, OCaml and C. LRC [2], UUAG [3] or Silver [4] are among the AG systems that are designed in this way.

For many applications, it is desirable to design and implement a special purpose language that is tailored to the characteristics of the problem domain. Perhaps the most well-known example of such a language is provided by the compiler compiler system *yacc*. In general, however, the design and implementation of a new programming language from scratch can be costly. For that reason, the designers of Simula-67 proposed that new languages are implemented through library interfaces, so the new *embedded* language inherits all the benefits of its *host* language, and the implementation effort is much reduced.

In the context of attribute grammars, this idea has already been explored [5–7]. Indeed, each of this embeddings benefits from the particular characteristics of the host language in order to achieve elegant attribute grammar solutions.

* This work has been partially supported by FCT (Portuguese Science Foundation) project AMADEUS, under grant (POCTI, PTDC/EIA/70271/2006)

Recently, an embedding for classic AGs has been developed in a functional language [8]. This embedding, of around 100 lines of code, relies on an extremely simple mechanism based on the notion of functional zippers [9].

With our work we intend to explore this new approach in a systematic way. Our goal is to fully develop a mature system with advanced AG constructions embedded in a zipper-based setting, namely circular attributes (fix-point computation), aspects (aspected oriented programming), higher-order and forwarding (functional programming), references (imperative programming), multiple inheritance (object oriented programming), strategies (strategic programming) and incremental attribute evaluation (incremental computation).

Once we define how such an embedding can be modeled in a functional setting, we will study different models of execution for the AGs. Finally, we will conduct a series of experiments in order to benchmark our system against other well-established ones.

2 Overview and Related Work

Attribute grammars have proven to be a suitable formalism to the design and implementation of both domain specific and general purpose languages, with powerful systems based on attribute grammars [10–12, 2] been constructed. While, in the beginning, AG systems were used mainly to specify and derive efficient (batch) compilers for formal languages, nowadays, AG-based systems are powerful tools that not only specify compilers, but also syntax editors [10], programming environments [2], visual languages [13], complex pretty printing algorithms [14], program animations [15], etc. More recently, new extensions/features have been defined for attribute grammars, like forwarding attribute grammars [16], higher-order attribute grammars [17, 15], reference attribute grammars [18], multiple inheritance [19], aspect oriented attribute grammars [20].

While these extensions can be considered standard in traditional systems, they have not yet been studied in the context of modern functional AG embeddings, such as [5, 7]. Given the simple mechanism of [8], however, we believe that some extensions should be simple to obtain in that setting, while others, although probably requiring improvements on the embedding mechanism itself, still can be achieved elegantly.

With our work, we also propose to make incremental computation available within our embedded AG system. Reps was the first to use incremental attribute evaluation in the structured editors produced by the Synthesizer Generator System [10]. The LRC system [2] is a programming environment generator that uses a strict, purely functional attribute evaluators and incrementality is obtained via function memoisation [21]. The Eli [12] AG systems produce visual programming environments but it does not support incremental evaluation, yet. The ASF+SDF is a programming environment generator based on the paradigm of term re-writing [22]. The action semantics environment was developed with ASF+SDF [23]. None of these systems use incremental evaluation.

In the context of functional programming, John Hughes was the first to work on the incremental evaluation of lazy programs [24]. Acar *et al* [25] presented

a general technique for the incrementalisation of functional programs. Magnus Carlsson modeled this work as a Haskell library [26]. These techniques, however, do not handle lazy evaluation. In [8], the authors show that within their embedding, incremental computation can be obtained via function memoization. Their techniques, however, do not require lazy evaluation. This means that if we rely on lazyness in any of our extensions, this approach to incrementality may break down and further studies are necessary. Even if this does not occur, the impact of memoization in the authors' approach still needs to be evaluated.

3 Research Method

The aim of this project is to embed advanced features of the AG paradigm into a general purpose, lazy, and purely functional language. This goal builds upon the definition attribute grammars as a domain-specific embedded language in Haskell by [8]. That is, attribute grammars become a Haskell library of higher-order and lazy functions, and we want to model in this library new language concepts like circular attributes, aspects, higher-ordeness, forwarding, references, multiple inheritance, strategies, and, finally, incremental evaluation.

We also intend to conduct a systematic performance study of traditional attribute grammar evaluators *versus* their implementation as a library in Haskell. We want to verify with realistic examples if a highly optimized functional implementation of AGs (lazy, strict and deforested evaluators [21]) is really faster than the simple Haskell library. The outcome of the performance experiments, will allow us to to recommend improvements both to existing compilers for Haskell, attribute grammar based systems, and incremental programming environments.

The phases described below constitute the main work areas for our work:

3.1 Design: For many applications, it is desirable to design and implement a special purpose language that is tailored to the characteristics of the problem domain. However, the design and implementation of a new programming language from scratch is usually costy. The idea of embedded languages has been enthusiastically embraced by the functional programming community [14, 7, 8].

The first goal to achieve in this project is to enhance the zipper-based embedding of [8] with advanced AG features such as aspects, higher-order and circular attributes, references, multiple inheritance and incremental attribute evaluation. In order to achieve this, we will design and propose different implementations for each feature, and each proposal will possibly rely on a different advanced feature of the host language, Haskell. Then, we will conduct several experiments in order to realize which proposal achieves our goal as elegantly as possible.

3.2 Implementation: One of the uses of AGs today is in the creation of programming environments (also called language-based editing environments), that report on syntactic and semantic errors as the program is being constructed. In such an environment, it is important that the attribute values are computed *incrementally* after each edit action, re-using results of previous computations where possible. Reps and Teitelbaum were the first to demonstrate the feasibility

of the idea [10]. It is partly because of this emphasis on incremental computation that the embedding of attribute grammars into functional programming was ignored: it was not clear at all how incremental computation could be achieved.

An important step was made by João Saraiva, who showed how the known attribute evaluation techniques could all be implemented in a strict, purely functional setting [21, 27]. He was however not able to apply these techniques as part of an implementation of attribute grammars as a software library in Haskell: a quite complex preprocessor was still required.

Acar *et al.* [25] presented a new technique for incremental computation of functional programs, which is completely general as it can be used to make any program incremental. Furthermore, it is implemented as a library of functions in ML. A remarkable property of Acar's work is that it maintains a dynamic dependency graph, as opposed to the static dependency graph used in all previous work on attribute grammars. Due to this, it potentially requires few recomputations after the input has been changed. This theoretical advantage may however be outweighed in practice by the additional book-keeping required.

With our work we will study different execution models for our extensions. In particular, we will work on the development of incremental models of execution.

3.3 Benchmarks: We will apply the library developed in the previous phases to realistic examples (Java grammar, Pretty-printing optimal algorithm [28], etc). Then, we will compare the performance of the obtained implementations with equivalent implementations obtained by other AG systems. Firstly, we will compare the performance of our library against other functional AG embeddings, whenever such a comparison is possible (recall that most of the features we want to embed in our system are not available in functional embeddings such as [5, 7]). Secondly, we will compare our implementations against the ones derived by standard AG systems from AG expressed in special purpose languages [2–4].

Research Questions: The following research questions have to be answered in the project: How can we correctly and elegantly embed advanced attribute grammar features,*e.g.*, reference, forwarding, higher-orderness, in the embedding of [8]? What is the impact of memoization in the embedding of [8] and in the extensions we define for it? If lazyness is necessary for any of our extensions, how can we restore incremental computation? In other words, how can we combine lazyness and incremental computation? How does the perfomance of our implementations compare to well-established others?

4 Conclusions

We propose to develop a mature attribute grammar system, embedded in a modern functional programming language, and with all advanced AG features incorporated. Later, a systematic analysis on this system will be conducted.

The beneficiaries of this research are implementors of functional programming languages [29, 30], attribute grammar-based systems [12, 14, 2] and programming environments [10, 22, 23, 31], because we provide experimental evidence to guide further work. The software artifacts we produce in the process will be of use to a wide audience of functional programmers.

References

1. Knuth, D.E.: Semantics of Context-free Languages. Mathematical Systems Theory **2**(2) (1968) 127–145 *Correction: Math. Systems Theory* 5, 1, pp. 95-96 (1971).
2. Kuiper, M., Saraiva, J.: Lrc - A Generator for Incremental Language-Oriented Tools. In Koskimies, K., ed.: 7th International Conference on Compiler Construction. Volume 1383 of LNCS., Springer-Verlag (1998) 298–301
3. Swierstra, D., Baars, A., Löh, A.: The UU-AG attribute grammar system (2004)
4. Wyk, E.V., Krishnan, L., Bodin, D., Johnson, E., Schwerdfeger, A., Russell, P.: Tool Demonstration: Silver Extensible Compiler Frameworks and Modular Language Extensions for Java and C. In: SCAM. (2006) 161
5. de Moor, O., Backhouse, K., Swierstra, D.: First-Class Attribute Grammars. In Parigot, D., Mernik, M., eds.: Third Workshop on Attribute Grammars and their Applications, WAGA'99, Ponte de Lima, Portugal, INRIA Rocquencourt (2000)
6. Sloane, A.M., Kats, L.C.L., Visser, E.: A pure object-oriented embedding of attribute grammars. In Ekman, T., Vinju, J., eds.: Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009). Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers (2009)
7. Viera, M., Swierstra, D., Swierstra, W.: Attribute Grammars Fly First-class: how to do Aspect Oriented Programming in Haskell. In: Procs. of the 14th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'09). (2009) 245–256
8. Fernandes, J., Sloane, A., Saraiva, J., Cunha, J.: A lightweight functional embedding of attribute grammars (in preparation). (2010)
9. Huet, G.: The zipper. Journal of Functional Programming **7**(5) (1997) 549–554
10. Reps, T., Teitelbaum, T.: The Synthesizer Generator. Springer (1989)
11. Jourdan, M., Parigot, D., Julié, C., Durin, O., Bellec, C.L.: Design, implementation and evaluation of the fnc-2 attribute grammar system. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation, New York, NY, USA, ACM (1990) 209–222
12. Kastens, U., Pfahler, P., Jung, M.T.: The Eli System. In: CC '98: Procs. of the 7th Int. Conf. on Compiler Construction, London, UK, Springer-Verlag (1998) 294–297
13. Kastens, U., Schmidt, C.: Vl-eli: A generator for visual languages (2002)
14. Swierstra, D., Azero, P., Saraiva, J.: Designing and Implementing Combinator Languages. In Swierstra, D., Henriques, P., Oliveira, J., eds.: 3rd Summer School on Adv. Funct. Programming. Volume 1608 of LNCS Tutorial. (1999) 150–206
15. Saraiva, J.: Component-based programming for higher-order attribute grammars. In: GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering, London, UK, Springer-Verlag (2002) 268–282
16. Wyk, E.V., Moor, O.d., Backhouse, K., Kwiatkowski, P.: Forwarding in attribute grammars for modular language design. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 128–142
17. Swierstra, D., Vogt, H.: Higher order attribute grammars. In Alblas, H., Melichar, B., eds.: International Summer School on Attribute Grammars, Applications and Systems. Volume 545 of LNCS., Springer-Verlag (1991) 48–113
18. Hedin, G.: Reference attributed grammars. In Parigot, D., Mernik, M., eds.: 2nd Workshop on Attribute Grammars and their Applications. (1999) 153–172
19. Mernik, M., Lenic, M., Avdicausevic, E., Zumer, V.: Multiple attribute grammar inheritance. In: Informatica. (2000) 319–328

20. de Moor, O., Peyton Jones, S., van Wyk, E.: Aspect-oriented compilers. In: Proceedings of the First International Symposium on Generative and Component-Based Software Engineering (GCSE '99). LNCS (1999)
21. Saraiva, J.: Purely Functional Implementation of Attribute Grammars. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands (1999)
22. van den Brand, M., Klint, P., Olivier, P.: Compilation and Memory Management for ASF+SDF. In Stefan Jähnichen, ed.: 8th International Conference on Compiler Construction. Volume 1575 of LNCS., Springer-Verlag (1999) 198–213
23. van den Brand, M., Iversen, J., Mosses, P.D.: An action environment. Sci. Comput. Program. **61**(3) (2006) 245–264
24. Hughes, J.: Lazy memo-functions. In Jouannaud, J.P., ed.: Functional Programming Languages and Computer Architecture. Volume 201 of LNCS., Springer-Verlag (1985) 129–146
25. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. In: POPL'02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, ACM (2002) 247–259
26. Carlsson, M.: Monads for incremental computing. In: ICFP'02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2002) 26–35
27. Saraiva, J., Swierstra, D., Kuiper, M.: Functional Incremental Attribute Evaluation. In David Watt, ed.: 9th International Conference on Compiler Construction, CC/ETAPS'2000. Volume 1781 of LNCS., Springer-Verlag (2000) 279–294
28. Swierstra, D., Chitil, O.: Linear, bounded, functional pretty-printing. Journal of Functional Programming **19**(01) (2009) 1–16
29. Peyton Jones, S., Hughes, J., Augustsson, L., et al.: Report on the programming language Haskell 98. Technical report (1999)
30. Leroy, X.: The Objective Caml System - Documentation and User's Manual (1997)
31. Michiel, M.: Proxima : a presentation-oriented editor for structured documents. PhD thesis, Utrecht University, The Netherlands (2004)