A Translator from BML annotated Java Bytecode to BoogiePL

Ovidio José Mallo

Master Project Report

Software Component Technology Group
Department of Computer Science
ETH Zurich

http://sct.inf.ethz.ch/

March 2007

Supervised by:
Hermann Lehner
Prof. Dr. Peter Müller

Software Component Technology Group
inf | Informatik
Computer Science

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Contents

Chapter 1

Introduction

Chapter 2

BoogiePL Abstract Syntax Tree

In the following, we will give a brief description of the abstract syntax tree used to represent a BoogiePL program. To that end, we use a BNF-like grammar definition which describes the abstract syntax of a BoogiePL program. Note that while similar grammars are also given elsewhere [2, 1], the main purpose of the definition provided here is to clearly specify the features of BoogiePL supported by our implementation. In addition, the grammar definition shall serve as a precise and compact description of the concrete implementation of the abstract syntax tree without referring to the actual code. This is achieved since every non-terminal symbol used in the grammar directly corresponds to an equally-named[1] class in the actual implementation. More precisely, the grammar is to be interpreted as follows:

- We use the meta-level symbols *, $^+$, and $^?$ to denote a sequence, a non-empty sequence, and an optional element, respectively. The angle brackets $\langle\cdot\rangle$ are used for grouping and the standard operator | separates different alternatives in a single production.
- All the terminal symbols denoting either keywords of the BoogiePL language or punctuation symbols are written in a **bold** font.
- Non-terminal symbols denoting *abstract* classes in the implementation of the abstract syntax tree are written in an *italic* font. Every production for such an abstract class consists of a set of alternatives representing all the subclasses of the abstract class.
- Non-terminal symbols denoting *concrete* classes in the implementation of the abstract syntax tree are written in a normal font. Every production for such a concrete class may consist of a set of terminal and non-terminal symbols, where the latter fully describe the corresponding concrete class.

2.1 Programs and declarations

A BoogiePL program consists of a set of declarations:

Program	::=	*Declaration**
Declaration	::=	VariableDeclaration
	|	ConstantDeclaration
	|	TypeDeclaration
	|	Function
	|	Axiom
	|	Procedure
	|	Implementation

[1] up to the prefix BPL which is prepended to every node class of the abstract syntax tree in the actual implementation while omitted in the grammar and in the following discussion

Variable and constant declarations are treated uniformally in that they both introduce a set of `Variable`s:

VariableDeclaration ::= **var** Variable$^+$ **;**
ConstantDeclaration ::= **const** Variable$^+$ **;**

A type declaration can be used in BoogiePL to declare a set of user-defined types:

TypeDeclaration ::= **type** String$^+$ **;**

Since such a declaration merely defines the names of the introduced types but does not contain any further information, we represent the individual types as normal `String`s instead of introducing a special purpose class for them.

Every function declaration defines exactly one function symbol (which, however, may have several names associated to it). Therefore, we do not make any special distinction between the declaration and the actual function symbol which can then be referenced in expressions:

Function ::= **function** String$^+$**(**FunctionParameter***)**
returns (FunctionParameter**)**
FunctionParameter ::= ⟨String **:**⟩$^?$ *Type*

In- and out-parameters to a function are represented by the special class `FunctionParameter` which simply represents an optionally named type.

An axiom is fully defined by an expression of type **bool** which specifies a constraint on the symbolic constants and functions:

Axiom ::= **axiom** *Expression* **;**

Procedures and implementations mainly differ from each other in that a procedure may have a specification attached to it:

Procedure ::= **procedure** String**(**Variable***)** ⟨**returns (**Variable***)**⟩$^?$
Specification$^?$ ImplementationBody$^?$
Implementation ::= **implementation** String**(**Variable***)** ⟨**returns (**Variable***)**⟩$^?$
ImplementationBody

The in- and out-parameters are in both cases represented by a set of `Variable`s. A procedure's specification consists of a number of **requires**, **modifies**, and **ensures** clauses, where a new feature recently added to BoogiePL allows requires and ensures clauses to be marked as **free** in which case the corresponding conditions need not be verified by Boogie but instead can just be assumed. A modifies clause is defined by an optional set of `VariableExpression`s (discussed later) which represent identifiers referencing variables:

Specification ::= *SpecificationClause*$^+$
SpecificationClause ::= RequiresClause
| ModifiesClause
| EnsuresClause
RequiresClause ::= **free**$^?$ **requires** *Expression* **;**
ModifiesClause ::= **modifies** VariableExpression* **;**
EnsuresClause ::= **free**$^?$ **ensures** *Expression* **;**

Finally, the implementation body belonging to a procedure or an implementation consists of an optional set of variable declarations followed by a number of basic blocks:

ImplementationBody ::= VariableDeclaration* BasicBlock$^+$

2.2 Types

Types are mainly used in the declarations of variables and function parameters but also in cast expressions. Currently, four kinds of types are supported in BoogiePL:

Type ::= BuiltInType
| TypeName
| ArrayType
| ParameterizedType

The built-in types are represented by the single class `BuiltInType`. In the actual implementation, we use a typesafe enumeration pattern to represent the individual types:

BuiltInType ::= **bool** | **int** | **ref** | **name** | **any**

User-defined types are represented by the `TypeName` class and are fully determined by their name which corresponds to a BoogiePL identifier:

TypeName ::= String

Array types are defined by a set of index types used to access the array together with the actual element type:

ArrayType ::= [*Type*$^+$] *Type*

Note that even though our grammar allows array types to have an arbitrary number of index types, BoogiePL only supports up to two-dimensional arrays.

A new feature recently added to BoogiePL is the support for what we call *parameterized types* which allows to parameterize some type by another type:

ParameterizedType ::= <*Type*> *Type*

In order to see what parameterized types can be used for, let us assume we want to model a heap in BoogiePL by using a two-dimensional array which is indexed by a **ref** type denoting an object and a **name** type representing an object's field (as is e.g. done in Spec#). Since the heap may contain objects of different types, the array's element type must be declared to be of type **any**. Therefore, whenever we extract an element from such an array, we must usually insert an appropriate cast to the actual type of the field. This is illustrated on the left hand side of the following listing:

```
var Heap: [ref, name]any;
const C.f: name;
var o: ref, i: int;
Heap[o, C.f] := 3;
i := cast(Heap[o, C.f], int);
```

```
var Heap: [ref, <t>name]t;
const C.f: <int>name;
var o: ref, i: int;
Heap[o, C.f] := 3;
i := Heap[o, C.f];
```

On the right hand side, by contrast, we see how one might achieve the same result by taking advantage of parameterized types: we see that the **name** type used to index the heap array is now parameterized by a type parameter `t` which is also used as the array's element type. As can be seen in the declaration of the constant `C.f` denoting the field being accessed, the actual type to be inserted for the type parameter `t` can then be specified individually for every **name** constant. If we now use the constant `C.f` to access an element of the array, the type **int** is automatically substituted for the type parameter `t` and no explicit cast is required anymore, thus improving the readability of the code. In addition – and more importantly – the fact that the field modeled by the constant `C.f` is of type **int** can be made explicit in the constant's declaration by using parameterized types.

2.3 Variables

We use the single class `Variable` to represent variables and constants, in- and out-parameters of procedures and implementations, as well as expression bound variables introduced by quantification expressions (discussed later). Therefore, a `Variable` covers exactly those elements which may be referenced by a BoogiePL identifier in expressions and procedure specifications. Our grammar defines a variable as follows:

Variable ::= String **:** *Type* ⟨**where** *Expression*⟩$^?$

As we can see, a variable can have a so-called *where clause* associated to it. The expression of such a where clause must be of type **bool** and it specifies a unary constraint on the variable's value. Where clauses are a convenience construct which allows to specify some properties of a variable's value along with its declaration instead of scattering that information over different points in the program. In addition, whenever such a variable is havoc'ed, its value becomes not totally arbitrary but is still constrained by the expression provided in the where clause. However, note that where clauses may only be used in variable declarations and for parameters of procedures and implementations but not in constant declarations or for expression bound variables. Note also that the concept of a variable as used in our context does not completely correspond to the notion of variables described in [2] where constant symbols are not considered to be variables.

2.4 Basic blocks and commands

The body of a procedure or implementation contains a set of basic blocks, where each of them consists of a label and a sequence of commands, followed by a single transfer command:

BasicBlock ::= String**:** *Command** *TransferCommand*

The set of commands and transfer commands should be self-explanatory and, thus, they are not discussed further at this point:

Command	::=	AssertCommand
	|	AssertCommand
	|	HavocCommand
	|	AssignmentCommand
	|	CallCommand
AssertCommand	::=	**assert** *Expression* **;**
AssumeCommand	::=	**assume** *Expression* **;**
HavocCommand	::=	**havoc** VariableExpression$^+$ **;**
AssignmentCommand	::=	*Expression* **:=** *Expression* **;**
CallCommand	::=	**call** ⟨VariableExpression$^+$ **:=**⟩$^?$ String**(***Expression****)** **;**
TransferCommand	::=	GotoCommand
	|	ReturnCommand
GotoCommand	::=	**goto** String$^+$ **;**
ReturnCommand	::=	**return;**

2.5 Expressions

All the classes used to represent the different kinds of expressions are given in the following grammar productions:

Expression	::=	VariableExpression
	|	*BinaryExpression*
	|	*UnaryExpression*

	|	QuantifierExpression
	|	*Literal*
	|	ArrayExpression
	|	CastExpression
	|	FunctionApplication
	|	OldExpression
BinaryExpression	::=	BinaryArithmeticExpression
	|	BinaryLogicalExpression
	|	EqualityExpression
	|	PartialOrderExpression
	|	RelationalExpression
Literal	::=	BoolLiteral
	|	NullLiteral
	|	IntLiteral
UnaryExpression	::=	LogicalNotExpression
	|	UnaryMinusExpression

A `VariableExpression` is the most common expression and simply represents a BoogiePL identifier referencing a variable (as modeled by the already discussed `Variable` class):

$$\text{VariableExpression} \quad ::= \quad \text{String}$$

The set of arithmetic and first-order logical expressions supported by BoogiePL are defined by the following grammar productions:

$$\begin{aligned}
\text{BinaryArithmeticExpression} &::= \textit{Expression} \ \langle + \mid - \mid * \mid / \mid \% \rangle \ \textit{Expression} \\
\text{BinaryLogicalExpression} &::= \textit{Expression} \ \langle \wedge \mid \vee \mid \Rightarrow \mid \Leftrightarrow \rangle \ \textit{Expression} \\
\text{EqualityExpression} &::= \textit{Expression} \ \langle = \mid \neq \rangle \ \textit{Expression} \\
\text{PartialOrderExpression} &::= \textit{Expression} <: \textit{Expression} \\
\text{EqualityExpression} &::= \textit{Expression} \ \langle < \mid > \mid \leq \mid \geq \rangle \ \textit{Expression} \\
\text{LogicalNotExpression} &::= \neg \ \textit{Expression} \\
\text{UnaryMinusExpression} &::= - \ \textit{Expression} \\
\text{QuantifierExpression} &::= (\langle \forall \mid \exists \rangle \ \text{Variable}^* \ \text{Trigger}^* \bullet \textit{Expression})
\end{aligned}$$

As we can see, in order to reduce the number of nodes in the abstract syntax tree, not every expression is represented by its own class but, instead, we group operations according to their operands and types. In the actual implementation, a special enumeration type denoting the individual operators is introduced for every group of operations in order to distinguish among them.

Triggers are a special feature recently added to BoogiePL which allows to pass information to an underlying theorem prover as of how to instantiate universal quantifiers, as described in [1]. As specified above, triggers can be used in quantification expressions and consist of a non-empty sequence of expressions:

$$\text{Trigger} \quad ::= \quad \{ \ \textit{Expression}^+ \ \}$$

The boolean, reference, and integer literals supported by BoogiePL are represented as follows:

$$\begin{aligned}
\text{BoolLiteral} &::= \textbf{true} \mid \textbf{false} \\
\text{NullLiteral} &::= \textbf{null} \\
\text{IntLiteral} &::= \cdots \mid \textbf{-1} \mid \textbf{0} \mid \textbf{1} \mid \cdots
\end{aligned}$$

The remaining expressions, which should be self-explanatory and, thus, are not further discussed at this point, are the following:

$$\text{ArrayExpression} \quad ::= \quad \textit{Expression}[\textit{Expression}^+]$$

CastExpression	::=	**cast**(*Expression*, *Type*)
FunctionApplication	::=	String(*Expression**)
OldExpression	::=	**old**(*Expression*)

2.6 Implementation notes

The actual implementation of the abstract syntax tree is fully described by the above discussed grammar. The individual nodes of the tree are implemented as simple classes which do not contain any specific functionality. Instead, operations on the abstract syntax tree are implemented using the visitor pattern. In particular, the implementation of the abstract syntax tree is completely self-contained in that the individual nodes of the tree only reference other nodes and the abstract visitor provided for the tree.

As a convenience, we provide the ability to decorate some of the nodes with node-specific data. As an example, the `Variable` referenced by a `VariableExpression` can be directly stored in the latter and the types to which expressions evaluate can be set on the corresponding `Expression` object. This allows to conveniently store information which might be used frequently in the nodes themselves. Note, however, that even though this information is kept in the nodes, the information itself is never computed by the nodes but is always set from the outside (e.g. by an appropriate visitor performing a semantic analysis on the tree).

Bibliography

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. FMCO 2005. Available from http://research.microsoft.com/~leino/papers/krml160.pdf, 2006.

[2] Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.