This is Boogie 2

K. Rustan M. Leino

Microsoft Research, Redmond, WA, USA
leino@microsoft.com

Manuscript KRML 178, working draft 24 June 2008.

Abstract. Boogie is an intermediate verification language, designed to make the prescription of verification conditions natural and convenient. It serves as a common intermediate representation for static program verifiers of various source languages, and it abstracts over the interfaces to various theorem provers. Boogie can also be used as a shared input and output format for techniques like abstract interpretation and predicate abstraction. As a language, Boogie has both mathematical and imperative components. The imperative components of Boogie specify sets of execution traces, the states of which are described and constrained by the mathematical components. The language includes features like parametric polymorphism, partial orders, logical quantifications, nondeterminism, total expressions, partial statements, and flexible control flow. The Boogie language was previously known as BoogiePL. This paper is a reference manual for Boogie version 2.

Hello, my name is Rustan Leino. I'm a computer scientist. I write a lot of papers. That little IPL article on subscripts in technical presentations? That was mine. In 1992, I went up to Palo Alto, California to a research lab called the Systems Research Center. Don't look for it; it's not there anymore. But that summer, I saw a tool that for me redefined the word "light-weight formal methods". I remember being knocked out by its... its potential, its raw power– and its automation. That tool was Modula-3's now-legendary Extended Static Checker. Sixteen years and several verification tools later, theorem-prover based software verification is still trying to reach that potential. And it's earned a distinguished place in computer science history as one the trade's most expertise-demanding tools. So in the late fall of 2006, when I heard that the intermediate verification language Boogie was releasing a new version called "Smell the Polymorphism", and was planning its first major publications in almost three years to promote that version, well needless to say I jumped at the chance to write the documentation—the, if you will, "rockumentation"—that you're about to read. I wanted to capture the... the uses, the sounds... the smells of a hard-working verification language, on the road. And I got that; I got more... a lot more. But hey, enough of my yakkin'. Whaddaya say? Let's Boogie!

0 Introduction

A standard technique in program verification is to transform a given program into a set of *verification conditions*, logical formulas whose validity implies that the program

satisfies the correctness properties under consideration. The verification conditions are then processed with the help of a theorem prover, where a successful proof attempt shows the correctness of the program, and a failed proof attempt may give an indication of a possible error in the program.

Experience with some automatic program verifiers [6, 4, 1] suggests that the complex task of generating verification conditions for modern programming languages can be managed by separating the task into two steps: a transformation of the program and its proof obligations into an intermediate language (an intermediate representation that is structured more like a program than a formula), and then a transformation of that intermediate-language program into logical formulas. The first of these steps encodes the semantics of the source program's constructs in terms of primitive program constructs, records as assumptions properties that are guaranteed in any execution of the source program, and prescribes what correctness means (that is, what conditions need to hold where in order for the program to be considered correct).

This paper is a reference manual for Boogie, an intermediate language designed to accommodate the encoding of verification conditions for imperative, object-oriented programs. Previously known as BoogiePL, the Boogie language is currently used and checked by the program verifier also called Boogie [1]. At the time of this writing, translations into Boogie exist or are under way for several languages: Spec# [1], C [2, 10], Dafny, Java bytecode with BML, and Eiffel. The paper describes the syntax, type checking, and semantics of Boogie, as well as a rationale for its design and examples of how to encode proof obligations and assumptions in the language.

The next section gives an overview of Boogie 2. The bulk of the remaining sections detail the language constructs, types, expressions, and statements. The last sections give additional examples and related work.

1 Overview

At the top level, Boogie features seven kinds of declarations. The mathematical constructs introduce types, constants, functions, and axioms. The imperative constructs introduce global variables, procedure declarations, and procedure implementations.

Type declarations introduce type constructors. For example,

$$\textbf{type}\ \mathit{Wicket};$$

declares a type (more precisely, a nullary type constructor) intended to represent wickets. Symbolic constants are introduced by *constant declarations*, like

$$\textbf{const}\ w\colon \mathit{Wicket};$$

which says that w is some fixed (but unspecified) value of type Wicket. *Function declarations* introduce mathematical functions. For example,

$$\textbf{function}\ \mathit{age}(\mathit{Wicket})\ \textbf{returns}\ (\textbf{int});$$

declares a function intended to return the age of wickets. Properties of constants and functions are postulated by *axiom declarations*. For example,

$$\textbf{axiom}\ \mathit{age}(w) == 7;$$

says that age returns 7 for w.

The program state is made up of mutable variables. Global *variable declarations* introduce the state on which all procedure operate. For example,

var $favorite$: $Wicket$;

introduces a variable for holding the current favorite wicket. A *procedure declaration* gives a name to a set of execution traces, which are specified by pre- and postconditions. For example,

procedure $NewFavorite(n$: $Wicket)$;
  modifies $favorite$;
  ensures $favorite == n$;

Finally, an *implementation declaration* spells out a set of execution traces by giving a body of code. The implementation is correct if and only if its set of traces is a subset of the traces specified by the corresponding procedure. For example,

implementation $NewFavorite(n$: $Wicket)$
{
  $favorite := n$;
}

gives a (correct) implementation of procedure $NewFavorite$.

The grammars shown in this paper use | to separate alternatives. The superscript $^?$ indicates that the preceding component is optional, the superscripts * and $^+$ indicate a repetition of 0-or-more and 1-or-more, respectively, components juxtaposed, and $,^*$ and $,^+$ indicate 0-or-more and 1-or-more components separated by commas. Boldface words indicate keywords, and non-| non-superscript symbols (like semi-colons and parentheses) stand for themselves.

A Boogie program has the following form:

$$\begin{array}{rcl} \mathit{Program} & ::= & \mathit{Decl}^* \\ \mathit{Decl} & ::= & \mathit{TypeDecl} \mid \mathit{ConstantDecl} \mid \mathit{FunctionDecl} \mid \mathit{AxiomDecl} \\ & \mid & \mathit{VarDecl} \mid \mathit{ProcedureDecl} \mid \mathit{ImplementationDecl} \end{array}$$

The order of the declarations in a program is immaterial.

Identifiers, indicated by the grammar component Id, consist of characters drawn from a large set. This set consists of letters (including non-English letters from the Unicode alphabet), digits (but an identifier may not being with a digit), and

_	underscore
.	dot
$	dollar sign
#	hash sign
′	single quote (prime)
`	back quote
∼	tilde
^	caret
\	backslash
?	question mark

The identifiers declared in a Boogie program fall into five independent name spaces: types, functions, constants plus variables, procedures, and attributes. The name space for types contains global type names (type constructors and type synonyms) and scoped type variables (appearing in map types and in quantification expressions). The global type names must be distinct from each other and the type variables in any scope must be distinct from each other, but scoped type variables are allowed to shadow global type names. Similarly, the name space for constants plus variables contains global names (constants and global variables) and scoped names (formal parameters, local variables, and quantified variables). The global names must be distinct from each other and the scoped names in any scope must be distinct from each other, but scoped variables are allowed to shadow global names.

2 Types

In addition to built-in types, Boogie allows user-defined type declarations:

$$\mathit{TypeDecl} \quad ::= \quad \mathit{TypeConstructor} \mid \mathit{TypeSynonym}$$

This section describes type constructors, then built-in types, and finally type synonyms.

2.0 Type constructors

Type constructors are declared according to the following grammar:

$$\mathit{TypeConstructor} \quad ::= \quad \mathbf{type}\ \mathit{Attribute}^*\ \mathbf{finite}^?\ \mathit{Id}\ \mathit{Id}^*\ ;$$

Type declarations, along with all other top-level declarations and some other constructs, can be decorated with a set of *attributes*; these are described in Section 11.

The declaration above defines a *type constructor* named by the first Id. The type constructors in a program must have distinct names, and these names must also be distinct from any type synonym. The number of remaining Id's specify the number of type arguments that the type constructor takes, but the particular Id's used to indicate the number of these arguments is not relevant. In particular, the type-argument Id's can contain duplicates; for example, they may all be the identifier _.

As an example,

$$\mathbf{type}\ \mathit{Barrel}\ \alpha;$$

declares a unary type constructor intended to represent barrels for some contents. For example, $\mathit{Barrel}\ \mathbf{int}$ and $\mathit{Barrel}\ \mathit{Wicket}$ are two instantiations of the Barrel type constructor, representing barrels of integers and barrels of wickets, respectively.

Each instantiation of a type constructor gives rise to a type. The cardinality of each type is non-zero; that is, each type represents a nonempty set of individuals. Unless

the type constructor is declared with **finite**, the type has an infinite number of individuals. A type constructed by a **finite** type constructor may have a finite number of individuals. For example, a closed enumeration can be declared as a finite type:

$$\begin{array}{l}\textbf{type finite } RGBColor;\\ \textbf{const unique } red\colon RGBColor;\\ \textbf{const unique } green\colon RGBColor;\\ \textbf{const unique } blue\colon RGBColor;\\ \textbf{axiom } (\,\forall\, ce\colon RGBColor \bullet\ ce == red \,\vee\, ce == green \,\vee\, ce == blue\,);\end{array} \tag{0}$$

A subtle, but important, point in this example is that without the **finite** keyword, $RGBColor$ would have an infinite number of values, in which case the axiom would be tantamount to **false**. Note that it is the declaration of the type constructor, not its type arguments, that determines whether or not it yields **finite** types.

2.1 Built-in types

A *type* is either a *primitive type*, an instantiated type constructor or type synonym, or a possibly polymorphic *map type*. In addition, universally quantified type identifiers can denote types.

$Type$	::=	$TypeAtom \mid MapType$	
	|	$Id\ TypeCtorArgs^{?}$	type identifier, or type constructor with type arguments
$TypeAtom$	::=	$\mathbf{bool} \mid \mathbf{int}$	primitive types
	|	$\mathbf{bv}0 \mid \mathbf{bv}1 \mid \mathbf{bv}2 \mid \cdots$	bit-vector types
	|	$(\ Type\ )$	
$MapType$	::=	$TypeArgs^{?}\ [\ Type,^{+}\ ]\ Type$	
$TypeArgs$	::=	$\langle\ Id,^{+}\ \rangle$	
	|	$<\ Id,^{+}\ >$	ASCII syntax for the same
$TypeCtorArgs$	::=	$TypeAtom\ TypeCtorArgs^{?}$	
	|	$Id\ TypeCtorArgs^{?}$	
	|	$MapType$	

The primitive type **bool** denotes the booleans and **int** denotes the mathematical integers. The various **bv** types represent bit vectors consisting of the indicated number of bits. The semantics of types is that no individual belongs to more than one type; that is, different types represent disjoint sets of individuals.

A (possibly polymorphic) map type denotes *maps*, aka update-able maps, aka non-rigid functions, aka heterogeneous arrays. Syntactically, the domain types are listed within square brackets, followed by the range type. For example, each individual of $[RGBColor]\ \mathbf{int}$ maps RGB-color individuals to integers, and each individual of the type $[\mathbf{int}, \mathbf{int}]\ \mathbf{bool}$ is a two-dimensional array of booleans. Domain types need not be finite; for example, $[Wicket]\mathbf{bool}$ represents maps from all wickets to booleans and $[\mathbf{int}]\,Wicket$ represents maps from all mathematical integers to wickets.

A map type also indicates a number of universally quantified type identifiers, syntactically listed first within angle brackets. The order of the bound type identifiers is

not important, but they must all be distinct from each other and from all bound type identifiers in enclosing types. If this list is empty, it is omitted and the map type is said to be *monomorphic*. If the list is nonempty, the map type is said to be *polymorphic*. For example, in the following, m maps wicket barrels to wickets whereas n maps any kind of barrel to individuals of that barrel's contents type:

```
const m: [Barrel Wicket] Wicket;
const n: ⟨α⟩[Barrel α] α;
```

So, if bi has type $Barrel\ \mathbf{int}$, then the map-selection expression $n[bi]$ denotes an integer, and if bw has type $Barrel\ Wicket$, then $n[bw]$ denotes a wicket. Boogie supports Unicode input and allows ASCII synonyms, like $<$ for $\langle$ and $>$ for $\rangle$.

There is a restriction on polymorphic map types. Each bound type variable must be mentioned somewhere in the map type's domain types. For example, $\langle\alpha\rangle[\mathbf{int}]\alpha$ is not allowed, because the domain type $\mathbf{int}$ does not mention α. Although it is not actually necessary for defining type checking for expressions per se—one could let the enclosing context further constrain the bound type variables, in the usual way—this restriction means that every map-selection expression has a unique type, regardless of the context where the map selection appears.

Modulo this restriction, Boogie's type system thus supports higher-rank types, in the sense that a map can take another map as a domain argument. In some languages, universal types $\langle\alpha\rangle\ldots$ are considered for any types, but in Boogie they are only used with map types, which is therefore reflected in the Boogie syntax.

Note that the parsing of type-constructor arguments and type-synonym arguments is right associative (see grammar production *TypeCtorArgs*). Stated differently, the arguments of a type constructor (or type synonym) are parsed as long as possible or until a map type is encountered. Parentheses can be used to override the precedence levels, as usual. Here are several examples:

```
type C α β;
const a: C Wicket Wicket;
const b: Barrel Barrel Wicket;        // error (here, the first Barrel is given
                                      // two arguments instead of one)
const c: Barrel (Barrel Wicket);
const d: Barrel [int] Barrel Wicket;  // same as Barrel ([int] (Barrel Wicket))
const e: C Wicket Barrel int;         // error (C expects 2 arguments, not 3)
const f: C Wicket (Barrel int);
const g: C Wicket [int]Barrel int;    // same as C Wicket ([int] (Barrel int))
const h: C [int] int Wicket;          // parse error (extraneous Wicket)
const i: C [int] Wicket Wicket;       // error (first Wicket expects 0 arguments)
const j: C ([int] Wicket) Wicket;
```

2.2 Type synonyms

Type synonyms are declared as follows:

TypeSynonym ::= **type** *Attribute** *Id Id** = *Type* ;

This declaration introduces a *type synonym* whose name is the first Id, whose formal type arguments are given by the remaining Id's, and whose definition is given by $Type$. The type synonyms in a program must must have distinct names, and these names must also be distinct from any type constructor. The type arguments must all be distinct. The right-hand side $Type$ can mention the type arguments (and other type synonyms and type constructors) and must resolve to a properly formed type. For example, in:

$\mathbf{type}\ MySynonym\ \alpha = \mathbf{int};$
$\mathbf{type}\ ComplicatedInt = MySynonym\ (MySynonym\ \mathbf{bool});$
$\mathbf{type}\ Bogus = MySynonym\ MySynonym;$ // error

the declaration of the unary type synonym $MySynonym$ is legal (it is allowed to ignore its argument, α, in the definition) and so is the declaration of $ComplicatedInt$, but the declaration of $Bogus$ is not legal, because the second occurrence of $MySynonym$ is used without an argument type.

A type synonym is simply an abbreviation for the given type; any use of it, which syntactically looks like the use of a type constructor, is simply replaced by the right-hand side $Type$ in which:

- all bound type arguments are renamed to avoid name capture, and
- the type arguments to the type synonym are replaced by the types provided in the use.

For example,

$\mathbf{type}\ MultiSet\ \alpha = [\alpha]\mathbf{int};$

defines, for any type α, $MultiSet\ \alpha$ as a synonym for the type $[\alpha]\mathbf{int}$ that represents maps from α to integers. Semantically, there is no difference between the use of a type synonym (like $MultiSet\ Wicket$) and the type to which it expands (here, $[Wicket]\mathbf{int}$).

As another example, given:

$\mathbf{type}\ S\ \alpha\ \beta = \langle\gamma\rangle[\beta, \gamma]\mathbf{int};$

(where the right-hand side is a polymorphic map type with bound type variable γ, as introduced below), the use:

$S\ \mathbf{bool}\ (S\ Wicket\ \langle\gamma\rangle[\gamma]\gamma)$

is semantically equivalent to:

$\langle\epsilon\rangle[\ \langle\delta\rangle[\ \langle\gamma\rangle[\gamma]\gamma\ ,\ \delta\ ]\ \mathbf{int}\ \ ,\ \ \epsilon\ ]\ \mathbf{int}$

Note that the formal type argument α is not used, so $\mathbf{bool}$ and $Wicket$ do not occur in the expansion. Note also that the bound type variable γ in the definition of the type synonym is renamed in order to avoid name capture.

Type synonyms are expanded before any restriction on the enclosing context is applied. For example, given the declaration of $MySynonym$ above,

$\langle\beta\rangle[MySynonym\ \beta]\ \mathbf{int}$ // error

is not a legal type: $MySynonym\ \beta$ is itself legal and is thus expanded to $\mathbf{int}$ according to its definition above. After expansion, the enclosing context becomes $\langle\beta\rangle[\mathbf{int}]\mathbf{int}$, which is not legal, because β is not used in the domain type of the map type.

Definitions of type synonyms are not allowed to be recursive (or mutually recursive). More precisely, consider the directed graph whose vertices are the type synonyms in a program, and in which there is an edge from a type synonym S to each type synonym mentioned in its definition. Then, this graph must be acyclic.

3 Constants and functions

Constant declarations have the following syntactic form:

$$\begin{array}{rcl} \mathit{ConstantDecl} & ::= & \mathbf{const}\ \mathit{Attribute}^*\ \mathbf{unique}^?\ \mathit{IdsType}\ \mathit{OrderSpec}\ ; \\ \mathit{IdsType} & ::= & \mathit{Id}\text{,}^+ : \mathit{Type} \end{array}$$

For each Id in the $IdsType$, the declaration introduces Id as a symbolic constant of type $Type$. The constants in a program must have names that are distinct from other constants and global variables. The attributes, $\mathbf{unique}$ designation, and $OrderSpec$ clause apply to each of the Id's introduced, as if each Id were introduced in a separate $\mathbf{const}$ declaration.

Declaring a constant with $\mathbf{unique}$ makes manifest that the constant has a value that is different from the values of other $\mathbf{unique}$ constants of the same type. By using $\mathbf{unique}$, one can avoid declaring axioms for the quadratically many distinctions. For example, the use of $\mathbf{unique}$ in the declaration (0) of $RGBColor$ constants implies the condition expressed by the following axiom:

$$\mathbf{axiom}\ \mathit{red} \neq \mathit{green} \wedge \mathit{green} \neq \mathit{blue} \wedge \mathit{blue} \neq \mathit{red};$$

The $OrderSpec$ clause optionally specifies the relative position of the constant in the partial order $<:$ for type $Type$. Section 10 gives the details.

A function declaration has one of the following syntactic forms:

$$\begin{array}{rcl} \mathit{FunctionDecl} & ::= & \mathbf{function}\ \mathit{Attribute}^*\ \mathit{Id}\ \mathit{FSig}\ ; \\ & | & \mathbf{function}\ \mathit{Attribute}^*\ \mathit{Id}\ \mathit{FSig}\ \{\ \mathit{Expr}\ \} \end{array}$$

It introduces a function named Id. The functions in a program must have distinct names. The function's type signature has the form:

$$\begin{array}{rcl} \mathit{FSig} & ::= & \mathit{TypeArgs}^?\ (\ \mathit{FArg}\text{,}^*\ )\ \mathbf{returns}\ (\ \mathit{FArg}\ ) \\ \mathit{FArg} & ::= & \mathit{FArgName}^?\ \mathit{Type} \\ \mathit{FArgName} & ::= & \mathit{Id}\ : \end{array}$$

The signature includes any number of arguments and one result type.

A function can be polymorphic. Universally quantified type identifiers are introduced in $TypeArgs$. They can then be used as types in the declaration of the function's arguments and result, as well as in the optional $Expr$. The identifiers introduced in $TypeArgs$ must not contain any duplicates. For example,

$$\mathbf{function}\ \mathit{volume}\langle\alpha\rangle(\mathit{Barrel}\ \alpha)\ \mathbf{returns}\ (\mathbf{int});$$

declares a function intended to return the volume of any kind of barrel.

There is a restriction that every type identifier introduced in $TypeArgs$ must be used (after the expansion of type synonyms) somewhere among the types of the function's arguments.

Optionally, the function arguments and result value can be named, for example to document the purpose of each argument. For example, the use of argument names in

$$\textbf{function}\ cylinderVolume(radius\colon \textbf{int}, height\colon \textbf{int})\ \textbf{returns}\ (\textbf{int});$$

serves as a human aid in remember the order of the arguments. Among the argument and result names supplied, no duplicates are allowed.

The second form of function defines a value for the function. The $Expr$ is allowed to refer to the Id's supplied as argument names. A declaration

$$\textbf{function}\ attrs\ F(args)\ \textbf{returns}\ (res)\ \{\ E\ \}$$

is equivalent to

$$\textbf{function}\ attrs\ F(args)\ \textbf{returns}\ (res);$$
$$\textbf{axiom}\ (\forall\ args' \bullet\ F(argIds) == E\ );$$

where $args'$ is $args$ with any omitted argument names filled in by fresh identifiers and $argIds$ is the list of argument names in $args'$.

4 Expressions

Boogie expressions include constants, variables, equality and arithmetic relations, boolean connectives, simple arithmetic operators, logical quantifiers, and an ordering operator. They follow the grammar in Fig. 0. The operators come in both Unicode form and ASCII form, as shown in Fig. 1.

Most of the expressions are standard and self-explanatory. Note that $\vee$ and $\wedge$ have equal binding power but do not associate with each other; thus, an expression that mentions both must necessarily use parentheses somewhere to disambiguate.

4.0 Division and modulo

Because division and modulo are defined differently in different source languages, Boogie provides syntax for the operators $/$ and $\%$ but gives them no meaning. Instead, the meaning of these operators can be axiomatized according to their desired meaning. For Modula-3 or Java, the following axioms can be used:

$$\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{x\,\%\,y\}\ \{x/y\}\quad x\,\%\,y == x - x/y * y\ );$$
$$\begin{aligned}&\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{x\,\%\,y\}\\ &\qquad (0 < y\ \ \Rightarrow\ \ 0 \leqslant x\,\%\,y\ \ \wedge\ \ x\,\%\,y < y)\ \wedge\\ &\qquad (y < 0\ \ \Rightarrow\ \ y < x\,\%\,y\ \ \wedge\ \ x\,\%\,y \leqslant 0)\ );\end{aligned}$$

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & E0 \\
E0 & ::= & E1 \mid E1\ \mathit{EquivOp}\ E0 \\
E1 & ::= & E2 \mid E2\ \mathit{ImplOp}\ E1 \\
E2 & ::= & E3 \mid E3\ \mathit{EOr}^{+} \mid E3\ \mathit{EAnd}^{+} \\
\mathit{EOr} & ::= & \mathit{OrOp}\ E3 \\
\mathit{EAnd} & ::= & \mathit{AndOp}\ E3 \\
E3 & ::= & E4 \mid E4\ \mathit{RelOp}\ E4 \\
E4 & ::= & E5 \mid E4\ \mathit{ConcatOp}\ E5 \\
E5 & ::= & E6 \mid E5\ \mathit{AddOp}\ E6 \\
E6 & ::= & E7 \mid E6\ \mathit{MulOp}\ E7 \\
E7 & ::= & \mathit{UnOp}^{*}\ E8 \\
E8 & ::= & E9\ \mathit{MapOp}^{*} \\
\mathit{MapOp} & ::= & [\ \mathit{Expr},^{+}\ \mathit{MapUpdate}^{?}\ ] \\
 & \mid & [\ \mathit{Number}\ :\ \mathit{Number}\ ] \\
\mathit{MapUpdate} & ::= & :=\ \mathit{Expr} \\
E9 & ::= & \mathbf{false} \mid \mathbf{true} \mid \mathit{Number} \mid \mathit{BitVector} \\
 & \mid & \mathit{Id}\ \mathit{FuncApplication}^{?} \\
 & \mid & \mathbf{old}\ (\ \mathit{Expr}\ ) \\
 & \mid & (\ \mathit{QOp}\ \mathit{TypeArgs}^{?}\ \mathit{IdsType},^{+}\ \mathit{QSep}\ \mathit{TrigAttr}^{*}\ \mathit{Expr}\ ) \\
 & \mid & (\ \mathit{Expr}\ ) \\
\mathit{FuncApplication} & ::= & (\ \mathit{Expr},^{*}\ ) \\
\mathit{TrigAttr} & ::= & \mathit{Trigger} \mid \mathit{Attribute} \\
\mathit{Number} & ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{BitVector} & ::= & 0\mathbf{bv}0 \\
 & \mid & 0\mathbf{bv}1 \mid 1\mathbf{bv}1 \\
 & \mid & 0\mathbf{bv}2 \mid 1\mathbf{bv}2 \mid 2\mathbf{bv}2 \mid 3\mathbf{bv}2 \\
 & \mid & 0\mathbf{bv}3 \mid 1\mathbf{bv}3 \mid 2\mathbf{bv}3 \mid 3\mathbf{bv}3 \mid 4\mathbf{bv}3 \mid 5\mathbf{bv}3 \mid 6\mathbf{bv}3 \mid 7\mathbf{bv}3 \\
 & \mid & \cdots
\end{array}
$$

Fig. 0. Boogie's expression grammar.

EquivOp	::=	$\Leftrightarrow$	|	<==>
ImplOp	::=	$\Rightarrow$	|	==>
OrOp	::=	$\vee$	|	||
AndOp	::=	$\wedge$	|	&&
RelOp	::=	==		
	|	$\neq$	|	!=
	|	$<$ | $>$		
	|	$\leqslant$ | $\geqslant$	|	<= | >=
	|	<:		
ConcatOp	::=	++		
AddOp	::=	+ | −		
MulOp	::=	$*$ | / | %		
UnOp	::=	$\neg$	|	!
	|	−		
QOp	::=	$\forall$ | $\exists$	|	**forall** | **exists**
QSep	::=	$\bullet$	|	::

Fig. 1. Expression operators, shown in both their Unicode (left column) and ASCII (right column) forms.

whereas for C, C#, and Spec#, the following axioms can be used:

$$\begin{array}{l}
\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{x \,\%\, y\}\ \{x/y\} \quad x \,\%\, y == x - x/y * y\ );\\
\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{x \,\%\, y\}\\
\qquad\qquad (0 \leqslant x\ \wedge\ 0 < y\ \Rightarrow\ 0 \leqslant x \,\%\, y\ \wedge\ x \,\%\, y < y)\ \wedge\\
\qquad\qquad (0 \leqslant x\ \wedge\ y < 0\ \Rightarrow\ 0 \leqslant x \,\%\, y\ \wedge\ x \,\%\, y < -y)\ \wedge\\
\qquad\qquad (x \leqslant 0\ \wedge\ 0 < y\ \Rightarrow\ -y < x \,\%\, y\ \wedge\ x \,\%\, y \leqslant 0)\ \wedge\\
\qquad\qquad (x \leqslant 0\ \wedge\ y < 0\ \Rightarrow\ y < x \,\%\, y\ \wedge\ x \,\%\, y \leqslant 0)\ );
\end{array}$$

These axioms are not complete and may in some applications of Boogie need to be extended with further axioms. For example, the following axioms may sometimes be useful:

$$\begin{array}{l}
\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{(x + y) \,\%\, y\}\\
\qquad\qquad 0 \leqslant x\ \wedge\ 0 < y\ \Rightarrow\ (x + y) \,\%\, y == x \,\%\, y\ );\\
\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{(y + x) \,\%\, y\}\\
\qquad\qquad 0 \leqslant x\ \wedge\ 0 < y\ \Rightarrow\ (y + x) \,\%\, y == x \,\%\, y\ );\\
\textbf{axiom}\ (\forall\, x\colon \textbf{int}, y\colon \textbf{int} \bullet\ \{(x - y) \,\%\, y\}\\
\qquad\qquad 0 \leqslant x - y\ \wedge\ 0 < y\ \Rightarrow\ (x - y) \,\%\, y == x \,\%\, y\ );\\
\textbf{axiom}\ (\forall\, a\colon \textbf{int}, b\colon \textbf{int}, d\colon \textbf{int} \bullet\ \{a \,\%\, d, b \,\%\, d\}\\
\qquad\qquad 2 \leqslant d\ \wedge\ a \,\%\, d == b \,\%\, d\ \wedge\ a < b\ \Rightarrow\ a + d \leqslant b\ );
\end{array}$$

4.1 Map selection and update

A map range value is selected by supplying values in the map's domain, given in the same order as the domain types are declared. For example, given the map

$$\textbf{const}\ a\colon [\textbf{int}, \mathit{RGBColor}]\, \mathit{Wicket};$$

the expression $a[5, red]$ denotes the wicket at $5, red$. The map update expression $a[5, red := w]$ returns a map, namely the map that is like a except possibly at $5, red$, where it has value w. Note that the map update expression does not change the value of the map a; it simply returns a map whose value is very much like that of a.

Maps do not necessarily satisfy *extensionality*—the property that maps are equal if all their elements are equal. For example, extensionality would imply:

$$b[j := b[j]] == b$$

but this property does not necessarily hold in Boogie.

4.2 Bit vectors

Bit-vector literals specify their own type. A literal $\mathrm{X}\mathbf{bv}\mathrm{K}$ is legal when X (which is a number represented in decimal) is 0 or is expressible within K bits. Other than equality and disequality ($\neq$), only two operations are defined on bit vectors: extraction and concatenation.

For a bit vector b of type $\mathbf{bv}\mathrm{K}$ and literals M and N, the *extraction* expression $b[N\!:M]$ denotes the bit vector obtained by first dropping the M least significant bits of b and then returning the next $N - M$ least significant bits. In other words, $b[N\!:M]$ denotes the $N - M$ bits of b starting at bit M. The result has type $\mathbf{bv}(\mathrm{N} - \mathrm{M})$. The extraction expression requires $\mathrm{K} \geqslant N \geqslant M \geqslant 0$.

For bit vectors b and c of types $\mathbf{bv}\mathrm{K}$ and $\mathbf{bv}\mathrm{N}$, respectively, the *concatenation* expression $b +\!+ c$ denotes the bit vector of type $\mathbf{bv}(\mathrm{K}+\mathrm{N})$ consisting of the bits from b concatenated by the bits of c.

For example, $(13\mathbf{bv}6 +\!+ 4\mathbf{bv}3)[5\!:2]$ is $3\mathbf{bv}3$, since (showing literals in non-Boogie binary notation):

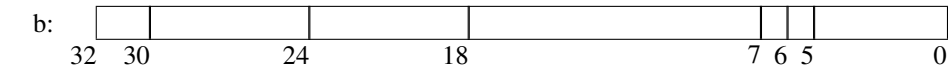
$$\begin{array}{ll} & (001101_2 +\!+ 100_2)[5\!:2] \\ = & \quad \{\ \text{concatenation}\ \} \\ & 001101100_2[5\!:2] \\ = & \quad \{\ \text{drop the 2 low-order bits}\ \} \\ & 0011011_2[3\!:0] \\ = & \quad \{\ \text{select the 3 low-order bits}\ \} \\ & 011_2 \end{array}$$

Note that an extraction expression $b[N\!:M]$ indicates a half-open interval of bits, going from bit M to, but not including, bit N. The advantages of half-open intervals are the familiar ones: subtracting the two bounds yields the number of bits in the result—for example, $b[24\!:18]$ is a vector of $24 - 18 = 6$ bits—and concatenation fuses its arguments if the "middle" bound is repeated—for example,

$$b[24\!:18] +\!+ b[18\!:7] \quad == \quad b[24\!:7]$$

since the "middle" bound 18 is repeated. The advantage of reversing the order of the arguments M and N (that is, listing them in reverse numerical order) is that this gives the familar representation of bit vectors as binary numbers, with the least-significant bit

to the right. For example, $0\mathbf{bv}10 ++ b[24\!:\!18]$ zero-extends the 6-bit quantity $b[24\!:\!18]$ to a 16-bit word. Pictorially, a familiar representation like:



has the following subdivisions:

$$b[32\!:\!30] ++ b[30\!:\!24] ++ b[24\!:\!18] ++ b[18\!:\!7] ++ b[7\!:\!6] ++ b[6\!:\!5] ++ b[5\!:\!0]$$

where b is of type $\mathbf{bv}32$.

A Boogie program can declare additional bit-vector operations as functions. The properties of these functions can be defined by axioms. Alternatively, as explained in Section 11, prover-specific attributes can be used to identify such functions with the prover's native support of bit vectors.

4.3 Old expressions

Postconditions and procedure implementations are *two-state contexts*. This means that it is possible to refer to two different values of each variable. In a postcondition, the two states are the pre- and post-states of the procedure's invocations, and in a procedure implementation, the two states are the pre-state of the procedure and the current state. In both cases, the pre-state value of an expression is denoted by enclosing it as the argument to $\mathbf{old}$. For example, in the postcondition of a procedure, if x and y are global variables, then $\mathbf{old}(x+y)$ refers to the value of $x+y$ on entry to the procedure, whereas $x + y$ not enclosed inside any $\mathbf{old}$ expression denotes the value of $x + y$ on exit from the procedure.

Only global variables are affected by $\mathbf{old}$ expressions. For example, if a is a global variable, b is a local variable, and c is an out-parameter, then the use of $\mathbf{old}(a+b+c)$ in a procedure implementation is equivalent to $\mathbf{old}(a) + b + c$. Stated differently, $\mathbf{old}$ distributes to the leaves of expressions and is the identity for every leaf expression that is not a global variable. Nested occurrences of $\mathbf{old}$ do not further change the meaning of the expression; $\mathbf{old}(\mathbf{old}(E))$ is equivalent to just $\mathbf{old}(E)$. In other words, $\mathbf{old}$ is idempotent.

4.4 Logical quantifiers

Boogie supports both universal and existential quantifiers. For example, the expression

$$(\,\forall\, t\colon \mathit{Wicket} \bullet\ \mathit{age}(t) \leqslant 20\ )$$

says that no wicket is older than 20.

The bound variables among the $\mathit{IdsType}$'s must be distinct and must be different from local variables, parameters, and other bound variables in scope (but they may have the same names as constants and global variables). The bound variables are in scope in the body of the quantifier expression and in its $\mathit{TrigAttr}$ components.

In addition to the $IdsType$ bound variables, the quantifier can declare bound type variables in $TypeArgs$. Such type variables must be distinct and must be different from other type variables in scope. The type variables may be used in the types of the $IdsType$ bound variables, in the $TrigAttr$ components, and in the body of the quantifier. In fact, each type variable introduced must be mentioned somewhere within the types of the $IdsType$'s. For example,

$$(\,\forall\,\langle\alpha\rangle\ x\colon \mathbf{int},\ y\colon Barrel\ \alpha\ \bullet\ Q(x,y)\,)$$

and

$$(\,\forall\,x\colon \mathbf{int}\ \bullet\ (\,\forall\,\langle\alpha\rangle\ y\colon Barrel\ \alpha\ \bullet\ Q(x,y)\,))$$

are legal whereas

$$(\,\forall\,\langle\alpha\rangle\ x\colon \mathbf{int}\ \bullet\ (\,\forall\,y\colon Barrel\ \alpha\ \bullet\ Q(x,y)\,))\qquad \text{// error}$$

is not (because the type of the bound variable in the outer quantifier does not depend on α).

The bound type variables are universally or existentially quantified as indicated by QOp. For example,

$$(\,\forall\,\langle\alpha\rangle\ b\colon Barrel\ \alpha\ \bullet\ P(b)\,)$$

says that $P(b)$ holds for all types α and all values $b\colon Barrel\ \alpha$. And:

$$(\,\forall\,\langle\alpha\rangle\ b\colon Barrel\ \alpha\ \bullet\ P(b)\,)$$

says that there exists a type α and a value $b\colon Barrel\ \alpha$ such that $P(b)$ holds.

Note, an expression that quantifies over types is not the same as a universal (or extential) type. Also, a quantified expression always has type $\mathbf{bool}$, never a universal (or existential) type. If this is a point of confusion, it may help to consider the higher-order function $\forall$, which takes a function, say f, as argument and returns $\mathbf{true}$ if and only if f returns $\mathbf{true}$ on all of f's inputs. Then, a quantification $(\,\forall\,x\colon T\ \bullet\ E\,)$ over a bound variable x can be viewed as the higher-order function $\forall$ applied to the value abstraction $(\,\lambda\,x\colon T\ \bullet\ E\,)$. And a quantification $(\,\forall\,\langle\alpha\rangle\ x\colon\cdots\ \bullet\ E\,)$ can be viewed as the higher-order function $\forall$ applied to the type abstraction $(\,\Lambda\,\alpha\ \bullet\ (\,\forall\,x\colon\cdots\ \bullet\ E\,))$.

Like top-level declarations, quantifiers can be decorated with attributes. In addition, quantifiers can include trigger expressions. Such attributes and triggers are described in Section 11.

5 Type checking of expressions

On the whole, type checking in Boogie is humdrum. Only map operations and equality are noteworthy. Map selection and update operate on polymorphic maps, which means that the corresponding type rules consider the instantiation of generic types. Equality is type checked rather liberally, allowing two expressions to be compared if there is some instantiation of type parameters that makes the two operands have the same type. The

semantic meaning of a == b is that the evaluation of expressions a and b lead not only to the same value but also to values of the same type.

Well-typing is expressed by the judgment $\mathcal{T}, \mathcal{V} \Vdash a : T$, which says that in a context with type parameters $\mathcal{T}$ and identifier type bindings $\mathcal{V}$ (like x: T and $f\langle\overline{\alpha}\rangle(\overline{T})$ **returns** (U)), expression a has type T.

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{bool} \qquad \mathcal{T}, \mathcal{V} \Vdash b : \mathbf{bool} \qquad \oplus \in \{ \iff , \Rightarrow , \vee , \wedge \}}{\mathcal{T}, \mathcal{V} \Vdash a \oplus b : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{bool}}{\mathcal{T}, \mathcal{V} \Vdash \neg a : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{int} \qquad \mathcal{T}, \mathcal{V} \Vdash b : \mathbf{int} \qquad \oplus \in \{<, \leqslant, \geqslant, >\}}{\mathcal{T}, \mathcal{V} \Vdash a \oplus b : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{int} \qquad \mathcal{T}, \mathcal{V} \Vdash b : \mathbf{int} \qquad \oplus \in \{+, -, *, /, \%\}}{\mathcal{T}, \mathcal{V} \Vdash a \oplus b : \mathbf{int}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{int}}{\mathcal{T}, \mathcal{V} \Vdash -a : \mathbf{int}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{bv}\mathrm{K} \qquad \mathcal{T}, \mathcal{V} \Vdash b : \mathbf{bv}\mathrm{N}}{\mathcal{T}, \mathcal{V} \Vdash a +\!+ b : \mathbf{bv}(\mathrm{K} + \mathrm{N})}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \mathbf{bv}\mathrm{K} \qquad \mathrm{K} \geqslant \mathrm{N} \geqslant \mathrm{M} \geqslant 0}{\mathcal{T}, \mathcal{V} \Vdash a[N\!:M] : \mathbf{bv}(\mathrm{N} - \mathrm{M})}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : T}{\mathcal{T}, \mathcal{V} \Vdash (a) : T} \qquad \frac{\mathcal{T}, \mathcal{V} \Vdash a : T}{\mathcal{T}, \mathcal{V} \Vdash \mathbf{old}(a) : T}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash \diamond}{\mathcal{T}, \mathcal{V} \Vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{\mathcal{T}, \mathcal{V} \Vdash \diamond}{\mathcal{T}, \mathcal{V} \Vdash \mathbf{true} : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash \diamond \qquad N \geqslant 0}{\mathcal{T}, \mathcal{V} \Vdash N : \mathbf{int}} \qquad \frac{\mathcal{T}, \mathcal{V} \Vdash \diamond \qquad K \geqslant 0 \qquad (N = 0) \text{ or } (0 \leqslant N < 2^K)}{\mathcal{T}, \mathcal{V} \Vdash N\mathbf{bv}\mathrm{K} : \mathbf{bv}\mathrm{K}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : T \qquad \mathcal{T}, \mathcal{V} \Vdash b : U \qquad T\sigma == U\sigma}{\mathcal{T}, \mathcal{V} \Vdash a == b : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash \neg(a == b) : \mathbf{bool}}{\mathcal{T}, \mathcal{V} \Vdash a \neq b : \mathbf{bool}} \qquad \frac{\mathcal{T}, \mathcal{V} \Vdash a : T \qquad \mathcal{T}, \mathcal{V} \Vdash b : T}{\mathcal{T}, \mathcal{V} \Vdash a <: b : \mathbf{bool}}$$

$$\frac{(\mathcal{T}, \overline{\alpha}), (\mathcal{V}, \overline{x}\!: \overline{T}) \Vdash a : \mathbf{bool} \qquad (\mathcal{T}, \overline{\alpha}), (\mathcal{V}, \overline{x}\!: \overline{T}) \Vdash_{attr} t \quad \text{for all } t \in \overline{t} \qquad \text{Æ} \in \{\forall, \exists\}}{\mathcal{T}, \mathcal{V} \Vdash (\ \text{Æ}\,\langle\overline{\alpha}\rangle\ \overline{x}\!: \overline{T} \bullet\ \overline{t}\ a\ ) : \mathbf{bool}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \langle\overline{\alpha}\rangle[\overline{U}]\,V \qquad \mathcal{T}, \mathcal{V} \Vdash b : U\sigma \quad \text{for all } (b, U) \in (\overline{b}, \overline{U}) \qquad dom(\sigma) = \{\overline{\alpha}\}}{\mathcal{T}, \mathcal{V} \Vdash a[\overline{b}] : V\sigma}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash a : \langle\overline{\alpha}\rangle[\overline{U}]\, V \qquad dom(\sigma) = \{\overline{\alpha}\} \quad \mathcal{T}, \mathcal{V} \Vdash e : V\sigma \qquad \mathcal{T}, \mathcal{V} \Vdash b : U\sigma \ \text{ for all } (b, U) \in (\overline{b}, \overline{U})}{\mathcal{T}, \mathcal{V} \Vdash a[\overline{b} := e] : \langle\overline{\alpha}\rangle[\overline{U}]\, V}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash \diamond \qquad x\colon T \in \mathcal{V}}{\mathcal{T}, \mathcal{V} \Vdash x : T}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash \diamond \qquad f\langle\overline{\alpha}\rangle(\overline{U})\ \mathbf{returns}\ (V) \in \mathcal{V} \qquad dom(\sigma) = \{\overline{\alpha}\} \quad \mathcal{T}, \mathcal{V} \Vdash a : U\sigma \ \text{ for all } (a, U) \in (\overline{a}, \overline{U})}{\mathcal{T}, \mathcal{V} \Vdash f(\overline{a}) : V\sigma}$$

We have now seen enough of Boogie's type system and expressions to consider some examples.

5.0 Example: Modeling memory

One of the first and most important decisions one makes when designing a translation of a source language into Boogie is the representation of memory. In a type-safe source language with object references and fields, one option is to split the memory (the heap) up into one part corresponding to each field. For example, a program containing the following class:

```
class C { int data; C next; }
```

would give rise to a Boogie program that includes the following declarations:

$$\begin{array}{l}\mathbf{type}\ \mathit{Ref};\\ \mathbf{var}\ C.\mathit{data}\colon [\mathit{Ref}]\mathbf{int};\\ \mathbf{var}\ C.\mathit{next}\colon [\mathit{Ref}]\mathit{Ref};\end{array}$$

where Ref is a type intended to represent object references. Here, each field is a map from object references to values. A field selection expression o.data in the source language is then translated into the Boogie expression $C.\mathit{data}[o]$, which has type $\mathbf{int}$ (see type rules above). In addition to the fields declared in the source program, it is typical that a Boogie encoding would model a number of ghost fields. For example,

$$\mathbf{var}\ \mathit{alloc}\colon [\mathit{Ref}]\mathbf{bool};$$

keeps track of which object references have been allocated in the current state.

However, it is also possible to define the heap as one variable, mapping object references and field names to values. Since the values are of different types, depending on which field name is used, it is convenient to use a polymorphic map, here defined via a type synonym:

$$\begin{array}{l}\mathbf{type}\ \mathit{Field}\ \alpha;\\ \mathbf{type}\ \mathit{HeapType} = \langle\alpha\rangle[\mathit{Ref}, \mathit{Field}\ \alpha]\alpha;\\ \mathbf{var}\ \mathit{Heap}\colon \mathit{HeapType};\end{array}$$

The fields in the source-program snippet above, and a ghost field for allocation, are now translated into distinct constants:

$$\textbf{const unique } C.data\colon \mathit{Field}\ \textbf{int};$$
$$\textbf{const unique } C.next\colon \mathit{Field}\ \mathit{Ref};$$
$$\textbf{const unique } \mathit{alloc}\colon \mathit{Field}\ \textbf{bool};$$

The field selection expression $o.data$ is now translated into the Boogie expression $\mathit{Heap}[o, C.data]$, which has type **int** (in the type rule for map selection above, use σ as the substitution of **int** for α).

There are several advantages to encoding the heap in this way, instead of as one individual variable per field. One advantage is conciseness, because functions that operate on the heap need only take one argument. For example, a boolean model field m in a class C can be modeled by a function:

$$\textbf{function } C.m(\mathit{heap}\colon \langle\alpha\rangle[\mathit{Ref}, \mathit{Field}\ \alpha]\alpha,\ \mathit{this}\colon \mathit{Ref})\ \textbf{returns}\ (\textbf{bool});$$

Another important advantage is the ability to quantify over fields. For example, the quantification can easily range over fields that are not in scope in the source-language module translated into Boogie, which enables modular verification in a nice way.

Yet another option to model the heap as mapping each object reference to a map from field names to values, in other words a currying of the heap representation we just considered:

$$\textbf{var } \mathit{ObjStore}\colon [\mathit{Ref}]\langle\alpha\rangle[\mathit{Field}\ \alpha]\alpha;$$

This representation may be advantageous if changes to entire objects (like $\mathit{ObjStore}[o] := r$) occur more frequently than changes to individual fields ($\mathit{ObjStore}[o][data] := 5$), or if it often is necessary to copy all fields of one object into the fields of another ($\mathit{ObjStore}[o] := \mathit{ObjStore}[template]$). It also gives a clean way to define a model-field function that depends only on the field of the object itself:

$$\textbf{function } C.m(\mathit{this}\colon \mathit{Ref},\ \mathit{dataRecord}\colon \langle\alpha\rangle[\mathit{Field}\ \alpha]\alpha)\ \textbf{returns}\ (\textbf{bool});$$

Unlike the $C.m$ function we considered earlier, this function takes as arguments only those parts of the heap that make up the fields of the object. For a model-field selection expression $o.m$, the translation using this function is $C.m(\mathit{ObjStore}[o])$. Note, however, that a model field is commonly defined over the fields of more than object, in which case it is necessary to pass in larger parts of the heap, possibly the entire heap as in the earlier definition of $C.m$.

5.1 Example: Frame conditions

In program verification, it is important to curb the effects that a procedure can have. In Boogie, this is done by a combination of modifies and ensures clauses. Since Boogie's modifies clauses are coarse-grained, one uses a postcondition that gives more detail. This particular postcondition is often referred to as the *frame condition* of the procedure. Suppose a source-language method is allowed to modify the $data$ field of a parameter

p, and that it is also allowed to allocate new objects and modify their fields. The Boogie modifies clause and frame condition for this can be written:

$$\begin{array}{l}\textbf{modifies } \mathit{Heap};\\ \textbf{ensures } (\,\forall\, \langle\alpha\rangle\ o\colon \mathit{Ref},\ f\colon \mathit{Field}\ \alpha \bullet\\ \qquad\qquad \mathit{Heap}[o,f] = \textbf{old}(\mathit{Heap})[o,f] \ \vee\\ \qquad\qquad (o = p \wedge f = C.\mathit{data}) \ \vee\\ \qquad\qquad \neg\textbf{old}(\mathit{Heap})[o, \mathit{alloc}]\ );\end{array}$$

There are several interesting things to note about this frame condition. First, it contains a quantification over all fields, f. Second, the type of f is supposed to be any field, which is written $\mathit{Field}\ \alpha$, where α denotes any type. Since α needs to be bound somewhere, the universal quantification over type α is used. Third, it is necessary to be able to compare f against the particular field name $C.\mathit{data}$, which is done by the equality $f = C.\mathit{data}$. But f (of type $\mathit{Field}\ \alpha$) and $C.\mathit{data}$ (of type $\mathit{Field}\ \textbf{int}$) do not have the same type; nevertheless, Boogie allows the equality comparison (see the type rules), because there is some overlap in their types, given the fact that α is a type variable.

6 Axioms

An axiom declaration has the form

$$\mathit{AxiomDecl} \quad ::= \quad \textbf{axiom}\ \mathit{Attribute}^*\ \mathit{Expr}\ ;$$

The given expression must be of type boolean. Global variables and **old** expressions are not allowed in Expr. The axiom expresses some properties about the program's constants and functions. The effect of this is to restrict execution traces, as described later.

Note that just like first-order logic makes it possible to write an antecedent that is equivalent to **false**, trivially rendering the formula valid, it is possible to write inconsistent axioms in Boogie. For example, the inconsistent axiom

$$\textbf{axiom false};$$

has the effect of eliminating all execution traces, which means that all implementations in the Boogie program are trivially correct.

7 Mutable variables, states, and execution traces

The imperative parts of a Boogie program operate on a *state space*. The program state space is the Cartesian product of variables, including a fictitious variable $g_{\textbf{old}}$ for every global variable g. A state thus assigns a value to each variable. In the body of a procedure implementation, local variables, out-parameters, and those global variables appearing in any of the procedure's modifies clauses are *mutable variables*, which means they can changed by statements in the procedure's implementation.

Global variables are in scope for all procedures and are declared as follows:

$$\begin{array}{rcl}\mathit{VarDecl} & ::= & \textbf{var}\ \mathit{Attribute}^*\ \mathit{IdsTypeWhere}{,}^+\ ;\\ \mathit{IdsTypeWhere} & ::= & \mathit{IdsType}\ \mathit{WhereClause}^?\end{array}$$

For each *IdsTypeWhere*, the declaration introduces each *Id* as a global variable of type *Type*. At times in an execution trace when the value of the variable is chosen arbitrarily, the value is chosen to satisfy *WhereClause*, as explained in Sections 8 and 9. The global variables in a program must have names that are distinct from constants and other global variables. The *Type* and *WhereClause* of an *IdsTypeWhere* apply to each of the *Id*'s declared, as if each *Id* were introduced in a separate **var** declaration. The attributes of the declaration apply to each of the *Id*'s introduced.

The expression given by a *WhereClause* appearing among the global variables must be of type boolean and can refer to any constant, function, and global variable. Here and throughout, if an optional *WhereClause* is omitted, it defaults to "**where true**".

An *execution trace* is a nonempty, possibly infinite sequence of states, optionally followed by the special value $\lightning$, where every state satisfies all axioms and order specifications declared in the program.

A finite execution trace not ending with $\lightning$ is said to *terminate*, and it corresponds to a well-behaved program execution that terminates after a finite number of steps. An execution trace that ends with $\lightning$ is said to *go wrong*, and it corresponds to an ill-behaved program exeuction, that is, a program execution that after a finite number of steps crashes. So, a *finite execution trace* either terminates or goes wrong. An infinite execution trace is said to *diverge*, and it corresponds to a well-behaved never-ending program execution.

8 Procedures and implementations

A *procedure* defines two sets of execution traces, the *caller traces* and the *callee traces*. These traces are determined by the procedure's signature and specification. A procedure *implementation* also defines a set of execution traces, the *implementation traces*. These traces are determined by the implementation's body, which is a statement. A procedure can be invoked by the call statement, which gives rise to the traces defined by the procedure's caller traces. An implementation is correct if and only if the implementation traces form a subset of the procedure's callee traces.

8.0 Syntax

A procedure declaration has one of two forms:

$$
\begin{array}{rcl}
\mathit{ProcedureDecl} & ::= & \textbf{procedure}\ \mathit{Attribute}^*\ \mathit{Id}\ \mathit{PSig}\ ;\ \mathit{Spec}^* \\
& | & \textbf{procedure}\ \mathit{Attribute}^*\ \mathit{Id}\ \mathit{PSig}\ \mathit{Spec}^*\ \mathit{Body} \\
\mathit{PSig} & ::= & \mathit{TypeArgs}^?\ (\ \mathit{IdsTypeWhere},^*\ )\ \mathit{OutParameters}^? \\
\mathit{OutParameters} & ::= & \textbf{returns}\ (\ \mathit{IdsTypeWhere},^*\ )
\end{array}
$$

It introduces *Id* as the name of a procedure. The procedure is parameterized according to the signature *PSig*, and it stands for the execution traces defined by the signature and the *Spec* clauses. If *OutParameters* is omitted, it defaults to "**returns** ()". The procedures in a program must have distinct names.

A procedure can be polymorphic. Universally quantified type identifiers are introduced in *TypeArgs*. They can then be used as types in the declaration of the procedure's

in- and out-parameters, as well as in the specification and optional $Body$. The identifiers introduced in $TypeArgs$ must not contain any duplicates. There is a restriction that every type identifier introduced in $TypeArgs$ must be used (after the expansion of type synonyms) somewhere among the types of the procedure's in-parameters.

A signature declares some number of type arguments, in-parameters, and out-parameters. The Id's declared as type arguments must be distinct (from each other), and the Id's declared as in- or out-parameters must also be distinct (from each other). The type arguments may be used in types appearing in the procedure signature, specification, and body. The expression given by a $WhereClause$ appearing among the parameters must be of type boolean and can refer to any constant, function, global variable, and in- and out-parameter. The $Type$ and $WhereClause$ of an $IdsTypeWhere$ apply to each of the Id's declared, as if each Id were introduced in its own $IdsTypeWhere$ declaration.

A procedure specification consists of three kinds of clauses. A *precondition* (requires clause) specifies a boolean condition that holds in the initial state of each execution trace of the procedure. Generally, it is the caller's responsibility to establish the precondition at a call site, and the implementation gets to assume the precondition to hold on entry. A *postcondition* (ensures clause) specifies a boolean condition that relates the initial and final states of each finite execution trace of the procedure. Generally, it is the implementation's responsibility to establish the postcondition on exit, and the caller gets to assume the postcondition to hold upon return. The *modifies clauses* of a procedure specification list those global variables that are allowed to change during the course of the procedure's execution traces.

Syntactically, the specification consists of any number of preconditions (requires clauses), modifies clauses, and postconditions (ensures clauses):

$$
\begin{array}{rcl}
Spec & ::= & \mathbf{free}^{?}\ \mathbf{requires}\ Attribute^{*}\ \ Expr\ ; \\
& | & \mathbf{free}^{?}\ \mathbf{modifies}\ Attribute^{*}\ \ Id\text{,}^{*}\ ; \\
& | & \mathbf{free}^{?}\ \mathbf{ensures}\ Attribute^{*}\ \ Expr\ ;
\end{array}
$$

A precondition can refer to global variables and in-parameters, but not to out-parameters. A precondition is not allowed to use **old** expressions. A modifies clause is only allowed to list global variables. A postcondition can refer to global variables and to in- and out-parameters. It uses **old** expressions to refer to the values of global variables in the initial state. A specification clause that uses the **free** keyword is called *free*; otherwise, it is called *checked*.

A procedure implementation is declared as follows:

$$
\begin{array}{rcl}
ImplementationDecl & ::= & \mathbf{implementation}\ Attribute^{*}\ Id\ ISig\ Body^{*} \\
ISig & ::= & TypeArgs^{?}\ (\ IdsType\text{,}^{*}\ )\ OutParameters^{?} \\
OutParameters & ::= & \mathbf{returns}\ (\ IdsType\text{,}^{*}\ )
\end{array}
$$

For every implementation P, the program must also contain a procedure declaration for P. The implementation declaration must repeat the signature of the procedure, with a few exceptions. First, the names and order of type arguments are not significant, so the implementation is allowed to rename these, provided the rest of the signature still uses the type arguments in the same way. More precisely, it must be possible in the implementation to

- reorder the type arguments and
- consistently rename the type arguments in the entire signature of the implementation declaration

such that the resulting type arguments and the types of in- and out-parameters are the same as in the procedure declaration. Second, the names of the in- and out-parameters are not significant, so the implementation is allowed to rename these parameters, provided the names provided are distinct (from each other in the implementation declaration). The collection of multiple variables into a single $IdsType$ declaration is also not significant; for example,

$$x, y\colon \mathbf{int}$$

and

$$x\colon \mathbf{int},\ y\colon \mathbf{int}$$

are treated in the same way. Third, the implementation signature does not contain **where** clauses; these are simply acquired from the procedure declaration (like the procedure specification).

A program can contain any number (zero or more) of implementations per procedure. The correctness of an implementation is considered separately from any other implementations given. For convenience, one implementation can be declared as part of the procedure declaration, as shown above in the second form of the **procedure** declaration. That is,

$$\mathbf{procedure}\ P\langle\alpha\rangle(ins)\ \mathbf{returns}\ (outs)\ spec\ \{\ body\ \}$$

is a shorthand for

$$\mathbf{procedure}\ P\langle\alpha\rangle(ins)\ \mathbf{returns}\ (outs);\ spec$$
$$\mathbf{implementation}\ P\langle\alpha\rangle(ins')\ \mathbf{returns}\ (outs')\ \{\ body\ \}$$

where ins' and $outs'$ are ins and $outs$ but with any **where** clauses dropped.

8.1 Caller and callee traces: Motivation

Recall, caller traces are used to define procedure calls, whereas callee traces are used to constrain implementations. It is well-known that one achieves sound modular verification by verifying every assumption. For procedures, this means verifying that callers establish every precondition assumed by the implementation, and verifying that the implementation establishes every postcondition assumed by the callers [8]. This soundness criterion suggests that caller traces and callee traces ought to be the same. Perhaps unexpectedly, this is not the case in Boogie.

Boogie is a language for expressing verification conditions. When designing what verification conditions to prescribe, the user of Boogie has the choice of relegating some proof obligations to meta proofs that are carried out by other means. For example, in a source-language that avoids dangling pointers, the heap maintains the property that

allocated objects can only reach other allocated objects. This can be modeled in Boogie by declaring a predicate like:

$$\textbf{function } \mathit{WellFormed}(\mathit{heap}\colon \mathit{HeapType}) \textbf{ returns } (\textbf{bool});$$

and giving a number of axioms that define $\mathit{WellFormed}$'s properties of interest. The correctness of these axioms—which, to establish, requires a detailed semantics of the source-language operations and type system—can be argued once and for all, along with the rest of the proof that the translation from the source language into Boogie is correct. There is no need to include such a meta-level proof obligation in the verification conditions expressed in Boogie. Rather, one would simply like to assume $\mathit{WellFormed}(\mathit{Heap})$ to hold during various points of the program execution. To prescribe the property to hold on entry and exit of every procedure, it is possible to place an **assume** statement (see Section 9.2) immediately inside every implementation and immediately after each call. However, Boogie offers the cleaner and less unwieldy solution of declaring $\mathit{WellFormed}(\mathit{Heap})$ as a free pre- and postcondition of every procedure. A free precondition is assumed by the implementation, but not checked at call sites, and a free postcondition is assumed upon return from calls, but is not checked on exit from implementations, creating a distinction between caller traces and callee traces.

When parameters or pre/post-states are involved, the flexibility of caller/callee trace sets becomes even more pronounced. For example, when modeling a source language without explicit deallocation of storage, every procedure invocation satisfies a postcondition like:

$$(\forall\, o\colon \mathit{Ref} \bullet \textbf{old}(\mathit{Heap})[o, \mathit{alloc}] \Rightarrow \mathit{Heap}[o, \mathit{alloc}])$$

An effective use of Boogie would declare this postcondition to be free. The alternative of checking this postcondition of every implementation is unnecessary, because the source language offers no way for a source-language program to violate the property. And the alternative of using **assume** statements after each call is unwieldly, because then one has to save the value of the heap before the call and then expression the assumption after the call in terms of the saved heap and the current heap.

8.2 Caller and callee traces: Definitions

Consider any procedure P in a program. In the following, let σ and τ range over states whose domains consist of the global variables of the program, the **old** versions of those variables, and the in- and out-parameters of P. Let X range over any finite and possibly empty sequence of states, and let Y range over any infinite sequence of states. Furthermore, we restrict the ranges of σ, τ, X, and Y so that every state:

V0 maps variables to values that are consistent with the declared types of these variables, and

V1 is consistent with the declared axioms in the program.

Note, if the program's axioms are inconsistent—for example, if they contain **axiom false**;—then σ, τ, X, and Y have empty ranges, so the trace sets defined below will all be empty.

A general recipe for defining traces from procedure P is the following:

- All traces σX, $\sigma X \lightning$, and σY where
  - **A0** for some **requires** R in the specification of P, $R(\sigma)$ is false, or
  - **A1** for some **where** A among the in-parameters of P, $A(\sigma)$ is false, or
  - **A2** for some **where** B among the out-parameters of P, $B(\sigma)$ is false, or
  - **A3** for some **where** G among the global variables of the program, $G(\sigma)$ is false.
- All traces $\sigma X \tau$ where
  - **B0** for every global variable g, $g(\sigma) = g_{\mathbf{old}}(\sigma)$, and
  - **B1** for every **requires** R in the specification of P, $R(\sigma)$ is true, and
  - **B2** for every **where** A among the in-parameters of P, $A(\sigma)$ is true, and
  - **B3** for every **where** B among the out-parameters of P, $B(\sigma)$ is true, and
  - **B4** for every **where** G among the global variables, $G(\sigma)$ is true, and
  - **B5** for every **ensures** Q in the specification of P, $Q(\tau)$ is true, where, here and throughout, any global variable g (whether or not it occurs in a modifies clause) occurring in an **old** expression in Q has the value of the fictitious variable $g_{\mathbf{old}}$ in τ, and
  - **B6** for every global variable g, $g_{\mathbf{old}}(\sigma) = g_{\mathbf{old}}(\tau)$, and
  - **B7** for every in-parameter a of P, $a(\sigma) = a(\tau)$, and
  - **B8** for every global variable g, either g appears in a **modifies** clause of P or $g(\sigma) = g(\tau)$, and
  - **B9** for every **where** B among the out-parameters of P, $B(\tau)$ is true, and
  - **B10** for every **where** G among those global variables that appear in the **modifies** clause, $G(\tau)$ is true.
- All traces σY where
  - **C0** for every global variable g, $g(\sigma) = g_{\mathbf{old}}(\sigma)$, and
  - **C1** for every **requires** R in the specification of P, $R(\sigma)$ is true, and
  - **C2** for every **where** A among the in-parameters of P, $A(\sigma)$ is true, and
  - **C3** for every **where** B among the out-parameters of P, $B(\sigma)$ is true, and
  - **C4** for every **where** G among the global variables, $G(\sigma)$ is true.

From this general recipe, we know the caller traces and callee traces of P. The *caller traces* of P are those traces prescribed by the following alteration of the general recipe:

- **where** clauses of in-parameters are ignored; that is, A1, B2, and C2 are dropped, and
- **where** clauses of out-parameters are applied only in the post-state; that is, A2, B3, and C3 are dropped (but B9 remains), and
- **where** clauses of global variables are applied only in the post-state; that is, A3, B4, and C4 are dropped (but B10 remains), and
- free preconditions are ignored; that is, A0, B1, and C1 consider only checked preconditions, and
- free modifies clauses are ignored; that is, B8 considers only checked modifies clauses.

The *callee traces* of P are those traces prescribed by the following alteration of the general recipe:

- **where** clauses are applied only in the pre-state; that is, B9 and B10 are ignored, and

– free postconditions are ignored; that is, B5 considers only checked postconditions.

Intuitively, these alterations say that callers are responsible only for establishing checked (not free) preconditions and only need to be prepared to handle changes prescribed by checked (not free) modifies claues. Dually, implementations do not need to establish free postconditions. Similarly, **where** clauses are like free preconditions and (for out-parameters and modified global variables only) free postconditions.

Note that if the Boogie program contains no **where** clauses and no free specification clauses, then the caller traces of a procedure are trivially the same as its callee traces.

8.3 Implementation traces

Each implementation of a procedure defines a set of *implementation traces*. These traces stem from the execution traces defined by the implementation's body, after an initial renaming of the parameters.

Consider an implementation

$$\textbf{implementation}\ P\langle\alpha'\rangle(ins')\ \textbf{returns}\ (outs')\ body$$

where P is defined as follows:

$$\textbf{procedure}\ P\langle\alpha\rangle(ins)\ \textbf{returns}\ (outs);\ spec$$

The implementation traces of this implementation are all the traces $\sigma\,\tau Z$, where

– the domain of σ consists of the global variables of the program, a fictitious variable $g_{\mathbf{old}}$ for every global variable g, and the in- and out-parameters of the *declaration* of P, and
– the domain of τ consists of the global variables, a fictitious variable $g_{\mathbf{old}}$ for every global variable g, the in- and out-parameters of the *implementation* of P, and the local variables of the body of the implementation, and
– σ and τ agree on the values of global variables, agree on the values of the **old** variables, and agree on the values of corresponding parameters, and
– Z denotes any (empty, finite, infinite) state sequence optionally followed by $\lightning$, and
– τZ is an execution trace of $body$.

As we will see in the next section, the traces of the body are defined so that each of their states satisfies V0 and V1 (see Section 8.2); consequently, because of the likeness of σ and τ, state σ also satisfies V0 and V1.

As stated above, an implementation is correct if its implementation traces form a subset of the procedure's callee traces. Note that the definition of implementation traces does not mention the procedure's pre- and postconditions, for example. However, callee traces that begin in a state where a precondition does not hold are subsequently unconstrained; thus, a correct implementation is allowed to proceed in any which way from such pre-states. Also, finite callee traces that start in states satisfying all preconditions have the property that their initial and final states are related by the postconditions; thus, the terminating traces of a correct implementation are constrained similarly.

Modifies clauses are enforced syntactically: in the body of an implementation, only those global variables appearing in the procedure's (free and checked) modifies clauses are allowed to be modified.

$$
\begin{array}{rcl}
\mathit{Body} & ::= & \{\ \mathit{LocalVarDecl}^*\ \mathit{StmtList}\ \} \\
\mathit{LocalVarDecl} & ::= & \textbf{var}\ \mathit{Attribute}^*\ \mathit{IdsTypeWhere}\text{,}^+\ ; \\
\mathit{StmtList} & ::= & \mathit{LStmt}^*\ \mathit{LEmpty}^? \\
\mathit{LStmt} & ::= & \mathit{Stmt} \mid \mathit{Id} : \mathit{LStmt} \\
\mathit{LEmpty} & ::= & \mathit{Id} : \mathit{LEmpty}^? \\
\mathit{Stmt} & ::= & \textbf{assert}\ \mathit{Attribute}^*\ \mathit{Expr}\ ; \\
 & \mid & \textbf{assume}\ \mathit{Attribute}^*\ \mathit{Expr}\ ; \\
 & \mid & \textbf{havoc}\ \mathit{Id}\text{,}^+\ ; \\
 & \mid & \mathit{Lhs}\text{,}^+ := \mathit{Expr}\text{,}^+\ ; \\
 & \mid & \textbf{call}\ \mathit{CallLhs}^?\ \mathit{Id}\ (\ \mathit{Expr}\text{,}^*\ )\ ; \\
 & \mid & \textbf{call forall}\ \mathit{Id}\ (\ \mathit{WildcardExpr}\text{,}^*\ )\ ; \\
 & \mid & \mathit{IfStmt} \\
 & \mid & \textbf{while}\ (\ \mathit{WildcardExpr}\ )\ \mathit{LoopInv}^*\ \mathit{BlockStmt} \\
 & \mid & \textbf{break}\ \mathit{Id}^?\ ; \\
 & \mid & \textbf{return}\ ; \\
 & \mid & \textbf{goto}\ \mathit{Id}\text{,}^+\ ; \\
\mathit{Lhs} & ::= & \mathit{Id}\ \mathit{MapSelect}^* \\
\mathit{MapSelect} & ::= & [\ \mathit{Expr}\text{,}^+\ ] \\
\mathit{CallLhs} & ::= & \mathit{Id}\text{,}^+ := \\
\mathit{WildcardExpr} & ::= & \mathit{Expr} \mid * \\
\mathit{BlockStmt} & ::= & \{\ \mathit{StmtList}\ \} \\
\mathit{IfStmt} & ::= & \textbf{if}\ (\ \mathit{WildcardExpr}\ )\ \mathit{BlockStmt}\ \mathit{Else}^? \\
\mathit{Else} & ::= & \textbf{else}\ \mathit{BlockStmt} \mid \textbf{else}\ \mathit{IfStmt} \\
\mathit{LoopInv} & ::= & \textbf{free}^?\ \textbf{invariant}\ \mathit{Attribute}^*\ \mathit{Expr}\ ;
\end{array}
$$

Fig. 2. Boogie's statement grammar.

9 Statements

Boogie statements are used in procedure implementations and provide a way to prescribe execution traces. A procedure implementation is correct if and only if its finite traces form a subset of the procedure's callee traces. The statements include state mutations, check and assume operations, conditional and blind (demonic) nondeterministic control flow (possibly irreducible), iteration, and procedural abstraction. The statements follow the grammar in Fig. 2.

Throughout this section, states implicitly range over those that satisfy V0 and V1 (see Section 8.2).

9.0 Implementation body

A body starts with a number of local-variable declarations. The names of the variables declared must be distinct from each other and distinct from the implementation's in- and out-parameters. The expression given by a *WhereClause* appearing among the local variables must be of type boolean and can refer to any constant, function, global variable, in- and out-parameter, and local variable. The *Type* and *WhereClause* of an *IdsTypeWhere* apply to each of the *Id*'s declared, as if each *Id* were introduced in

a separate **var** declaration. The attributes of the declaration apply to each of the Id's introduced.

The execution traces of a body $\{\ locals\ stmts\ \}$ consists of all traces σZ such that:

- the domain of σ consists of the global variables of the program, a fictitious variable $g_{\mathbf{old}}$ for every global variable g, the in- and out-parameters of the implementation declaration, and the local variables declared in $locals$, and
- for every **where** L among the local variables, $L(\sigma)$ is true, and
- Z denotes any (empty, finite, infinite) state sequence optionally followed by $\lightning$, and
- σZ is an execution trace of the $start$ label of

  $start$: transform$(\emptyset)[\![stmts]\!]$ **return**;

  where $start$ is a fresh label symbol, $\emptyset$ denotes the empty map (see Section 9.5), and transform is a function defined below; more precisely, $\sigma Z \in Traces(start)$, where $Traces$ is defined in Section 9.1 below.

The semantics of an implementation body's statements is given by trace sets via a basic-block transformation of the statement. The transform function rewrites statements to form a sequence of *basic blocks*, each of which has the form

$$label\colon SimpleStmt^*\ Goto$$

where $SimpleStmt$ is a subset of the statements in Fig. 2 and $Goto$ is either a **goto** statement or a **return** statement. The $SimpleStmt$ statements include assert, assume, havoc, and call statements, as well as assignment statements whose left-hand side is a list of simple variables.

Function transform is applied only in contexts where there is a preceding label (without an intervening $Goto$) and a succeeding $Goto$ (without an intervening label); from this condition and the definition of transform below, it follows that the result of $start$: transform$(\emptyset)[\![stmts]\!]$ **return**; is indeed a sequence of basic blocks.

By the way, note that a sequence of basic blocks is a special case of $StmtList$ in the Boogie statement grammar.

9.1 Basic blocks

The definition of the semantics of basic blocks rests on the definition of several other operators and functions.

The concatenation of two trace sets $\mathcal{A}$ and $\mathcal{B}$, written $\mathcal{A} \fatsemi \mathcal{B}$, is defined as follows:

$$\begin{array}{rcl} \mathcal{A} \fatsemi \mathcal{B} & = & \{\ X\lightning \mid X\lightning \in \mathcal{A}\ \} \cup \\ & & \{\ Y \mid Y \in \mathcal{A}\ \} \cup \\ & & \{\ X\sigma Z \mid X\sigma \in \mathcal{A} \text{ and } \sigma Z \in \mathcal{B}\ \} \end{array}$$

where X ranges over possibly empty, finite state sequences, Y ranges over infinite state sequences, and Z ranges over any (empty, finite, infinite) state sequence optionally followed by $\lightning$. Note that concatenation of traces fuses, as opposed to repeats, the last state of one trace with the first state of the next.

Function stmt concatenates the trace sets of a sequence of statements. For any statement S and possibly empty sequence of statements $stmts$:

$$\begin{array}{l} \mathsf{stmt}[\![\,]\!] \;=\; \{\, \sigma\sigma \;\mid\; \sigma \text{ is any state satisfying V0 and V1 } \} \\ \mathsf{stmt}[\![stmts\ S]\!] \;=\; \mathsf{stmt}[\![stmts]\!] \fatsemi \mathsf{traces}[\![S]\!] \end{array}$$

where $\mathsf{traces}[\![S]\!]$ denotes the trace set of S. Note that all traces returned by stmt have length at least 2. This is important below, because it ensures that the execution traces of infinite loops (like ℓ: **goto** ℓ;) are infinite.

For any label ℓ among the basic blocks of a procedure's transformed implementation, $Traces(\ell)$ denotes the execution traces that result from basic block ℓ forward. Function $Traces$ is defined as the least fixpoint to the system of equations obtained as follows for each label ℓ:

$$Traces(\ell) = \begin{cases} \mathsf{stmt}[\![stmts]\!] & \text{for } \ell\text{: } stmts \textbf{ return}; \\ \mathsf{stmt}[\![stmts]\!] \fatsemi \bigcup_{\pounds \in succs} Traces(\pounds) & \text{for } \ell\text{: } stmts \textbf{ goto } succs; \end{cases}$$

The remaining subsections consider the various statements, defining trace sets for the simple statements and decomposing the other statements using the transform function.

9.2 Assertions and assumptions

The assert statement gives an expression that holds in every correct trace. The assume statement gives an expression that holds in every feasible trace. The expressions given by these statements must be boolean.

The assert statement is used to prescribe a check. For example, a source-language assignment x = y / z; may be translated into the Boogie statements:

$$\textbf{assert } z \neq 0;\ \ x := y/z;$$

As another example, the array assignment $a[i] := e;$ in a language with co-variant array types (like in Java or C#) is modeled by prefixing the actual state update with assertions like:

$$\begin{array}{ll} \textbf{assert } a \neq \mathit{null}; & \text{// check receiver to be non-null} \\ \textbf{assert } 0 \leqslant i; & \text{// check lower bound of index} \\ \textbf{assert } i < \mathit{ArrayLength}(a); & \text{// check upper bound of index} \\ \textbf{assert } \mathit{type}(e) <: \mathit{elementType}(\mathit{type}(a)); & \text{// check co-variance} \end{array}$$

where the source-language **null** is modeled by:

$$\textbf{const } \mathit{null}\text{: } \mathit{Ref};$$

In order to report a meaningful error message to the user of the source language, the source-language verification tool needs to remember why each of these assertions was generated in the translation to Boogie and then simply maps the failure of any of asserts to a message reflecting that reason.

An assume statement is used to introduce an assumption in the program to be verified, with the effect of rendering infeasible those execution traces where the assumption

would not hold. For example, the function $WellFormed$ in Section 8.1 that expresses the well-formedness of the heap in a program can be assumed after each update of the heap. For example,

$$Heap[this, C.data] := 5; \ \ \mathbf{assume}\ WellFormed(Heap);$$

Such use of $WellFormed$ affords the possibility of axiomatizing only some of the properties of $WellFormed$; that is, using an implication in

$$\mathbf{axiom}\ (\forall\, heap\colon HeapType \bullet \ WellFormed(heap) \ \Rightarrow \ \ldots\ );$$

instead of an if-and-only-if (since there would never be a need to prove $WellFormed$ in the program).

In important special case of the assume statement is $\mathbf{assume\ false};$, which has the effect of not doing any further checking along that program path. This is useful, for example, when debugging a Boogie program: to prevent the program verifier from spending time trying to prove proof obligations beyond some particular program point, simply insert an $\mathbf{assume\ false};$ at that point, which has the effect of rendering infeasible those execution paths that would go through that point.

Other important uses of assume statements are explained in the subsections dealing with **if** statements (Section 9.7) and the havoc statement (Section 9.4).

Formally, the trace sets of these statements are:

$$\begin{array}{l}\mathsf{traces}[\![\mathbf{assert}\ E;]\!] \ = \\ \quad \{\, \sigma \ \mid\ E(\sigma) \text{ is true } \} \ \cup\ \{\, \sigma\lightning \ \mid\ E(\sigma) \text{ is false } \} \\ \mathsf{traces}[\![\mathbf{assume}\ E;]\!] \ = \\ \quad \{\, \sigma \ \mid\ E(\sigma) \text{ is true } \}\end{array}$$

To understand how trace sets are built up, it is instructive to consider the effect of composing the assert and assume statements with other statements. For example, the trace set $\mathsf{stmt}[\![\mathbf{assert}\ 0 \leqslant x; \mathbf{assume}\ 0 \leqslant y;]\!]$ consists of the traces $\sigma\sigma\lightning$, where $x(\sigma) < 0$ and the traces $\sigma\sigma$ where both $0 \leqslant x(\sigma)$ and $0 \leqslant y(\sigma)$. However, the traces $\sigma\sigma$ where $0 \leqslant x(\sigma)$ and $y(\sigma) < 0$ are not included at all. Attempting an operational description of a "run" of these two statements, one could say:

0. Evaluate $0 \leqslant x$. If it holds, continue the run; otherwise, go wrong.
1. Evaluate $0 \leqslant y$. If it holds, continue the run, which in the case of just these two statements means terminate. If $0 \leqslant y$ does not hold, then shrivel up the entire execution trace, as if the run never happened.

From this step-by-step run point of view, this "shriveling up" business removes the steps "already taken" if it arrives at an assume statement whose condition does not hold. This may seem magical, but all it means is that this trace is not feasible; it's just that the infeasibility of it is discovered only after some other statements have already happened.

Note, however, that an assume statement cannot undo going wrong. For example, $\mathsf{stmt}[\![\mathbf{assert}\ x < 100; \mathbf{assume}\ x == 0;]\!]$ contains the execution traces $\sigma\sigma$ where $x(\sigma) = 0$, which correspond to those feasible traces where the assertion holds. But it also contains the execution traces $\sigma\sigma\lightning$ where $100 \leqslant x(\sigma)$, which correspond to the runs that go wrong and do not even make it to the assume statement.

9.3 Assignments

An assignment changes the value of one or more mutable variables in parallel. The number of Lhs's on the left-hand side must equal the number of $Expr$'s on the right-hand side. The type of an Lhs and the type of the corresponding $Expr$ must be equal. The Id in an Lhs must denote a mutable variable, and these Id's must all be distinct. Note that in-parameters are not mutable variables, so they cannot be assigned to.

For example, the simple assignment statement:

$$x := x + 1;$$

increments the value of variable x by 1. The assignment statement:

$$a[i] := 12;$$

changes what map a maps i to, making the new value map i to 12. The parallel assignment statement:

$$x, y := y, x;$$

swaps the values of variables x and y. The statement:

$$x, a[i] := x + 1, x;$$

increments x and sets $a[i]$ to the old value of x. The statement:

$$a[i], a[j] := a[j], a[i];$$

is not allowed, because the Id's in the left-hand side are not distinct.

The trace set of a simple assignment statement where all left-hand sides are Id's is defined as follows:

$$\begin{array}{l} \mathsf{traces}[\![\overline{x} := \overline{E};]\!] \ = \\ \quad \{\, \sigma\tau \mid \sigma \text{ and } \tau \text{ have the same domains and agree on the values} \\ \qquad\quad \text{of all variables except possible those in } \overline{x}, \text{ and for each} \\ \qquad\quad x \text{ in } \overline{x} \text{ and corresponding } E \text{ in } \overline{E},\ x(\tau) = E(\sigma) \ \ \} \end{array}$$

If the left-hand side is not just Id's, then the semantics is defined in terms of the semantics of simpler assignment statements. For any list $\overline{x}$ of Id's; any map variable or map-selection expression A; any lists $\overline{J}$, $\overline{X}$, and $\overline{F}$ of expressions; any list $\overline{L}$ of Rhs's; and any expression E:

$$\begin{array}{l} \mathsf{traces}[\![\overline{x}, A[\overline{J}], \overline{L} \ := \ \overline{X}, E, \overline{F};]\!] \ = \\ \quad \mathsf{traces}[\![\overline{x}, A, \overline{L} \ := \ \overline{X}, A[\overline{J} := E], \overline{F};]\!] \end{array}$$

For example, the semantics of the statements:

$$\begin{array}{l} a[j] := E; \\ b[i][m, n] := F; \end{array}$$

is defined as having the same semantics as:

$$\begin{array}{l} a := a[j := E]; \\ b := b[i := b[i][m, n := F]]; \end{array}$$

Note that **where** clauses do not play a role for assignment statements. In particular, an assignment statement can set a variable to a value that does not satisfy its **where** clause. For example,

var $volume$: **int where** $0 \leqslant volume \wedge volume \leqslant 10$;
$x := 11$;

sets $volume$ to 11, despite the fact that the **where** clause suggests $volume$ is intended to be between 0 and 10. **where** clauses apply only in places where a variable gets an arbitrary value; by not applying them at assignment statements, Boogie provides the ability to use a sequence of statements that (perhaps temporarily) violates **where** conditions.

To explicitly say that a condition holds after an assignment, use an assume statement, as in:

$Heap[this, C.data] := Heap[this, C.data] + 1$;
assume $WellFormed(Heap)$;

9.4 Havoc

The havoc statement assigns arbitrary (blindly chosen) values to a set of variables. Each Id listed in the statement must denote a mutable variable. The values assigned are arbitrary, but within limits: the values are drawn from the types of the respective variables (in accordance with V0), the values satisfy the program's axioms (in accordance with V1), and the values satisfy the **where** clauses of the variables.

For example, for variables x and y declared as:

var x: **int where** $0 \leqslant x$;
var y: **int where** $0 \leqslant y \wedge y < x$;

the statement **havoc** x, y; sets x and y to non-negative integers where x is strictly larger than y. Note how, in this example, one gets the same result with the sequence of statements:

havoc x; **havoc** y;

Although the **havoc** x; by itself may set x to 0, there is no feasible trace that passes the **havoc** y; with such a value for x. The sequence:

havoc y; **havoc** x;

is different, because it first sets y to a value between 0 and less than x, and then sets x to a non-negative integer without further regard for the value of y.

An important and frequently occuring idiom is to follow a **havoc** with an **assume**, which has the effect of taming the nondeterminism offered by the **havoc**. For example,

havoc a, b; **assume** $3 * a + b == 25$;

will have the effect of setting a and b to an arbitrary solution of the given equation. Another example is:

havoc c; **assume** $\neg Heap[c, alloc]$; $Heap[c, alloc] := \mathbf{true}$;

which sets c to a unallocated reference (one whose $alloc$ field is **false**) and then allocates it (by setting its $alloc$ field to **true**).

Some variables are intended always to satisfy some condition. An example in Section 9.2 shows how the heap variable is intended to satisfy the predicate $WellFormed$. Such conditions are often appropriate in **where** clauses. For example,

var $Heap$: $HeapType$ **where** $WellFormed(Heap)$;

makes sure that all arbitrary values chosen for $Heap$ satisfy $WellFormed$. (As noted in Section 9.3, assignment statements still need to be followed by an **assume** statement if it is desirable to know this predicate to hold after an explicit update.)

The havoc statement works only on entire variables. To use an arbitrary value in another context (for example, assigning a map element or passing a argument to function), use a temporary variable. For example,

var tmp: **int**;
...
$Heap[this, C.data] := tmp$;

The trace set of **havoc** on a list of variables $\overline{x}$ is defined as:

$\mathsf{traces}[\![\mathbf{havoc}\ \overline{x};]\!] =$
$\{\ \sigma\tau$ | σ and τ have the same domains and agree on the values of all variables except possible those in $\overline{x}$, and for every **where** P among the declarations of the variables $\overline{x}$, $P(\tau)$ $\}$

Note that from some states σ, there may not be any τ that satisfies V0, V1, and the **where** clauses of the havocked variable. For example, with x and y declared with the **where** clauses above,

$x := 0$; **havoc** y;

is tantamount to **assume false**;.

9.5 Label statements and jumps

A label statement is used to give a name to a program point. The label can be used as a target in a **goto** statement and can be used in the **break** statement to indicate an enclosing statement. Labels used within the same implementation body must be distinct.

Label statements occur in the $LStmt$ and $LEmpty$ non-terminals of the grammar in Fig. 2, and together these define three kinds of label statements: those that label other label statements, those that label non-label statements (that is, non-terminal $Stmt$), and those that label no statement at all.

A **goto** statement transfers control flow to any one of the specified labels. The choice between the labels is done blindly. The labels listed in the **goto** statement must all be defined, as label statements, somewhere within the same implementation body.

A **break** statement transfers control flow to the point immediately following a designated enclosing statement. Any **break** statement that omits the label (that is, the

Id in the grammar) must occur as a substatement of the body of a **while** statement; control is transferred to the point immediately following the closest enclosing **while** statement. Any **break** statement that specifies a label must occur as a substatement of the *LStmt* that defines that label; control is transferred to the point immediately following the so-labeled enclosing *LStmt* statement.

Note that the **goto** statement can give rise to any kind of control flow, including loops and irreducible control flow. In contrast, the **break** statement only gives rise to structured, forward jumps.

Here are some examples. The following while loop stops iterating if the value X is found in the map:

```
i := 0;
while (i < N) {
    if (a[i] == X) { break; }
    i := i + 1;
}
```

The **break** statement in this loop can be replaced by a **goto**:

```
i := 0;
while (i < N) {
    if (a[i] == X) { goto Done; }
    i := i + 1;
}
Done:
```

The **while** statement itself can also be replaced by a label and a **goto**:

```
i := 0;
Head:
if (i < N) {
    if (a[i] == X) { goto Done; }
    i := i + 1;
    goto Head;
}
Done:
```

Although less common, the **break** statement can also be used to break out of an **if** statement; here is the same program again:

```
i := 0;
Head:
if (i < N) {
    if (a[i] == X) { break Head; }
    i := i + 1;
    goto Head;
}
```

To break out of a while loop that is not the innermost **while** statement that encloses the **break**, one can use a label, as in the following search of two-dimensional map:

```
i := 0;
Outer:
while (i < N) {
    j := 0;
    while (j < i) {
        if (m[i, j] == X) { break Outer; }
        j := j + 1;
    }
    i := i + 1;
}
```

Finally, the **break** in the following statement is a no-op:

```
L: break L;
```

and could just as well have been written as, for example:

```
L: assert true;
```

The semantics of labels and jumps is defined as part of the transformation into basic blocks. The map argument to transform is used to keep of exit labels. In the following, *ls* denotes an *LStmt* and *le* denotes an *LEmpty* (see grammar in Fig. 2), η denotes any map, and *Done* and *Unreachable* denote fresh labels:

```
transform(η)⟦ℓ: ls⟧ =
    goto ℓ;
    ℓ: transform(η[ℓ ↦ Done])⟦ls⟧ goto Done;
    Done:
transform(η)⟦ℓ: le⟧ =
    goto ℓ;
    ℓ: transform(η)⟦le⟧
transform(η)⟦goto L;⟧ =
    goto L;  Unreachable:
transform(η)⟦break L;⟧ =
    goto η⟦L⟧;  Unreachable:
transform(η)⟦break;⟧ =
    goto η⟦⋆⟧;  Unreachable:
```

The special value $\star$ denotes the closest enclosing while loop and gets introduced into the domain of the map η in Section 9.8. Note, the at-first-sight gratuitous labels and goto's in these definitions ensure that recursive calls to transform are applied in an appropriate context, so that the implementation body is properly transformed into a sequence of basic blocks (*cf.* Section 9.0).

9.6 Return statements

A **return** statement terminates any execution trace that reaches it. As shown in Section 9.0, there is also an implicit **return** statement at the end of an implementation body. For any map η and with $Unreachable$ denoting a fresh label,

$$
\begin{array}{l}
\mathsf{transform}(\eta)[\![\mathbf{return};]\!] \ = \\
\quad \mathbf{return};\ \ Unreachable\colon
\end{array}
$$

9.7 If statements

There are two kinds of **if** statements, those that use a boolean expression to choose between their alternatives and those that blindly choose between the alternatives. The former uses an expression as the $WildcardExpr$, and that expression must be boolean; the latter uses a $*$ as the $WildcardExpr$.

The semantics of **if** statements is defined as follows:

$$
\begin{array}{l}
\mathsf{transform}(\eta)[\![\mathbf{if}\ (we)\ Thn]\!] \ = \\
\quad \mathsf{transform}(\eta)[\![\mathbf{if}\ (we)\ Thn\ \mathbf{else}\ \{\ \}]\!] \\
\mathsf{transform}(\eta)[\![\mathbf{if}\ (*)\ Thn\ \mathbf{else}\ Els]\!] \ = \\
\quad \mathbf{goto}\ L0, L1; \\
\quad L0\colon\ \mathsf{transform}(\eta)[\![Thn]\!];\ \ \mathbf{goto}\ Done; \\
\quad L1\colon\ \mathsf{transform}(\eta)[\![Els]\!];\ \ \mathbf{goto}\ Done; \\
\quad Done\colon \\
\mathsf{transform}(\eta)[\![\mathbf{if}\ (E)\ Thn\ \mathbf{else}\ Els]\!] \ = \\
\quad \mathbf{goto}\ L0, L1; \\
\quad L0\colon\ \mathbf{assume}\ E;\ \ \mathsf{transform}(\eta)[\![Thn]\!];\ \ \mathbf{goto}\ Done; \\
\quad L1\colon\ \mathbf{assume}\ \neg E;\ \ \mathsf{transform}(\eta)[\![Els]\!];\ \ \mathbf{goto}\ Done; \\
\quad Done\colon
\end{array}
$$

where η is any map and $L0$, $L1$, $Done$ are fresh labels.

9.8 While loops

Boogie supports the usual while loop as well as a while loop that exits after an arbitrary number of iterations. Both kinds support loop invariants. For any list $invs$ of invariant declarations, the semantics of while loops is defined as follows:

$\mathsf{transform}(\eta)[\![\mathbf{while}\ (*)\ invs\ S]\!] =$
  $\mathbf{goto}\ Head;$
  $Head:\ \mathsf{loopinv}[\![invs]\!]\ \mathbf{goto}\ Body, Done;$
  $Body:\ \mathsf{transform}(\eta[\star \mapsto Done])[\![S]\!];\ \mathbf{goto}\ Head;$
  $Done:$
$\mathsf{transform}(\eta)[\![\mathbf{while}\ (E)\ invs\ S]\!] =$
  $\mathbf{goto}\ Head;$
  $Head:\ \mathsf{loopinv}[\![invs]\!]\ \mathbf{goto}\ Body, GuardedDone;$
  $Body:\ \mathbf{assume}\ E;\ \mathsf{transform}(\eta[\star \mapsto Done])[\![S]\!];\ \mathbf{goto}\ Head;$
  $GuardedDone:\ \mathbf{assume}\ \neg E;\ \mathbf{goto}\ Done;$
  $Done:$
$\mathsf{loopinv}[\![\ ]\!] =$
  // nothing
$\mathsf{loopinv}[\![\mathbf{invariant}\ E;\ invs]\!] =$
  $\mathbf{assert}\ E;\ \mathsf{loopinv}[\![invs]\!]$
$\mathsf{loopinv}[\![\mathbf{free\ invariant}\ E;\ invs]\!] =$
  $\mathbf{assume}\ E;\ \mathsf{loopinv}[\![invs]\!]$

where η is any map and $Head$, $Body$, $GuardedDone$, and $Done$ are fresh labels.

There is a subtle difference between the use of the **free** keyword in loop invariants compared to the use of **free** in procedure specifications. To see the difference, think of a procedure specification as a contract between two parties, the implementation and the call site. A checked specification clause is assumed (as with an **assume** statement) by one party and checked (as with an **assert** statement) for the other. A free specification clause is assumed by one party and *ignored* by the other. For example, all preconditions are assumed by the implementation, checked preconditions are checked at the call site, and free preconditions are ignored by the call site.

Likewise, a loop invariant is a contract between two parties, the iterations of the past and the iterations of the future. In standard Hoare logic, the "past" party consists of the program point immediately before the loop and the program point at the very end of the loop body; the "future" party consists of the program point at the very beginning of the loop body and the program point immediately following the loop. A checked loop invariant is checked for the "past" party and assumed by the "future" party. And here is the difference with free procedure-specification clauses: a free loop invariant is assumed by both parties. An example consequence of this subtle difference is that a precondition

free requires false;

will cause the verification of the implementation to succeed trivially), but does not affect callers. In contrast, listing

free invariant false;

first among the invariants of a loop will cause the verification of both loop parties to succeed trivially. This is because the free invariant is assumed just before the loop, and just before the other loop invariants if this one is declared first.

9.9 Call statements

A call statement stands for the caller traces defined by the called procedure, suitably parameterized for the actual in- and out-parameters. The Id mentioned in the call statement must refer to a procedure declared in the program. The number of $Expr$'s (the actual in-parameters) must agree with the number of formal in-parameters of the procedure, and the number of Id's in $CallLhs$ (the actual out-parameters) must agree with the number of formal out-parameters of the procedure. Each Id in the $CallLhs$ must denote a mutable variable, and these Id's must all be distinct. There must exist some way to instantiate the procedure's type arguments so that the types of the actual in- and out-parameters are the same as the types of the formal in- and out-parameters (*cf.* the type rule for function application in Section 5).

Consider a procedure P declared as follows:

$$\textbf{procedure}\ P\langle\alpha\rangle(ins)\ \textbf{returns}\ (outs);\ spec$$

Let $CallerTraces$ denote the set of caller traces for P. For any states σ and τ, define $Ins(\sigma,\tau)$ to hold when:

- the domains of σ and τ contain the same global variables and **old** variables, and
- for every global variable g, $g(\sigma) = g(\tau)$, and
- for every global variable g, $g_{\mathbf{old}}(\tau) = g(\tau)$, and
- for every $(a, E) \in (ins, \overline{E})$, $E(\sigma) = a(\tau)$.

Similarly, for any states σ and τ, define $Out(\tau, \upsilon)$ to hold when:

- the domains of σ and τ contain the same global variables and **old** variables, and
- for every global variable g, $g(\sigma) = g(\tau)$, and
- for every $(b, x) \in (outs, \overline{x})$, $b(\sigma) = x(\tau)$.

Then, the call statement is defined to have the following trace set:

$$\begin{array}{l}
\mathsf{traces}[\![\textbf{call}\ \overline{x} := P(\overline{E})]\!] \ = \\
\quad \{\, \sigma\tau X\!\not\downarrow \ \mid\ Ins(\sigma,\tau) \text{ and } \tau X\!\not\downarrow \in CallerTraces \,\} \ \cup \\
\quad \{\, \sigma\tau Y \ \mid\ Ins(\sigma,\tau) \text{ and } \tau Y \in CallerTraces \,\} \ \cup \\
\quad \{\, \sigma\tau X\upsilon\phi \ \mid\ Ins(\sigma,\tau) \text{ and } \tau X\upsilon \in CallerTraces \text{ and } Out(\upsilon,\phi), \\
\qquad\qquad \text{and } \sigma \text{ and } \phi \text{ have the same domains, and for every} \\
\qquad\qquad \text{global variable } g,\ g_{\mathbf{old}}(\sigma) = g_{\mathbf{old}}(\phi) \,\}
\end{array}$$

where X ranges over possibly empty, finite state sequences, and Y ranges over infinite state sequences.

9.10 Call-forall statements

The call-forall statement requires some motivation. In more advanced verifications, especially verifications authored by hand in Boogie, it is sometimes necessary to make use of a lemma. The lemma might help the program verifier along, perhaps with a subtle hint or perhaps with a deep property. For example, consider a piece of code A that establishes a property $P(x, y)$ and that is followed by a piece of code B whose correctness depends on starting with the property $Q(x, y)$. With the lemma $P(x, y) \Rightarrow Q(x, y)$, used at the program point between A and B, one can prove the correctness of the whole code sequence.

One way to introduce a lemma in a Boogie program is to use an **assert** statement. The program verifier gets the hint that the lemma is important (at the program point where the **assert** is placed) and is also faced with the obligation to prove the lemma. If the proof of the lemma is too difficult for the program verifier to tackle without further hints, or if the same lemma is used many times and one would like the program verifier to produce the proof just once, then one can also encode the lemma as a procedure. For example, with the procedure:

procedure $Lemma(x\colon X,\ y\colon Y)$;
  ensures $P(x, y) \Rightarrow Q(x, y)$;

any program point that wants to invoke the lemma simply calls the procedure:

call $Lemma(x, y)$;

A procedure, like $Lemma$, that has no checked modifies clause and no out-parameters is called a *lemma procedure*. A call to a lemma procedure does not change the program state. Indeed, the effect of a call to $Lemma$ is simply to assume the postcondition.

With a procedure that defines the lemma, the proof of the lemma now falls on the implementation of the procedure. (If no implementation is supplied, the lemma remains unproved, a conjectured lemma perhaps.)

Suppose the desired lemma has a form like:

$$(\forall\, x\colon X,\ y\colon Y \bullet P(x, y) \Rightarrow Q(x, y))$$

A lemma procedure that uses this property as its postcondition puts the burden on the procedure implementation to prove the implication for all x and y. A standard way of dealing with such a proof obligation is to start the proof with "Consider any arbitrary x and $y \ldots$", and it would be nice to let the implementation do the proof in this way. The implementation of $Lemma$ above comes tantalizingly close—it actually has to prove the postcondition for any values of in-parameters x and y, but its postcondition only gives the caller the ability to use the lemma for a particular x and y, namely whatever x and y are supplied at the call site.

Call-forall statement to the rescue. With a call-forall statement, a call site can invoke a lemma procedure for an arbitrary number of parameter values at the same time. This is indicated by giving a wildcard $*$ instead of an $Expr$ for any of in-parameters. For example, the call:

call forall $Lemma(*, *)$;

will have the same meaning as the statement:

$$\textbf{assume}\ (\forall\, x\colon X,\ y\colon Y \bullet\ P(x,y)\ \Rightarrow\ Q(x,y)\ );$$

yet an implementation of $Lemma$ still just needs to establish the postcondition for an arbitrary value of the in-parameters.

When a lemma procedure is invoked in a call-forall statement, the precondition of the procedure plays the role of an antecedent in the condition assumed. So the call-forall statement above would still have the same effect if the declaration of $Lemma$ were changed to:

$$\begin{array}{l}\textbf{procedure}\ Lemma(x\colon X,\ y\colon Y);\\ \quad\textbf{requires}\ P(x,y);\\ \quad\textbf{ensures}\ Q(x,y);\end{array}$$

Note, however, that there is a difference between a call-forall that supplies values for all in-parameters:

$$\textbf{call forall}\ Lemma(x,y);$$

and a regular call:

$$\textbf{call}\ Lemma(x,y);$$

because the regular call will enforce the precondition at the call site whereas the call-forall will just assume the lemma-procedure's postcondition if its precondition does. As an extreme example, any call-forall invocation of the following lemma procedure:

$$\begin{array}{l}\textbf{procedure}\ UselessLemma(x\colon \textbf{int}, y\colon \textbf{int}, z\colon \textbf{int}, n\colon \textbf{int});\\ \quad\textbf{requires false};\\ \quad\textbf{ensures}\ exp(x,n) + exp(y,n) \neq exp(z,n);\end{array}$$

is tantamount to $\textbf{assume true};$, whereas a regular call to it is tantamount to $\textbf{assert false};$.

Having given that motivation for call-forall statement, here are the rules for using it. The Id mentioned in the call-forall statement must refer to a lemma procedure declared in the program. The number of *WildcardExpr*'s (the actual in-parameters) must agree with the number of formal in-parameters of the procedure. There must exist some way to instantiate the procedure's type arguments so that the types of the actual in-parameters are the same as the types of the formal in-parameters (*cf.* the type rule for function application in Section 5), actual in-parameters given by the wildcard $*$ excluded.

Consider a lemma procedure P declared as follows:

$$\textbf{procedure}\ P\langle\alpha\rangle(ins);\ \ spec$$

and a call-forall statement:

$$\textbf{call forall}\ P(\overline{we});$$

Let $insExpr$ be the subset of ins for which $\overline{we}$ supplies an corresponding $Expr$, and let $insWild$ be the rest of ins but with **where** clauses removed, that is, the subset of ins (and dropping **where** clauses) for which $\overline{we}$ supplies the wildcard $*$. Let $\overline{\beta}$

be the subset of $\overline{\alpha}$ that is not mentioned among the types of $insExpr$. Let Pre be the conjunction of checked preconditions in $spec$, with the formal in-parameters of $insExpr$ replaced by the corresponding $Expr$ in $\overline{we}$. Let $Post$ be the conjunction of checked and free postconditions in $spec$, also with the formal in-parameters of $insExpr$ replaced by the corresponding $Expr$ in $\overline{we}$. The trace set of the call-forall statement is now defined as:

$$\begin{array}{l}\mathsf{traces}[\![\textbf{call forall } P(\overline{we});]\!] \ = \\ \quad \mathsf{traces}[\![\textbf{assume } (\ \forall\, \langle\beta\rangle\ ins\mathit{Wild} \bullet\ Pre\ \Rightarrow\ Post\ );]\!]\end{array}$$

where "$\langle\beta\rangle$" is dropped if β is empty, and where the entire **assume** is taken to be

$$\textbf{assume } Pre\ \Rightarrow\ Post;$$

if $ins\mathit{Wild}$ is empty.

Note that call-forall can be used with a mix of wildcards and in-parameter expressions. For example, if type Y is **int**, then:

$$\textbf{call forall } Lemma(*, 6);$$

has the same semantics as:

$$\textbf{assume } (\ \forall\, x{:}\ X \bullet\ P(x, 6)\ \Rightarrow\ Q(x, 6)\ );$$

10 Orders

Boogie supplies one partial-order operator, written $<:$, for each built-in and user-defined type. Being a partial order, $<:$ is reflexive, transitive, and antisymmetric. Additional properties of $<:$ for a given type can be axiomatized. For example,

$$\begin{array}{l}\textbf{const unique } puny{:}\ Wicket;\\ \textbf{axiom } (\ \forall\, w{:}\ Wicket \bullet\ puny <:\ w\ );\end{array}$$

defines $puny$ to be the smallest of all wickets.

Certain kinds of order specifications occur frequently and receive special syntax as part of **const** declarations. Not only do these declarations make order specifications more convenient, but they may also make it easier to take advantage of decision-procedure support of partial orders when the Boogie program is analyzed. The grammar is as follows:

$$\begin{array}{rcl} \mathit{OrderSpec} & ::= & \mathit{ParentInfo}^{?}\ \textbf{complete}^{?} \\ \mathit{ParentInfo} & ::= & <:\ \mathit{ParentEdge}\text{,}^{*} \\ \mathit{ParentEdge} & ::= & \textbf{unique}^{?}\ \mathit{Id} \end{array}$$

A constant n with the order specification "$<:\ ids$" says that the constants in ids are the immediate parents of n in the ordering. If $ParentInfo$ is supplied, it lists all immediate parents. For example,

$$\begin{array}{l}\textbf{const unique } a, b{:}\ Wicket;\\ \textbf{const unique } c{:}\ Wicket <:\ a, b;\end{array}$$

is an economical way to say:

$$\textbf{const unique } a, b, c\text{: } Wicket;$$
$$\textbf{axiom } (\,\forall\, w\text{: } Wicket \bullet\ c <\text{: } w \ \Rightarrow\ c == w \ \vee\ a <\text{: } w \ \vee\ b <\text{: } w\ );$$

The use of **const unique** in these declarations is usually desired, but order specifications can also be used with non-**unique** constants.

The $ParentInfo$ of an order specification always determines the complete set of immediate parents in the partial order. To say that a constant has no parents, leave the $ParentEdge$ list empty (but include the $<$: sign in the declaration, as required by $ParentInfo$). To omit a specification of immediate parents, omit the entire $ParentInfo$ clause.

Some children of a node are determined by other constant declarations. For example, the fact that c is an immediate child of a is declared above as part of the declaration of c, not the declaration of a. The order specifications give complete immediate-parent information, but the set of immediate children of a node is in general open-ended. For example, the declarations above says that c is an immediate child of a, but do not preclude the possibility that there are other, possibly un-named (that is, not declared as constants), immediate children of a. To specify that a Boogie program declares all immediate children of a node n, include the keyword **complete** in the order specification for that node n. For example, a program whose constant declarations are:

$$\textbf{const unique } a\text{: } Wicket <\text{: } \textbf{complete};$$
$$\textbf{const unique } b\text{: } Wicket;$$
$$\textbf{const unique } c\text{: } Wicket <\text{: } a, b\ \textbf{complete};$$
$$\textbf{const unique } d\text{: } Wicket <\text{: } c;$$
$$\textbf{const unique } e\text{: } Wicket;$$

says that

- a has no immediate parents and c is its only immediate child is c (since a is declared with **complete** and c declares a as an immediate parent),
- b has immediate child c but may possibly also have other immediate children (e for example, or some other wicket values for which the program does not contain any **const** declaration),
- a is not an immediate parent of b (since a is declared **complete** and b does not specify a as an immediate parent), and c is not an immediate parent of b (since b and c are distinct by their **unique** declarations, c declares b as an immediate parent, and the partial order has no cycles), but nothing else is said about the immediate parents of b,
- c has exactly two immediate parents, a and b, and exactly one immediate child, d,
- d has immediate parent c and may have any number of immediate children (but not a, b, or c, since the partial order has no cycles).

Finally, the edge to each direct parent may optionally be marked with **unique**, which makes the edge a *unique edge*. Suppose m has a unique edge to parent p (that is, suppose m is a constant declared with **unique** p as a parent) and n has a unique edge

to parent q; then this means: $p = q \ \wedge \ m \neq n$ implies that the respective subgraphs reachable from m and n are disjoint. For example, given the declarations

const unique r: $Wicket$;
const unique s, t: $Wicket$ <: **unique** r;

the following property holds:

$$(\forall\, x\text{: } Wicket, y\text{: } Wicket \bullet\ x <: s \wedge y <: t \Rightarrow x \neq y\ )$$

In words, if x is in the subgraph below s and y is in the subgraph below t, then the facts that s and t are distinct (which follows from "**const unique**") and that each has a unique edge to r (by "<: **unique** r") entail that the subgraphs below s and t are disjoint, and therefore one concludes x and y to be distinct.

10.0 Example: Class and interface hierarchy

Although the partial-order operator can be used for any purpose in Boogie, the order specifications that are part of constant declarations have been tailored to support the ordering among reference types found in object-oriented languages. These order specifications can be complemented by supplying other axioms. Let us look at how the partial order can be used.

To model the class and interface types in a language like Spec#, C#, Java, or Eiffel, we declare a Boogie type for the names of these types:

type $TName$;

Every class declaration in the source language then gives rise to a particular $TName$ value, and we introduce that value with a declaration of a unique constant. For example, the class object at the root of the class hierarchy in Spec# can be declared as follows:

const unique $System.Object$: $TName$;

If we use <: to express the subtyping relation on $TName$, we'll want to introduce some edges in the ordering according to the subclasses declared in the program. For example, if C is declared to be a subclass of B, we may want to define C <: B. The set of subtypes of a class is usually open-ended, allowing arbitrarily many and arbitrarily refined subclasses; however, the set of supertypes of a class is, in common object-oriented languages, known directly from the declaration of the class. For example, the Spec# declaration:

```
class C : B
{
  ...
}
```

may be translated into the following Boogie declaration:

const unique C: $TName$ <: B;

which says that B is the only direct parent of C.

In a single-inheritance language, all subtypes of a user-defined class are disjoint. Stated differently, the user-defined classes form a tree. Thus, one may choose to translate the Spec# class C above into:

$$\textbf{const unique}\ C\colon TName <\colon \textbf{unique}\ B;$$

Note that the interface types in Spec#, C#, and Java do not have this tree-like property, so the declarations:

```
interface J { ... }
class D : B, J { ... }
```

are translated into just:

$$\textbf{const unique}\ J\colon TName <\colon System.Object;$$
$$\textbf{const unique}\ D\colon TName <\colon \textbf{unique}\ B,\ J;$$

Note the absence of "**unique**" in the edges $J <\colon System.Object$ and $D <\colon J$.

In some cases, the complete set of subtypes of a class may be known. Let's consider three cases.

Here is a simple case where the complete set of subtypes of a class is known: if the class is declared to be sealed (in Spec# and C#) or final (in Java), then it has no proper subtypes. For such a class:

```
sealed class S : B { ... }
```

one can use the translation:

$$\textbf{const unique}\ S\colon TName <\colon \textbf{unique}\ B\ \textbf{complete};$$

Since the Boogie translation will not include any other constants below S, the use of **complete** here implies the property:

$$(\forall\ T\colon TName \bullet\ T <\colon S\ \Rightarrow\ T == S\ )$$

Another case where the complete set of subtypes of a class is known is the following. Declarations in Spec# and C# are compiled into so-called assemblies, and a class is available to other assemblies only if it is declared as public. Thus, if the program verifier that translates the source program into Boogie reads an entire assembly, then it can discover all subclasses of a non-public class. The translation of an assembly that contains only the following declarations:

```
internal class W { ... }
internal class X : W { ... }
internal class Y : W { ... }
```

where **internal** explicitly declares the class not to be accessible outside the assembly, can use the following Boogie declarations:

$$\textbf{const unique}\ W\colon TName <\colon \textbf{unique}\ System.Object\ \textbf{complete};$$
$$\textbf{const unique}\ X\colon TName <\colon \textbf{unique}\ W\ \textbf{complete};$$
$$\textbf{const unique}\ Y\colon TName <\colon \textbf{unique}\ W\ \textbf{complete};$$

which, for example, implies the following property about values below W:

$$(\forall\, T\colon TName \bullet\ T <: W\ \Rightarrow\ T == W \vee T <: X \vee T <: Y\ )$$

A third case where the complete set of subtypes of a class is known arises in Spec# and C# if the constructors of the class are not available outside the assembly. For example, an assembly that contains only the declarations:

```
public abstract class Expr {
  internal Expr(...) { ... }
  ...  // but no other constructors
}
public class IdExpr : Expr { ... }
public class BinExpr : Expr { ... }
```

makes all three classes Expr, IdExpr, and BinExpr publicly available, but does not admit Expr subclasses beyond the two given here. This can be encoded in Boogie as follows:

$$\begin{array}{l}\textbf{const unique}\ \mathit{Expr}\colon \mathit{TName} <: \textbf{unique}\ \mathit{System.Object}\ \textbf{complete};\\ \textbf{const unique}\ \mathit{IdExpr}\colon \mathit{TName} <: \textbf{unique}\ \mathit{Expr};\\ \textbf{const unique}\ \mathit{BinExpr}\colon \mathit{TName} <: \textbf{unique}\ \mathit{Expr};\end{array}$$

Note that $IdExpr$ and $BinExpr$ do not use the **complete** modifier; but the fact that the parent $Expr$ does implies that a $TName$ is strictly below $Expr$ if and only if it is below $IdExpr$ or $BinExpr$.

As a final aspect of this example, let's consider a use of $TName$ in the translation. The dynamic type (that is, the allocated type) of an object can be modeled as a function from object references to $TName$:

$$\textbf{function}\ \mathit{dtype}(\mathit{Ref})\ \textbf{returns}\ (\mathit{TName});$$

This function and the ordering on $TName$ can be used to encode static type information. For example, consider a method:

```
class MyClass {
  public void M(InputStream s) { ... }
  ...
}
```

It may be translated into a Boogie procedure along the following lines:

$$\begin{array}{l}\textbf{procedure}\ \mathit{MyClass.M}(\mathit{this}\colon \mathit{Ref},\ s\colon \mathit{Ref});\\ \quad \textbf{free requires}\ \mathit{this} \neq \mathit{null} \wedge \mathit{dtype}(\mathit{this}) <: \mathit{MyClass};\\ \quad \textbf{free requires}\ s == \mathit{null} \vee \mathit{dtype}(s) <: \mathit{InputStream};\\ \quad \ldots\end{array}$$

Note that all reference types in the source language are translated into Boogie's type Ref and that the distinction between different source-level object types is made through the $dtype$ function and the $<:$ ordering on $TName$. The condition $\mathit{dtype}(s) <: \mathit{InputStream}$ states the well-known programming-language property "the dynamic type of the reference stored in a variable is a subtype of the static type of that variable", which is guaranteed by the static type system of the source language (which makes it appropriate for these preconditions to be declared with **free**).

If MyClass and InputStream are direct subclasses of object, then the corresponding constant declarations in the Boogie translation might be:

$$\begin{array}{l}\textbf{const unique } System.Object\colon TName;\\ \textbf{const unique } MyClass\colon TName <\colon \textbf{unique } System.Object;\\ \textbf{const unique } InputStream\colon TName <\colon \textbf{unique } System.Object;\end{array}$$

The **unique** modifiers in these declarations imply $MyClass \neq InputStream$. However, note that that does not by itself imply $this \neq s$ for $MyClass.M$, because the uniqueness of the type names does not rule out the possibility that:

$$\begin{array}{l}s \neq null \ \wedge\ dtype(this) = dtype(s) \ \wedge\\ dtype(this) <\colon MyClass \ \wedge\\ dtype(this) <\colon InputStream\end{array}$$

But the use of unique parent edges to $System.Object$ in the declarations of $MyClass$ and $InputStream$ rules out this possibility, and thus $this \neq s$ follows from the preconditions of procedure $MyClass.M$.

Interfaces and abstract classes have no instances, yet the names of such interfaces and classes are included as values of $TName$. The fact that such types have no instances is modeled as a property of $dtype$. For example, for interface J and abstract class Expr above, one may include the following axioms:

$$\begin{array}{l}\textbf{axiom } (\forall\, r\colon Ref \bullet\ dtype(r) \neq Expr\ );\\ \textbf{axiom } (\forall\, r\colon Ref \bullet\ dtype(r) \neq J\ );\end{array}$$

11 Tool directives

Every top-level declaration, local-variable declaration, assert and assume statement, procedure specification clause, loop invariant, and quantifier can be annotated with a number of *attributes*. Attributes record meta data that can be used by tools that process Boogie programs. The meta data can act as directives to such tools, but the Boogie language does not assign any formal meaning to the attributes.

In addition, quantifiers can be annotated with *triggers*, which have a form similar to attributes. A trigger is a directive that tells a theorem prover how to instantiate quantifiers. Triggers can be crucial to get good performance from provers.

The grammars for attributes and triggers are:

$$\begin{array}{rcl}Attribute & ::= & \{\ :\ Id\ AttrArg{,}^{*}\ \}\\ AttrArg & ::= & Expr \mid StringLiteral\\ Trigger & ::= & \{\ Expr{,}^{+}\ \}\end{array}$$

where $StringLiteral$ denotes a terminal that begins with a double-quote character and ends at the subsequent double-quote character (with no intervening line-breaking whitespace). Note that string literals are not used anywhere else in Boogie.

The arguments to attributes and triggers must themselves type check:

$$\frac{(e \text{ is string literal}) \text{ or } (\mathcal{T}, \mathcal{V} \Vdash e : T) \text{ for all } (e, T) \in (\overline{e}, \overline{T})}{\mathcal{T}, \mathcal{V} \Vdash_{attr} \{: id\ \overline{e}\}}$$

$$\frac{\mathcal{T}, \mathcal{V} \Vdash\ e : T \text{ for all } (e, T) \in (\overline{e}, \overline{T})}{\mathcal{T}, \mathcal{V} \Vdash_{attr} \{\overline{e}\}}$$

A tool that uses a directive can therefore rely on the given expression arguments to be well-formed Boogie expressions. There are additional restrictions on triggers, as described below.

11.0 Examples: Using attributes

Consider a tool that reads a Boogie program, infers some loop invariants, and writes out the Boogie program where the inferred invariants are recorded as assumptions in loop heads. (If the invariants are recorded as **assume** statements, the inference tool takes responsibility for the correctness of the invariants. If the inference engine is not sound or is itself being debugged, then one can instead produce **assert** statements of the inferred purported invariants, which means they can be checked by other tools, like a theorem prover.)

It may be that one wants the inference engine to infer invariants only for some of the program variables. For example, perhaps one does not want any inference for some variables that are introduced in the translation from the source language to Boogie. Such a variable x can be marked an attribute, like:

$$\mathbf{var}\ \{ :\mathit{noInference} \}\ x\colon \mathbf{int};$$

or perhaps:

$$\mathbf{var}\ \{ :\mathit{inference}\ \mathbf{false} \}\ x\colon \mathbf{int};$$

The inference tool would then pay attention to the $\mathit{noInference}$ or $\mathit{inference}$ attribute and act accordingly. Since Boogie only provides type checking of each given attribute argument, any tool that looks for a particular attribute needs must first inspect the number of types of arguments provided.

There is only one name space for all attributes, so a tool may want to design its attributes in such a way they not easily accidentally confused with attributes supported by other tools. For example, an inference engine called Clousot may choose to support an attribute like:

$$\mathbf{var}\ \{ :\mathit{ClousotInference}\ \mathbf{false} \}\ x\colon \mathbf{int};$$

or maybe even:

$$\mathbf{var}\ \{ :\mathit{inference}\ \text{"}\mathit{Clousot}\text{"}, \mathbf{false} \}\ x\colon \mathbf{int};$$

As another example, a verifier for Boogie programs may have a mode where it does not verify each procedure implementation separately, but where it instead performs a symbolic execution of the program from its entry points. Attributes can be used to specify the entry points, for example:

$$\mathbf{procedure}\ \{ :\mathit{public} \}\ \mathit{MyClass..ctor}(\mathit{this}\colon \mathit{Ref}) \ldots$$

As a final small example, a tool that attempts to verify the correctness of a program may output the assertions, preconditions, postconditions, and loop invariants that could not be proved. While returning the identity of the internal AST nodes for these various assertions would produce the most accurate results for a frontend that wants to map back these proof failures into error messages pertinent to the source language, some other frontends may rely on using a Boogie tool via a textual interface. In these cases, the tool might output the line and column information for the failing assertions. Alternatively, a tool can make use of attributes, specifying what message to output in the event that an assertion cannot be proved. For example, a source line:

```
218     x = p.f;
```

can be translated into the following snippet of Boogie code:

$$
\begin{array}{l}
\textbf{assert} \\
\quad \{\, :errorMessage\ \text{"possible null dereference on line 218, column 10"}\,\} \\
\quad p \neq null; \\
x := Heap[p, C.f];
\end{array}
$$

11.1 Example: Bit-vector operations

The Boogie language includes types for bit vectors, but the only built-in support for bit-vector operations are literals, bit extraction, concatenation, and equality. Further operations, like modulo arithmetic, can be defined by functions and axioms. However, some theorem provers may have direct support for bit-vector operations. When such a theorem prover is to be used, one would like to direct the prover in two ways. First, the prover needs to be directed to connect the functions used in the Boogie program with the syntax or function names used by the theorem prover. Second, the Boogie program may include axioms about these functions, so that theorem provers without built-in support still can give some interpretation to the functions. A prover with built-in bit-vector support needs to be directed to ignore such axioms.

Here is an example of how one can express these two directives:

$$
\begin{array}{l}
\textbf{function}\ \{\, :bvBuiltin\ \text{"}bvadd8\text{"}\,\}\ ByteAdd(\mathbf{bv}8, \mathbf{bv}8)\ \textbf{returns}\ (\mathbf{bv}8); \\
\textbf{axiom}\ \{\, :bvIgnore\,\}\ (\forall\, a\colon \mathbf{bv}8,\ b\colon \mathbf{bv}8 \bullet\ \ \ldots ByteAdd(a, b) \ldots\ );
\end{array}
$$

11.2 Triggers

Some theorem provers in the style of the SMT solver Simplify use triggers (aka *matching patterns*) that determine how to instantiate universal quantifiers [3]. Default triggers are inferred by the SMT solver from the body of the quantifier; user-specified triggers override the defaults. Use of appropriate triggers is crucial to getting good performance and desirable results from the SMT solver. Therefore, Boogie provides a convenient built-in syntax for them.

A quantifier $(\forall\, x\colon T \bullet\ \ \ldots x \ldots\ )$ says that its body holds for every value x of type T. Thus, it is mathematically sound to instantiate it with any T value. However, most T values are likely not to get closer to the proof goal. A trigger limits the possible

instantiations by describing terms that must already be present in the proof context at the time of instantiation and describing how the instantiations are picked from those terms. For example,

$$(\,\forall\, x\colon T \bullet \ \{f(x)\}\ g(f(x)) < 100\ )$$

directs the prover to choose the instantiations of this quantifier to be those x's that occur as terms $f(x)$ in the current proof context.

There are some restrictions on the use of triggers. The list of terms in a trigger must mention all bound variables of the quantifier (this restriction ensures that some discrimination is applied to all bound variables). Furthermore, a term listed in a trigger must not be a bound variable by itself (this restriction ensures that the term contributes to the discrimination). Finally, a trigger must not include logical operators or quantifiers.

For more information about triggers, see the Simplify paper [3]. To learn more about using triggers effectively in the axiomatization of a problem, see the formalization of summation-like comprehensions in Spec# [5].

11.3 Example: Exists unique

[To-do: can use exists unique *as an example in this section]*

[To-do: As further examples, discuss some specific attributes that are supported by the Boogie tool.]

[To-do: Talk about the intention of **where** *clauses and free specifications and use these to motivate why the inline directive ignores them.]*

12 Examples

TBW

13 Related work

TBW

14 Conclusions

TBW

Acknowledgments

The type system of Boogie 2 has taken considerable time to figure out, primarily because of the desire to model the heap as something of type $\langle\alpha\rangle[Ref\,, Field\ \alpha]\alpha$ and the need to write frame conditions that quantify over fields and say where the heap might be changed (which affects equality). I have learned a lot and benefited greatly from discussions with Nikolaj Bjørner, Jean-Christophe Filliâtre, Ian Hayes, Viktor Kuncak,

William Lovas, Claude Marché, Martin Odersky, Michel Sintzoff, Cesare Tinelli, and Burkhart Wolff, and the continuous guidance of Daan Leijen. Philipp Rümmer and I are defining a semantic basis for the types and expressions of the language.

The **call forall** statement was designed with Itay Neeman, who also discovered the need for it when attempting to write a convincing "proof" using Boogie.

The **returns** syntax is influenced by CLU [7]. The wildcard syntax in **if** and **while** statements is borrowed from the input language of Bebop [0].

Radu Grigore and Jochen Hoenicke provided detailed comments on drafts of this manual. Jochen Hoenicke, Michał Moskal, and Philipp Rümmer helped sort out some tricky issues.

Tony Hoare and Butler Lampson have both suggested that I change the name of the language BoogiePL (which, by the way, had been named that in order to distinguish it from the defunct object-oriented language BoogieOOL that translated into BoogiePL), and Wolfram Schulte has often referred to the language simply as Boogie. Thank you, and I have now renamed this major revision of the language to just Boogie.

I'm indebted to the makers of the fabulous, classic movie *This is Spinal Tap* [9], which, as will be immediately evident to other fans of the movie, inspired the title and prelude of this document.

A Grammar

The grammar of the language has been presented throughout the document. For convenience, here it is repeated in one place.

The reserved keywords are:

assert, **assume**, **axiom**, **bool**, **break**, **bv**0, **bv**1, **bv**2, ..., **call**, **complete**, **const**, **else**, **ensures**, **exists**, **false**, **finite**, **forall**, **free**, **function**, **goto**, **havoc**, **if**, **implementation**, **int**, **invariant**, **modifies**, **old**, **procedure**, **requires**, **return**, **returns**, **true**, **type**, **unique**, **var**, **where**, **while**.

The grammar is:

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Decl}^* \\
\mathit{Decl} & ::= & \mathit{TypeDecl} \mid \mathit{ConstantDecl} \mid \mathit{FunctionDecl} \mid \mathit{AxiomDecl} \\
& \mid & \mathit{VarDecl} \mid \mathit{ProcedureDecl} \mid \mathit{ImplementationDecl}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{TypeDecl} & ::= & \mathit{TypeConstructor} \mid \mathit{TypeSynonym} \\
\mathit{TypeConstructor} & ::= & \mathbf{type}\ \mathit{Attribute}^*\ \mathbf{finite}^?\ \mathit{Id}\ \mathit{Id}^*\ ; \\
\mathit{TypeSynonym} & ::= & \mathbf{type}\ \mathit{Attribute}^*\ \mathit{Id}\ \mathit{Id}^* = \mathit{Type}\ ;
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Type} & ::= & \mathit{TypeAtom} \mid \mathit{MapType} \\
& \mid & \mathit{Id}\ \mathit{TypeCtorArgs}^{?} \\
\mathit{TypeAtom} & ::= & \mathbf{bool} \mid \mathbf{int} \\
& \mid & \mathbf{bv}0 \mid \mathbf{bv}1 \mid \mathbf{bv}2 \mid \cdots \\
& \mid & (\ \mathit{Type}\ ) \\
\mathit{MapType} & ::= & \mathit{TypeArgs}^{?}\ [\ \mathit{Type},^{+}\ ]\ \mathit{Type} \\
\mathit{TypeArgs} & ::= & \langle\ \mathit{Id},^{+}\ \rangle \\
& \mid & <\ \mathit{Id},^{+}\ > \\
\mathit{TypeCtorArgs} & ::= & \mathit{TypeAtom}\ \mathit{TypeCtorArgs}^{?} \\
& \mid & \mathit{Id}\ \mathit{TypeCtorArgs}^{?} \\
& \mid & \mathit{MapType}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{ConstantDecl} & ::= & \mathbf{const}\ \mathit{Attribute}^{*}\ \mathbf{unique}^{?}\ \mathit{IdsType}\ \mathit{OrderSpec}\ ; \\
\mathit{IdsType} & ::= & \mathit{Id},^{+} : \mathit{Type}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{FunctionDecl} & ::= & \mathbf{function}\ \mathit{Attribute}^{*}\ \mathit{Id}\ \mathit{FSig}\ ; \\
& \mid & \mathbf{function}\ \mathit{Attribute}^{*}\ \mathit{Id}\ \mathit{FSig}\ \{\ \mathit{Expr}\ \} \\
\mathit{FSig} & ::= & \mathit{TypeArgs}^{?}\ (\ \mathit{FArg},^{*}\ )\ \mathbf{returns}\ (\ \mathit{FArg}\ ) \\
\mathit{FArg} & ::= & \mathit{FArgName}^{?}\ \mathit{Type} \\
\mathit{FArgName} & ::= & \mathit{Id}\ :
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & E0 \\
E0 & ::= & E1 \mid E1\ \mathit{EquivOp}\ E0 \\
E1 & ::= & E2 \mid E2\ \mathit{ImplOp}\ E1 \\
E2 & ::= & E3 \mid E3\ \mathit{EOr}^{+} \mid E3\ \mathit{EAnd}^{+} \\
\mathit{EOr} & ::= & \mathit{OrOp}\ E3 \\
\mathit{EAnd} & ::= & \mathit{AndOp}\ E3 \\
E3 & ::= & E4 \mid E4\ \mathit{RelOp}\ E4 \\
E4 & ::= & E5 \mid E4\ \mathit{ConcatOp}\ E5 \\
E5 & ::= & E6 \mid E5\ \mathit{AddOp}\ E6 \\
E6 & ::= & E7 \mid E6\ \mathit{MulOp}\ E7 \\
E7 & ::= & \mathit{UnOp}^{*}\ E8 \\
E8 & ::= & E9\ \mathit{MapOp}^{*} \\
\mathit{MapOp} & ::= & [\ \mathit{Expr},^{+}\ \mathit{MapUpdate}^{?}\ ] \\
 & \mid & [\ \mathit{Number}\ :\ \mathit{Number}\ ] \\
\mathit{MapUpdate} & ::= & :=\ \mathit{Expr} \\
E9 & ::= & \mathbf{false} \mid \mathbf{true} \mid \mathit{Number} \mid \mathit{BitVector} \\
 & \mid & \mathit{Id}\ \mathit{FuncApplication}^{?} \\
 & \mid & \mathbf{old}\ (\ \mathit{Expr}\ ) \\
 & \mid & (\ \mathit{QOp}\ \mathit{TypeArgs}^{?}\ \mathit{IdsType},^{+}\ \mathit{QSep}\ \mathit{TrigAttr}^{*}\ \mathit{Expr}\ ) \\
 & \mid & (\ \mathit{Expr}\ ) \\
\mathit{FuncApplication} & ::= & (\ \mathit{Expr},^{*}\ ) \\
\mathit{TrigAttr} & ::= & \mathit{Trigger} \mid \mathit{Attribute} \\
\mathit{Number} & ::= & 0 \mid 1 \mid 2 \mid \cdots \\
\mathit{BitVector} & ::= & 0\mathbf{bv}0 \\
 & \mid & 0\mathbf{bv}1 \mid 1\mathbf{bv}1 \\
 & \mid & 0\mathbf{bv}2 \mid 1\mathbf{bv}2 \mid 2\mathbf{bv}2 \mid 3\mathbf{bv}2 \\
 & \mid & 0\mathbf{bv}3 \mid 1\mathbf{bv}3 \mid 2\mathbf{bv}3 \mid 3\mathbf{bv}3 \mid 4\mathbf{bv}3 \mid 5\mathbf{bv}3 \mid 6\mathbf{bv}3 \mid 7\mathbf{bv}3 \\
 & \mid & \cdots
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{EquivOp} & ::= & \iff & \mid\ \texttt{<==>} \\
\mathit{ImplOp} & ::= & \Rightarrow & \mid\ \texttt{==>} \\
\mathit{OrOp} & ::= & \vee & \mid\ \texttt{||} \\
\mathit{AndOp} & ::= & \wedge & \mid\ \texttt{\&\&} \\
\mathit{RelOp} & ::= & == & \\
 & \mid & \neq & \mid\ \texttt{!=} \\
 & \mid & < \mid > & \\
 & \mid & \leqslant \mid \geqslant & \mid\ \texttt{<=} \mid \texttt{>=} \\
 & \mid & <: & \\
\mathit{ConcatOp} & ::= & ++ & \\
\mathit{AddOp} & ::= & + \mid - & \\
\mathit{MulOp} & ::= & * \mid / \mid \% & \\
\mathit{UnOp} & ::= & \neg & \mid\ \texttt{!} \\
 & \mid & - & \\
\mathit{QOp} & ::= & \forall \mid \exists & \mid\ \mathbf{forall} \mid \mathbf{exists} \\
\mathit{QSep} & ::= & \bullet & \mid\ :: \\
\mathit{AxiomDecl} & ::= & \mathbf{axiom}\ \mathit{Attribute}^{*}\ \mathit{Expr}\ ; &
\end{array}
$$

VarDecl ::= **var** *Attribute** *IdsTypeWhere*,$^+$;
IdsTypeWhere ::= *IdsType WhereClause*$^?$

ProcedureDecl ::= **procedure** *Attribute** *Id PSig* ; *Spec**
| **procedure** *Attribute** *Id PSig Spec** *Body*
PSig ::= *TypeArgs*$^?$ (*IdsTypeWhere*,*) *OutParameters*$^?$
OutParameters ::= **returns** (*IdsTypeWhere*,*)

Spec ::= **free**$^?$ **requires** *Expr* ;
| **free**$^?$ **modifies** *Id*,* ;
| **free**$^?$ **ensures** *Expr* ;

ImplementationDecl ::= **implementation** *Attribute** *Id ISig Body**
ISig ::= *TypeArgs*$^?$ (*IdsType*,*) *OutParameters*$^?$
OutParameters ::= **returns** (*IdsType*,*)

Body ::= { *LocalVarDecl** *StmtList* }
LocalVarDecl ::= **var** *Attribute** *IdsTypeWhere*,$^+$;
StmtList ::= *LStmt** *LEmpty*$^?$
LStmt ::= *Stmt* | *Id* : *LStmt*
LEmpty ::= *Id* : *LEmpty*$^?$
Stmt ::= **assert** *Expr* ;
| **assume** *Expr* ;
| **havoc** *Id*,$^+$;
| *Lhs*,$^+$:= *Expr*,$^+$;
| **call** *CallLhs*$^?$ *Id* (*Expr*,*) ;
| **call forall** *Id* (*WildcardExpr*,*) ;
| *IfStmt*
| **while** (*WildcardExpr*) *LoopInv** *BlockStmt*
| **break** *Id*$^?$;
| **return** ;
| **goto** *Id*,$^+$;
Lhs ::= *Id MapSelect**
MapSelect ::= [*Expr*,$^+$]
CallLhs ::= *Id*,$^+$:=
WildcardExpr ::= *Expr* | $*$
BlockStmt ::= { *StmtList* }
IfStmt ::= **if** (*WildcardExpr*) *BlockStmt Else*$^?$
Else ::= **else** *BlockStmt* | **else** *IfStmt*
LoopInv ::= **free**$^?$ **invariant** *Expr* ;

OrderSpec ::= *ParentInfo*$^?$ **complete**$^?$
ParentInfo ::= <: *ParentEdge*,*
ParentEdge ::= **unique**$^?$ *Id*

Attribute ::= { : *Id AttrArg*,* }
AttrArg ::= *Expr* | *StringLiteral*
Trigger ::= { *Expr*,$^+$ }

References

0. Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In Klaus Havelund, John Penix, and Willem Visser, editors, *SPIN Model Checking and Software Verification, 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, August–September 2000.
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. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
2. Shaunak Chatterjee, Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS 2007*, Lecture Notes in Computer Science. Springer, March 2007.
3. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
4. Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
5. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *Workshop on Formal Techniques for Java-like Programs (FTfJP 2007)*, July 2007.
6. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. In Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*, Technical Report 251. Fernuniversität Hagen, May 1999. Also available as Technical Note 1999-002, Compaq Systems Research Center.
7. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
8. D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
9. Rob Reiner. *This is Spinal Tap®: A Rockumentary by Martin Di Bergi*. Christopher Guest, Michael McKean, Harry Shearer, and Rob Reiner, writers. Spinal Tap Prod, March 1984.
10. Wolfram Schulte, Songtao Xia, Jan Smans, and Frank Piessens. A glimpse of a verifying C compiler (extended abstract). In *C/C++ Verification Workshop*, 2007.