Applying Attribute Grammars for Metamodel Semantics

Christoff Bürger
Software Technology Group
TU Dresden
Dresden, Germany
christoff.buerger@tu-dresden.de

Sven Karol
Software Technology Group
TU Dresden
Dresden, Germany
sven.karol@tu-dresden.de

Christian Wende
Software Technology Group
TU Dresden
Dresden, Germany
c.wende@tu-dresden.de

ABSTRACT

While current metamodelling languages are well-suited for the structural definition of abstract syntax and metamodelling infrastructures like the Eclipse Modelling Framework (EMF) provide various means for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured. Therefore, we propose the application of reference attribute grammars (RAGs) to alleviate the lack of support for formal semantics specification in metamodelling. We contribute the conceptual foundations to integrate metamodelling languages and RAGs, and present *JastEMF* — a tool for the specification of EMF metamodel semantics using JastAdd RAGs. The application of JastEMF is illustrated by an integrated metamodelling example.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.4 [**Programming Languages**]: Processors—*Translator writing systems and compiler generators*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

Keywords

Metamodelling, Attribute Grammars, Ecore, EMF, JastAdd

1. INTRODUCTION

Metamodelling is a vital activity for Model-Driven Software Development (MDSD). It covers the definition of structures to represent abstract syntax models, the specification of a concrete syntax, and the specification of the meaning of models [12]. While infrastructures like the Eclipse Modelling Framework (EMF)[1] provide various means for the specification of a textual or graphical concrete syntax, techniques for the specification of model semantics are not as matured [12].

[1]http://www.eclipse.org/emf/

ECOOP '2010 Maribor, Slovenia EU

In this paper we propose the application of RAGs [5] — a well-researched extension of Knuth's classic attribute grammars (AGs) [7] — to alleviate the lack of support for formal semantics specification in metamodelling. They enable (1) the specification of semantics on tree and graph-based abstract syntax structures with unique spanning trees, (2) completeness and consistency checks of semantic specifications, and (3) the generation of an implementation of semantics specifications.

The contributions of this paper are as follows: The next section provides a motivating example to explain common concerns in the specification of metamodels. In Section 3 we sketch general foundations for the application of RAGs for metamodel semantics. In Section 4 we demonstrate the feasibility of this idea by a technical integration of the Ecore metamodelling language and the JastAdd [3] attribute grammar system. In Section 5 we discuss the application of the integrated metamodelling approach using the example introduced in Section 2. Finally, we discuss related work in Section 6 and conclude our contribution in Section 7.

2. MOTIVATION

In this section we introduce the language *SiPLE* (Simple imperative Programming Language Example) that serves as a continuous example throughout this paper. Afterwards, we summarise several general tasks and objectives relevant for metamodelling and implementing a language like SiPLE.

2.1 A Quick Look at SiPLE

SiPLE's main features are boolean, integer and real arithmethics, nested scopes, nested procedure declarations, recursion, while loops and conditionals. We have chosen the example of SiPLE, because on the one hand its semantics are complex enough to show the scalability of the presented solution and on the other hand readers should be familiar with programming language semantics. To exemplify how SiPLE programs look, Listing 1 shows a basic program that computes and prints the first ten Fibonacci numbers.

Listing 1: Fibonacci Numbers in SiPLE

```
Procedure main()
Begin
 Procedure fibonacci(Var n:Integer) : Integer
 Begin
  Var result:Integer;
  If(n>1) Then
   result := fibonacci(n-2) + fibonacci(n-1);
  Else
```

```
    result := 1;
  Fi;
  Write result;
  Return result;
 End;
 fibonacci(10);
End;
```

Besides the classical language tooling, like a compiler or interpreter, we also like to support an editor with syntax highlighting, code completion and semantic property view and a common metamodel — i.e., compiler API — for SiPLE. To do so we intend to implement SiPLE using a standard metamodeling framework. Relying on a standardised framework enables the integration with tools for model analysis, or model transformation that are fundamental for MDSD. Furthermore, we benefit from framework services like editors, model persistency, or versioning.

2.2 Metamodelling Tasks and Objectives

The implementation of a language like SiPLE has to provide means to transform programs (i.e., models) starting from (1) their textual representation to (2) their abstract syntax representation and finally (3) representations of their static and execution semantics. In the metamodelling world, all these representations and transformations are related to the language's metamodel. It typically *declares* the data structures that are used for representing language constructs in abstract syntax models and is the interface for the *specification* of concrete syntax and semantics.

As depicted in Figure 1, each transformation's input and output model is built using specific kinds of constructs of the language metamodel. A first transformation — typically specified using regular expressions and context-free grammars — derives an abstract syntax tree (AST) from textual symbols. The data structure required to represent the AST is solely declared by the `Metaclasses`, `Attributes` and `Containment References` in the metamodel. In a second transformation, the structures that are declared as `Non-Containment References` (e.g., connecting variable usage with variable declarations) need to be derived, resulting in a reference-attributed model — i.e., the abstract syntax tree with a superimposed reference graph. Graphical editors often directly operate on such reference-attributed models (cf. Figure 1) and rely on a direct manipulation of non-containment references. A last transformation performs semantics evaluations to derive values for `Derived Attributes` and executes `Operations` declared in the metamodel. Note that the reference-attributed model and the full-attributed model still contain the abstract syntax tree as their *spanning tree*.

All the above mentioned transformations are important artefacts of a language and thus motivate a formal definition. However, formal approaches for the specification of static or execution semantics are not yet established in the metamodelling world. As depicted in Figure 1 we aim at closing this gap by the application of RAGs for the specification of metamodel semantics. Because RAGs rely on a specific representation of abstract syntax, their application in metamodeling requires an integration with metamodel constructs. In the next section we prepare such integration.

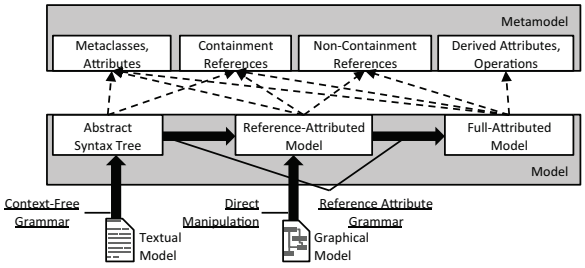

Figure 1: Transformations in Metamodelling.

3. APPLYING RAGS ON METAMODELS

RAGs are used to specify semantics for tree structures that are usually specified using a context-free grammar (CFG). Given a tree the RAG imposes a graph on it representing the language's semantics. This suggests that RAGs can be used to specify the transformation of abstract syntax trees to reference attribute graphs and full-attributed graphs (cf. Figure 1). To substantiate this intuition we investigate concepts of common metamodelling languages and distinguish them into syntactic (tree structure defining) and semantic (graph structure declaring) ones, such that RAGs can be applied to define the semantic concepts. Definition 3.1 summarises the foundations of integrating RAGs and metamodels:

Definition 3.1 (Metamodel Semantics): Let Ω be a metamodel and E_Ω the finite set of its elements. Let E_{syn_Ω} and E_{sem_Ω} be disjunct subsets of E_Ω, whereas $E_\Omega = E_{syn_\Omega} \cup E_{sem_\Omega}$. Let E_ω be the set of entities of a model instance $\omega \in \Omega$. Since $\omega \in \Omega$, all entities $e \in E_\omega$ have a type $t_e \in E_\Omega$. Let S_Ω be a function that defines for all $\omega \in \Omega$ for each entity $e \in E_\omega$ with $t_e \in E_{sem_\Omega}$ the value of e. We call S_Ω a metamodel semantic for Ω. Iff E_{syn_Ω} specifies a spanning tree for each $\omega \in \Omega$, S_Ω can be specified with an RAG.

The metamodel semantics S_Ω can depend on any metamodel element $me \in E_\Omega$. They even can depend on themselves, in which case they are only well-defined if there exists a fixpoint. Thus, different model instances can only have different semantics, if the semantics depend on semantic elements $me \in E_{syn_\Omega}$. Of course, it is no problem for S_Ω if elements of E_{sem_Ω} of a model instance have a predefined value — i.e., if instead of a tree the semantic evaluation starts from a graph as depicted in Figure 1. What remains to show is, which metamodeling concepts belong to E_{syn_Ω} and E_{sem_Ω} and that indeed E_{syn_Ω} specifies a tree structure.

Most metamodelling languages support (1) *metaclasses* consisting of (2) *non-derived* and (3) *derived attributes* and (4) *operations*. Metaclasses can be related to each other by (5) *containment* and (6) *non-containment relationships*. Non-derived attribute values must be given by the model (e.g., in concrete syntax) and, thus, are in E_{syn_Ω}. Containment references model that instances of a metaclass C_1 consist of instances of a metaclass C_2. The contained C_2 instances are an inextricable, structural part of the C_1 instances. The relationship between C_1 and C_2 is a composite and iff an instance $e_2 \in C_2$ is a composite of an instance $e_1 \in C_1$, e_1 cannot be a composite of e_2. Thus, containment relationships specify tree structures. They are in

E_{syn_Ω}. Derived attributes and operations model the values that can be calculated from other values of a given model. Non-containment relationships model arbitrary references between metaclasses. Thus, derived attributes, operations, and non-containment relationships are in E_{sem_Ω}.

4. A CONCRETE INTEGRATION

In this section, we discuss the integration of an exemplary metamodelling and RAG system. We introduce both approaches and discuss the details of their integration.

4.1 The EMF Metamodelling Infrastructure

The EMF is a common metamodelling infrastructure for the Eclipse platform providing metamodel development and implementation tools for Ecore. For the specification of textual syntax for metamodels and the generation of parsers, pretty printers and editors tools like EMFText [6] exist. There are also tools for graphical (diagrammatic) metamodel syntax, e.g., the Graphical Modelling Framework (GMF)[2]. For the specification of semantics, the EMF relies on Java source code that evaluates derived attributes or implements operations declared in the metamodel. Besides the application of the Object Constraint Language (OCL) or model-transformations, we are not aware of formal, mature techniques to specify static and execution semantics in EMF. For a discussion of existing semantics approaches we refer to Section 6.

4.2 The JastAdd Attribute Grammar System

JastAdd [3, 2] is an object-oriented compiler generation system. It allows to generate a Java implementation of a demand-driven semantics evaluator from a given RAG. Besides the basic attribute grammar concepts [7], JastAdd supports advanced AG extensions such as reference [5] (RAGs) and circular attributes [4]. Given a set of AST and attribute specifications the JastAdd compiler generates a Java class for each node type, accessors for the node's children nodes, and methods for each attribute defined for the node type. The code required for attributes' evaluation is generated into their method bodies. Consequently, evaluation of semantics can be triggered by accessing the corresponding methods.

4.3 Integration of EMF and JastAdd

The integration of EMF metamodels and JastAdd-based semantics requires (1) the integration of the Java classes that represent a language's abstract syntax in EMF and in JastAdd and (2) the application of the generated attribute evaluator to compute semantics.

Based on the integration foundations presented in Section 3 we derived a mapping of elements in the Ecore metamodel and concepts provided by JastAdd. The concrete mappings depicted in Figure 2 are grouped in two sets. The first set contains elements related only to model *syntax* (E_{sny_Ω}). The second set contains the elements related to model *semantics* (E_{sem_Ω}). In the second set constructs of the Ecore metamodel are typically used to declare the *semantic interface* of specific elements while the corresponding JastAdd construct specifies the element's derivation. Depending on its semantics multiple mappings are possible.

[2]http://www.eclipse.org/gmf/

Syntax in Ecore	Syntax in JastAdd
EClass	Node Type
EReference [containment]	Non-Terminal Child
EAttribute [non-derived]	Terminal Child
Semantics Interface in Ecore	**Semantics in JastAdd**
EAttribute [derived]	synthesized attribute, inherited attribute
EAttribute [derived, multiple]	collection attribute
EReference [non-containment]	collection attribute, reference attribute
EOperation	synthesized attribute, inherited attribute

Figure 2: Integrating Ecore and JastAdd.

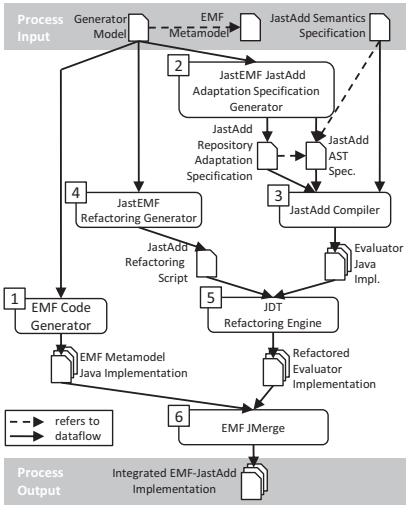

Figure 3: JastEMF's Generation Process.

To realise the integration of EMF and JastAdd, we implemented *JastEMF*[3]. Its generation process (cf. Figure 3) reuses the generators for JastAdd and EMF and merges the generated Java classes in accordance to the introduced mapping. As a first input the process requires an EMF generator model, which is used by the `(1) EMF Code Generator` to generate a metamodel implementation and the `(2) JastEMF JastAdd Adaptation Specification Generator` to derive an AST and JastAdd Repository Adaptation Specification. The repository adaptation specification contributes attribute specifications that adapt the JastAdd evaluator to use the EMF repository instead of its own internal repository. As a second input the process requires the developed metamodel semantics (JastAdd Semantics Specification). The JastAdd AST Specification, the JastAdd Repository Adaptation Specification and the JastAdd Semantics Specification are used by the `(3) JastAdd Compiler` to generate an appropriate AG Evaluator. To finally integrate this Evaluator with the metamodel implementation it has to be slightly refactored to incroporate metamodel naming conventions and package structures. Therefore, the `(4) JastEMF Refactoring Generator` derives a JDT[4] refactoring script from the metamodel and applies it `(5)`[5]. Finally, the Refactored Evaluator is merged with the EMF metamodel implementation using `(6) EMF JMerge`.

[3]`www.jastemf.org`
[4]http://www.eclipse.org/jdt/
[5]For an overview of the refactorings applied we refer to [1].

5. IMPLEMENTING SIPLE

In the following, we present our SiPLE implementation that satisfies the objectives of Section 2. The implementation is built around an Ecore metamodel that serves as API and repository for a compiler and interpreter for SiPLE. JastAdd is used to specify SiPLE's static and execution semantics. Using JastEMF we automatically integrate the EMF metamodel and JastAdd semantics. To generate a parser, pretty printer and sophisticated editor for SiPLE's textual syntax we used EMFText. Because the specification of concrete syntax with CFGs is out of scope in this paper, we will not investigate SiPLE's concrete syntax specification.

5.1 Modelling Abstract Syntax with Ecore

The SiPLE metamodel is presented in Figure 4. A `CompilationUnit` consists of `Declarations`, which can be `VariableDeclarations` declaring a variable's name and type or `ProcedureDeclarations` declaring a procedure. Each procedure has a name, a return type, a list of parameters and a body that is a `Block`. Each `Block` consists of a `Statement` sequence. In SiPLE, nearly everything is a `Statement`, such as `While` loops, `If` conditionals, `Expressions`, `Declarations` and `VariableAssignments`. Expressions can be `BinaryExpressions` or `UnaryExpressions`, whose concrete sub-classes, e.g., `Addition` and `Not`, are not presented in the figure. Furthermore, there are primitive expressions such as `Constants`, `References` and `ProcedureCalls`.

As discussed in Section 2 and 3 this metamodel specifies a spanning tree (containment references, non-derived attributes) enriched with semantic interfaces (non-containment references, derived attributes, operations). For a better understanding, we assigned numbers to different parts of the semantic interfaces declared in the metamodel. Parts that are to be computed by name analysis are marked with **1**, e.g., each `VariableAssignment` and each `ProcedureCall` in a well-formed program has a reference pointing to their respective declaration. Parts depending on type analysis are labeled by **2**, e.g., the derived attribute `Type` represents the actual type of an `Expression`. **3** marked parts belong to dataflow analysis, which consists currently only of the `MaybeUndecidable` attribute. Parts labeled with **4** (e.g., `Interpret()`) declare the execution semantics interface for both runtime evaluation and constant folding.

5.2 Specifying Semantics with JastAdd

There are four semantic concerns that completely specify SiPLE's static and execution semantics. With JastAdd, each concern can be specified as an `aspect`. A `core` specification is used to *declare* all `concerns` (cf. Listing 2). The actual *definitions* reside in separate JastAdd specifications[6].

Listing 2: Excerpt from the SiPLE Core Spec.

```
aspect NameAnalysis { // [1] in Figure 4
  // Procedure name space:
  inh Collection<ProcedureDeclaration>
        ASTNode.LookUpPDecl(String name);
  syn ProcedureDeclaration
        ProcedureCall.Declaration();
  syn ProcedureDeclaration
        CompilationUnit.MainProcedure();
  // Variable name space:
  inh Collection<VariableDeclaration>
        ASTNode.LookUpVDecl(String name);
  syn VariableDeclaration Reference.Declaration();
  syn VariableDeclaration
        VariableAssignment.Declaration();
}
aspect TypeAnalysis { // [2] in Figure 4
  syn Type VariableDeclaration.Type();
  syn Type VariableAssignment.Type();
  syn Type ProcedureReturn.Type();
  syn Type Expression.Type();
}
aspect DataflowAnalysis { // [3] in Figure 4
  syn boolean Statement.MaybeUndecidable();
}
aspect Interpretation { // [4] in Figure 4
  public abstract Object Expression.Value(State vm)
    throws InterpretationException;

  syn State CompilationUnit.Interpret();
  public abstract void Statement.Interpret(State vm)
    throws InterpretationException;
}
```

[6]The specifications can be found at `www.jastemf.org`.

NameAnalysis SiPLE uses separate block-structured namespaces for variable and procedure declarations. Although there is a single global scope in each `CompilationUnit`, each block introduces a new private scope, shadowing declarations in the outside scope. No explicit symbol tables are required to resolve visible declarations — the AST is the symbol table.

TypeAnalysis SiPLE is meant to be a statically, strongly typed language. For each kind of expression its type is computed from the types of its arguments, e.g., if an addition has a `Real` number argument and an `Integer` argument the computed type will be `Real`. Static type checks are performed for arithmetic operations, assignments, conditionals (`If`, `While`), procedure calls and procedure returns.

DataflowAnalysis SiPLE currently only supports an analysis to decide for each statement and expression whether its evaluation always terminates — i.e., does not depend on procedure calls or loops (`MaybeUndecidable` attribute). This analysis in combination with an analysis that extracts all statements a reference's value can depend on can be used to perform constant folding. For a general discussion of dataflow analysis using JastAdd we refer to [10].

Interpretation SiPLE's execution semantics is also specified using JastAdd. We applied JastAdd's abbility to use Java method bodies for attribute specifications. This allows for a seamless integration of a Java implementation of the operational semantics and the declarative, RAG-based static semantics analysis. The interpretation is triggered with a call to a `CompilationUnit`'s `Interpret()` operation that initialises a state object representing a stack for procedure frames and traverses the program's statements by calling their `Interpret(State)` operation.

6. RELATED WORK

We are not aware of another approach integrating AGs with metamodelling. However, there are a number of approaches dealing with metamodel semantics.

First, constraint languages like the OCL standard[7] or XCheck[8] enable the specification of constraints over metamodels. Com-

[7]http://www.eclipse.org/modeling/mdt/?project=ocl
[8]http://www.openarchitectureware.org/

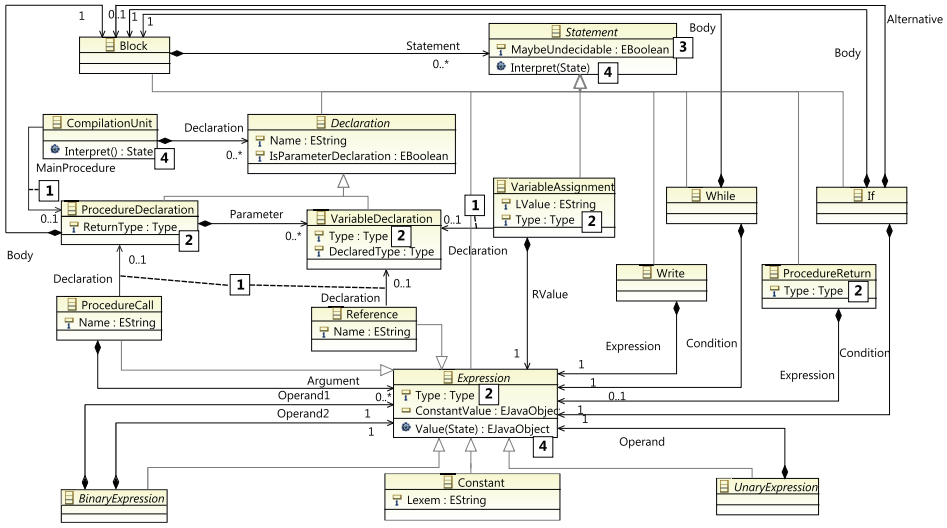

Figure 4: SiPLE's Metamodel.

pared to AGs, they are mainly used to define well-formedness of models and compute simple derived values. However, the lack of means for checking the consistency of such specifications (e.g., circularity in constraint definitions) discourages their application for complex semantics specifications.

A second category of approaches aims at providing an integrated environment for developing domain-specific languages (DSLs) (e.g., Kermeta [8], openarchitectureware[7]). They typically provide dedicated languages to specify abstract syntax and semantics but lack a formal background. Our approach aims at combining the convenience of such environments with the flexibility and reliability that a formal approach like AGs provides.

A third category of tools relies on model transformations or formalisms for graph rewriting (e.g., FUJABA [9]) for semantics specification. In general, graph-rewriting systems are harder to understand than AGs. Given a set of rewrite rules, it is complicated to foresee all possible consequences w.r.t. their application on start graphs. On the other hand, AGs require a basic spanning tree whereas graph rewriting does not rely on such assumptions. Furthermore, RAGs can only add information to an AST but not remove them or even change its structure. However, there are AG concepts such as higher order attributes [13] and rewrite rules which improve in that direction.

7. CONCLUSION

We presented a novel approach for specifying metamodel semantics based on RAGs and JastEMF as respective tool implementation. We practically demonstrated that RAGs can be beneficially combined with MDSD by specifying a simple imperative language based on the EMF. Our example solution includes the language's metamodel, its static semantics, execution semantics, a feature-rich text editor and a tight integration into the Eclipse platform. This shows, that for MDSD the well-investigated formalism of RAGs is a valuable alternative for specifying metamodel semantics. On the other hand, MDSD introduces interesting application areas and new challenges for RAG tools such as interactive usage scenarios, e.g., in graphical editors. In the future, we intend to improve the implementation of JastEMF to better support incremental RAG evaluation as presented in [11].

8. REFERENCES

[1] C. Bürger and S. Karol. Towards attribute grammars for metamodel semantics. Technical report, Technische Universität Dresden, 2010.

[2] T. Ekman. *Extensible Compiler Construction*. PhD thesis, University of Lund, 2006.

[3] T. Ekman and G. Hedin. The JastAdd system — modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[4] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPLAN '86*, pages 85–98. ACM, 1986.

[5] G. Hedin. Reference Attributed Grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[6] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *ECMDA-FA '09*, pages 114–129. Springer, 2009.

[7] D. E. Knuth. Semantics of Context-Free Languages. *Theory of Computing Systems*, 2(2):127–145, 1968.

[8] P. Muller, F. Fleurey, and J. Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713 of *LNCS*, pages 264–278. Springer, 2005.

[9] U. Nickel, J. Niere, and A. Zündorf. The FUJABA environment. In *ICSE '00*, pages 742–745. ACM, 2000.

[10] E. Nilsson-Nyman, G. Hedin, E. Magnusson, and T. Ekman. Declarative intraprocedural flow analysis of java source code. *Electron. Notes Theor. Comput. Sci.*, 238(5):155–171, 2009.

[11] T. Reps, T. Teitelbaum, and A. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM (TOPLAS)*, 5(3):449–477, 1983.

[12] B. Selic. The Theory and Practice of Modeling Language Design for Model-Based Software Engineering – A Personal Perspective. In *GTTSE '09*, LNCS. Springer, 2010. to appear.

[13] H. H. Vogt, D. Swierstra, and M. F. Kuiper. Higher Order Attribute Grammars. In *PLDI '89*, pages 131–145. ACM, 1989.