Generating Code and Documentation from Lightweight Abstract Grammars

Radu Grigore

University College Dublin

Abstract. Abstract grammars are used in papers when toy languages are described. For the development of language processing tools it is also useful to have a short and explicit description of the abstract grammar of the language being processed. This paper presents a little language for expressing lightweight abstract grammars, a little language for writing templates that refer to the information in the abstract grammar, and the tool to process these. The tool generates, for example, code for the Abstract Syntax Tree (AST) data structures, a table view and a graphical view of the abstract grammar.

1 Introduction and Related Work

Repetition is a sign that the task is better suited for a tool than a human. If the task is not completely repetitive the tool needs guidance, and one common way to give it is by writing a program in a little language [1,2] that the tool understands. The architecture of tools that process languages (such as compilers, interpreters, and translators) is fairly standard. The pipeline begins with a parser that builds an AST, which is then transformed and evaluated in stages until the output is produced. Because the architecture is standard, writing a language processing tool feels somewhat repetitive and, indeed, there are tools that help.

Commonly used compiler compilers in the Java world include Antlr [3], JavaCC [4], and SableCC [5]. They provide significant help for the early pipeline stages. The JastAdd system [6] aims to aid the construction of the whole pipeline and generates common stages such as symbol resolution and typechecking.

In Antlr the user writes a grammar that defines the concrete syntax. Antlr generates a parser from the grammar and from built-in code templates, the latter of which contain library calls to the Antlr API. Optionally, Antlr generates AST data structures, also from the grammar and other built-in code templates that refer to the Antlr API. The grammar can be annotated to say which information should end up in the AST and which not (for example, semicolons at the end of statements don't need a field in the AST).

Idea: The programmer can describe a separate abstract grammar and write templates. It is easier to work with an abstract grammar when it is described in a simple language and decoupled from the concrete grammar. Since the programmer writes templates, the generated code is easy to integrate with the rest of the system. Moreover, templates for generating documentation can be written

too. This combination proved successful in the development of the frontend of FreeBoogie [7], which will be a backend for software verifiers.

2 A Small Example

We want to write a tool that uses Bentley's survey language [1], say, to administer questionnaires interactively. Here is an example of the input we want to be able to process.

```
Q What is your political party?
  1 Democrat
  2 Republican
  3*Other
```

The first thing we do is to describe the abstract grammar.

```
Questionnaire = Item! item, Questionnaire tail;
Item = String! question, Boolean! custom, Choices choices;
Choices = String! choice, Choices tail;
```

The second thing we do is to write down a template.

```
\normal_classes{
\file{\ClassName.java}
public class \ClassName { ...
  public \ClassName(\members[, ]{\MemberType \memberName}) {
\members{
    \if_nonnull{assert \memberName != null; }{}this.\memberName = \memberName;}
  } ...
}}
```

One of the generated files is `Item.java`.

```
public class Item { ...
  public Item(String question, Boolean custom, Choices choices) {
    assert question != null; this.question = question;
    assert custom != null; this.custom = custom;
    this.choices = choices;
  } ...
}
```

We can customize the template to generate a HTML description of the abstract grammar, a TeX file, a `dot` file to visualize the AST, a `Makefile`, and so on.

3 The Abstract Grammar

The abstract grammar is a list of rules, of three possible types (inheritance, composition, and specification):

```
Class1 :> Class2, Class3;
Class2 = Type1 name1, Type2 name2;
Class: arbitrary specification text on one line
```

An identifier denotes a *class* if it appears on the left side of a rule. A class is *normal* if it appears on the left side of a composition rule; otherwise it is *abstract*. Types are arbitrary identifiers (including those that denote classes) or enumerations:

```
enum(EnumTypeName: VALUE1, VALUE2, VALUE3)
```

The type can be followed by the '!' sign, meaning that the current member is *nonnull*. A member is a *child* if its type is a class; otherwise it is a *primitive*. Specification rules are used, for example, if one wants to specify class invariants to be checked later with a tool such as ESC/Java [8,9].

To summarize the information that goes into an abstract grammar we can use an abstract grammar.

```
AbstractClass = String! name;
NormalClass = String! name, String! base, Members members, Enums enums, Invariants invs;
Members = String! type, String! name, boolean isPrimitive, boolean nonNull, Members tail;
Enums = String! name, EnumValues values, Enums tail;
EnumValues = String! name, EnumValues tail;
Invariants = String! inv, Invariants tail;
```

4 The Templates

Templates are arbitrary text files that contain macros, similar in form to TEX macros [10]. There are four types of macros:

- macros for directing the output of the template processor (`\file`)
- list macros (`\classes`, `\members`, `\enums`, `\values`, `\invariants`)
- data macros (`\class_name`, `\base_name`, `\member_type`, `\member_name`, `\enum_name`, `\value_name`, `\inv_text`)
- test macros (`\if_abstract`, `\if_primitive`, `\if_nonnull`, `\if_enum`)
- helper macros (`\abstract_classes`, `\normal_classes`, `\children`, `\primitives`)

The `\file`{*name*} can appear anywhere and it instructs the template processor to send the following characters to the file *name*. By default the output goes to `/dev/null`, which allows one to easily add comments at the beginning of a template.

The list macros take two parameters, the first of which is optional and defaults to the empty string: `\macro`*[sep]*{*text*}. The *text* is processed multiple times setting the *current* class (or member, or enumeration, or value, or invariant) to all values appearing in the abstract grammar. The current class, the current member, the current enumeration, the current value, and the current invariant for the current *environment*.

The data macros expand to the current class (or member name, or member type, etc.). Hence, they must be enclosed by the proper list macro. (It is safe to nest list macros of the same kind but only the current value corresponding of the innermost list can be accessed.) All data macros, except for `\inv_text`, come in four flavors, corresponding to different case conventions for the output (for example, `\class_name`, `\ClassName`, `\className`, and `\CLASS_NAME`).

The test macros receive two parameters, as in `\if_abstract`{*yes*}{*no*}, and select one of two pieces of the template to process based on the current environment.

Helper macros are defined in terms of others. For example, `\children`*[sep]*{*text*} is the same as `\members`*[sep]*{`\if_primitive`{}{*text*}}. These are currently built-in, but it is probably a good idea to allow the user to define such aliases.

5 Conclusion and Future Work

An explicit abstract grammar representation is helpful in presenting a language, can be used as a basis for code and documentation generation, and makes maintenance easier. In general, it is a good idea to use such an approach when the abstract grammar is evolving, has medium or large size, and when documentation must be generated alongside the code.

One possible evolution is to support lists in the abstract grammar explicitly. Then the templates can map them to a collection data structure. (Note that Java collections are mutable.) Another possible evolution is to allow users to define macro aliases. This would reduce repetition in the templates.

Acknowledgments. This work is funded by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the author's views and the Community is not liable for any use that may be made of the information therein.

The author would like to thank Joseph Kiniry, Mikoláš Janota, and Fintan Fairmichael for advice on how to improve the presentation.

References

1. Bentley, J.: Little languages. Communications of the ACM **29**(8) (1986) 711–721
2. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4) (December 2005) 316–344
3. Parr, T.: The ANTLR system homepage. `http://www.antlr.org/`
4. Sun Microsystems: The JavaCC homepage. `https://javacc.dev.java.net/`
5. Gagnon, E.M., Hendrie, L.J.: The SableCC homepage. `http://sablecc.org/`
6. Hedin, G., Magnusson, E.: JastAdd—an aspect-oriented compiler construction system. Science of Computer Programming **47**(1) (2003) 37–58
7. Grigore, R.: The FreeBoogie SVN repository. `https://mobius.ucd.ie/browser/src/freeboogie/trunk/FreeBoogie`
8. Cok, D., Kiniry, J.: ESC/Java2: Uniting ESC/Java and JML. Construction and Analysis of Safe, Secure and Interoperable Smart devices **3362** (2005) 108–128
9. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J., Stata, R.: Extended static checking for Java. ACM SIGPLAN Notices **37**(5) (2002) 234–245
10. Knuth, D.E.: The TeXbook. Addison–Wesley (1986)

Appendix for Evaluation Only: Demonstration

The demo will consist of a short introduction using slides followed by an interactive showcase of the tool. If desired, copies of the tool and the data files used can be provided in advance so that participants can replicate the steps themselves during the demo.

Introduction

The demonstration will begin with the slides attached to the end of this document. The purpose is to present the main definitions and to emphasize the design decisions with the aid of an example. For example, I will say that from the point of view of the design it is much more important what *types* of macros are available than what exactly are the macros in each category. Also, the hierarchical structure of the abstract grammar leads to the natural rules for nesting macros. Finally, the selection of helper macros is rather arbitrary—they were added as needed. In fact, I am considering adding a mechanism that allows the user to add such extra helpers and I will definitely pass the 'considering' stage if I shall need more than four. This illustrates the "keep it simple" principle of design. Another result of this principle is that the abstract grammar cannot explicitly express lists. That is because I prefer using functional data structures for the AST. The AST is usually huge and mutating things gets one in trouble in no-time. Of course, adding explicit support for lists is possible and, in fact, a previous implementation had such support. This should give you a flavor of the kind of things I will talk about while presenting the slides. (Please note that the slides are only a first draft.)

FreeBoogie as a Case Study

During this portion of the demo the projector will show whatever appears on the screen of my laptop. The plan is to first go through the FreeBoogie abstract grammar, then through the templates, and finally show how to perform a modification in the abstract grammar. The first two steps will showcase what kind of documents can be generated, and the last step will show how the use of this tool eases maintenance.

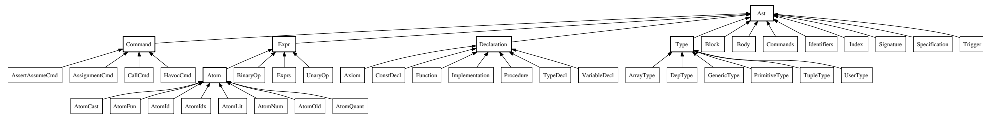
To familiarize the audience with the little language BoogiePL I will show a BoogiePL program. Then I will briefly explain the main features of the language while showing its abstract grammar, which takes about 70 lines with comments. This will illustrate how the explicit representation of the abstract grammar facilitates communication.

```
// This is the abstract grammar for FreeBoogie used to generate
// the AST. See also the template (*.tpl) files.

// The program is a (functional) list of (global) declarations
Declaration :> TypeDecl, ConstDecl, Function, Axiom,
               VariableDecl, Procedure, Implementation;
```

```
TypeDecl =     String! name, Declaration tail;
ConstDecl=     String! id, Type! type, Declaration tail;
Function =     Signature! sig, Declaration tail;
Axiom =        Expr! expr, Declaration tail;
VariableDecl = String name, Type! type, Declaration tail;
Procedure =    Signature! sig, Specification spec, Declaration tail;
Implementation = Signature! sig, Body! body, Declaration tail;


// Types and constants
Type :> PrimitiveType, UserType, ArrayType,
        GenericType, DepType, TupleType;
ArrayType = Type! rowType, Type colType, Type! elemType;
PrimitiveType = enum(Ptype: BOOL, INT, REF, NAME, ANY, ERROR) ptype;
GenericType = Type! param, Type! type;
DepType = Type! type, Expr! pred;
UserType = String! name;
TupleType = Type! type, TupleType tail;

// Signatures are used to represent uninterpreted functions
Signature = String! name, Declaration args, Declaration results;

// Procedures and implementations
Specification = enum(SpecType: REQUIRES, MODIFIES, ENSURES) type,
                Expr! expr, boolean free, Specification tail;
Body = Declaration vars, Block blocks;
Block = String! name, Commands cmds, Identifiers succ, Block tail;
  // succ == null means that we return

// Commands
Commands = Command! cmd, Commands tail;
Command :> AssignmentCmd, AssertAssumeCmd, HavocCmd, CallCmd;
AssignmentCmd = Expr! lhs, Expr! rhs;
AssertAssumeCmd = enum(CmdType: ASSERT, ASSUME) type, Expr! expr;
HavocCmd = AtomId! id;
CallCmd = String! procedure, Identifiers results, Exprs args;

// Expressions
Expr :> BinaryOp, UnaryOp, Atom, Exprs;
BinaryOp = enum(Op: PLUS, MINUS, MUL, DIV, MOD, EQ, NEQ, LT, LE,
                    GE, GT, SUBTYPE, EQUIV, IMPLIES, AND, OR) op,
           Expr! left, Expr! right;
UnaryOp = enum(Op: MINUS, NOT) op, Expr! e;
Atom :> AtomLit, AtomNum, AtomId, AtomFun, AtomOld,
        AtomCast, AtomQuant, AtomIdx;
AtomLit = enum(AtomType: FALSE, TRUE, NULL) val;
AtomNum = BigInteger val;
AtomId = String! id;
AtomFun = String! function, Exprs args;
AtomOld = Expr! e;
AtomCast = Expr! e, Type! type;
AtomQuant = enum(QuantType: EXISTS, FORALL) quant, Declaration! vars,
            Trigger trig, Expr! e;
AtomIdx = Atom! atom, Index! idx;
Index = Expr! a, Expr b;

// Some simple lists
Trigger = String label, Exprs! exprs, Trigger tail;
Identifiers = AtomId! id, Identifiers tail;
Exprs = Expr! expr, Exprs tail;

// Some class invariants
Specification: type==Type.MODIFIES ==> \typeof(expr)==AtomId
```

In FreeBoogie, templates are used to generate:

– documentation
– AST data structures (Java)
– the Visitor base classes (Java)
– some help for the build system



The documentation template is presented first. It generates a HTML page with a table view and a graphic view of the abstract grammar that should reinforce the audience's grasp of BoogiePL. The AST data structures are rather simple; we shall observe how the Visitor pattern is supported. Next we move on to the Visitor pattern. Two base classes, `Evaluator` and `Transformer`, are generated automatically. The implementation is a variation of the standard Visitor pattern that makes it easy to deal with the immutable nature of the AST data structures. (For example, path copying is done in the class `Transformer` and the writer of concrete visitors doesn't need to worry about it.) Related to visitors, we shall see how some boilerplate code is generated that helps implement new visitors. Finally, a list of the generated files is created, so that the build script need not be changed when the abstract grammar is modified.

The last part involves modifying the program and show what this entails. One possibility is to show how we can eliminate the `TupleType` from the grammar. Of course, its function should be delegated somehow and at least the typechecker needs to be updated. We'll see, however, that the modifications are much less repetitive and error prone than they would be without the use of this tool.

I will also try to cover some things that were mentioned only in passing in the paper or not at all (such as how the inheritance rules work, how to integrate with a parser generator, and more related work, even unpublished).

The Abstract Grammar — little language

- a sequence of rules: inheritance, composition, specification
  - `Class :> Class1, Class2;`
  - `Class = Type1 name1, Type2 name2;`
  - `Class: one line of text`
- An identifier that appears on the left of a rule is called a class.
- A class is normal if it appears on the left of a composition rule; otherwise it is abstract.
- A member is a child if its type is a class; otherwise it is a primitive.
- A type can be either an identifier or might have the following form to denote an enumeration:
  - `enum(EnumName: VALUE1, VALUE2, VALUE3)`
- A member is nonnull if its type is followed by '!'.

The Abstract Grammar — example

```
Decl :> Axiom, Procedure;
Axiom = Expr! expr, Decl tail; // functional list
Procedure = Signature! sig, Spec spec, Decl tail;
Expr :> BinaryOp, Atom;
BinaryOp = enum(Op: PLUS, MINUS, EQ, NEQ) op,
           Expr! left, Expr! right;
Atom :> AtomLit, AtomNum, AtomId;
AtomLit = enum(AtomType: FALSE, TRUE, NULL) val;
AtomNum = BigInteger val;
AtomId = String! id;
Spec = enum(SpecType: REQUIRES, MODIFIES) type,
       Expr! expr, boolean free, Spec tail;
Spec: type==Type.MODIFIES ==> \typeof(expr)==AtomId
```

The Abstract Grammar — review

The information that goes into the abstract grammar is:

```
AbstractClass = String! name;
NormalClass =
  String! name, String! base, Members members,
  Enums enums, Invariants invs;
Members =
  String! type, String! name, boolean isPrimitive,
  boolean nonNull, Members tail;
Enums = String! name, EnumValues values, Enums tail;
EnumValues = String! name, EnumValues tail;
Invariants = String! inv, Invariants tail;
```

(Yes, that is recursive.)

Templates — little language

- direct output: `\file`{*name*}
- list macros: `\classes`*[sep]*{*text*}, `\members`*[sep]*{*text*}, `\enums`*[sep]*{*text*}, `\values`*[sep]*{*text*}, `\invariants`*[sep]*{*text*}
- data macros: `\class_name`, `\BaseName`, `\memberType`, `\MEMBER_NAME`, `\enum_name`, `\ValueName`, `\inv_text`
- test macros: `\if_abstract`{*yes*}{*no*}, `\if_primitive`{*yes*}{*no*}, `\if_nonnull`{*yes*}{*no*}, `\if_enum`{*yes*}{*no*}
- helper list macros: `\abstract_classes`*[sep]*{*text*}, `\normal_classes`*[sep]*{*text*}, `\children`*[sep]*{*text*}, `\primitives`*[sep]*{*text*}

Template — example

```
This generates a graphical view of the abstract grammar.
\file{hierarchy.dot}
digraph hierarchy {
  rankdir=RL;
  node [shape=box];
  \classes{"\ClassName" -> "\BaseName";
}
  "Ast" [style=bold];
  \abstract_classes{"\ClassName" [style=bold];
}
}
```

Template — image

The command line
`fbrun freeboogie.ast.gen.Main -b Ast fb.ag dot.tpl`
produces a dot file that renders as seen.