BoGL Syntax*

Martin Erwig

March 10, 2021

1 Lexical Syntax

The nonterminal *name* stands for alphanumeric strings that begins with a lowercase letter and may contain underscores; it is used for function and variable names, which include parameters. The nonterminal *Name* stands for alphanumeric strings that begin with an uppercase letter and may contain underscores; it is used for type names, game names, and values other than numbers. To better distinguish values from type names in the grammar, we also use the nonterminal *Type* for type names and the nonterminal *Value* for values. The nonterminal *int* stands for integer constants.

The type names `Int`, `Bool`, `Board`, and `Input` are reserved. While `Int` and `Bool` are predefined types, `Board` and `Input` are type names that can be defined in a program, and when they are, they have a special meaning representing game boards and input, respectively. The values `True` and `False` are predefined values of type `Bool`. Any other name that starts with an uppercase letter (`A`, `Player`, `Nothing`, etc.) can be a value or type name, depending on how it is used in a program. Predefined functions are discussed in Section 6.

Comments are handled as in Haskell, that is, everything after `--` in a line is treated as a comment, and multi-line comments are enclosed by `{-` and `-}`.

2 Game and Type Definitions

A BoGL program has a name and consists of a number of type definitions followed by any number of value and function definitions. Note that in the syntax for type definitions the nonterminal *Type* represents a name, whereas the nonterminal *type* represents a type expression to be (defined in Section 3). Type definitions may include a definition for a board and an input type. The input type definition has the following form.

`type Input =` *type*

This definition ensures that all input read during the execution of a program has the type *type*. The board type definition determines the board dimensions and the type of values that can appear on a board. A board type definition must use the type name `Board` and thus has the following general form.

`type Board = Array (`*int*`,`*int*`) of` *type*

A board type definition defines implicitly the names `width` and `height` that are bound to the width and height of the board. The board type definition also implicitly assigns the type name `Content` to *type*, that is, it performs the following type definition where *type* is the same type as used in the board type definition.

*An initial version of BoGL was created by a group of OSU students in the context of two senior design projects during the 2019/2020 academic year. The latest version of the syntax is defined in this document.

```
type Content = type
```

Built-in functions for boards are briefly discussed in Section 6.

Syntax of Games and Type Definitions

game	::=	`game` *Name typedef* valuedef**	(*game definition*)
typedef	::=	`type` *Type* `=` *type*	(*general type definition*)
	|	`type Board = Array (`*int*`,`*int*`) of` *type*	(*board type definition*)

3 Types

The type system offers two predefined types: `Int` for integers and `Bool` for the boolean values `True` and `False`. A type definition (see Section 2) assigns a type name to a type expression. The newly introduced type name is also called a *type synonym* (for the type expression). If defined, the type name `Input` refers to the type of information that is gathered from the user during an execution of the game, and the type name `Board` refers to the type that represents game boards. A tuple type stands for the set of tuples (of 2 or more components) that can be formed with the values of the argument types. For example, `(3,True)` is a value of the tuple type `(Int,Bool)`. Predefined types, type synonyms, and tuple types are called *base types*.

An enumeration type is given by a set of values. For example, the enumeration type `{Left, Right}` contains the two values `Left` and `Right`. Enumeration types can occur only as part of type definitions, which assign a type name to each of the values. The following type definition introduces the type (name) `Move` and assigns this type to the values `Left` and `Right`.

```
type Move = {Left, Right}
```

Any type can be extended by (a type synonym for) an enumeration type. Such a type is then called an *extended type*. Here are two examples.

```
type Result = Int & {Undefined}
type Direction = Move & {Up, Down}
```

Note that each value can be used at most once in an enumeration type.

Base types, enumeration types, and extended types can all be used as type expressions in a type definition. A *function type* consists of an argument and a result type, which both have to be base types, and represents functions that map values of the argument type to values of the result type. A *value type* is either a base type or a function type. Each value definition (see Section 4) requires an explicit annotation with a value type.

Note that we have excluded the array type definition for boards from the syntax of types, since it is not generally available and can be only used to define the the one type `Board`.

Syntax of Types

btype	::=	*Type* | `(`*btype*`, ... ,` *btype*`)`	(*base type*)
etype	::=	`{`*Value*`, ... ,` *Value*`}`	(*enumeration type*)
type	::=	*btype* | *etype* | *type* `&` *Type* | *type* `&` *etype*	(*type expression*)
ftype	::=	*btype* `->` *btype*	(*function type*)
vtype	::=	*btype* | *ftype*	(*value type*)

4 Value Definitions

BoGL has three kinds of values: *basic values*, *boards*, and *functions*, and any such value can be bound to a name through a *value definition*. A value definition requires a *signature* that declares the type of the value and one or more equations, which consists of a name and an expression (see Section 5) that denotes the value to be bound to the name. A basic value definition just needs the name and the expression. In the case of a function definition, the name is followed by a tuple of names, which represent the function's parameters. A board is often defined by multiple board equations. In each such equation, the name is followed by a `!` and a pair of positions (which can be integers or variable names), indicating one or more positions on the board.[1] For example, the meaning of the board equation `board!(x,2) = Empty` is to set all fields in the second row to `Empty`. Multiple board equations are evaluated in a top down order to construct the board. All positions on a board must be defined.

Syntax of Value Definitions

valuedef	::=	*signature equation*	(*value definition*)
signature	::=	*name* `:` *vtype*	(*type signature*)
equation	::=	*name* `=` *expr*	(*value equation*)
	|	*name*`(`*name*`, ... ,` *name*`)` `=` *expr*	(*function equation*)
	|	*boardeq* ... *boardeq*	(*board equations*)
boardeq	::=	*name*`!(`*pos*`,`*pos*`)` `=` *expr*	(*board range definition*)
pos	::=	*int* | *name*	(*board position*)

5 Expressions

Atomic expressions consist of all the basic values, which include integers, the boolean values `True` and `False`, plus all values introduced through enumeration types, as well as all names defined through value definitions, with the exception of function names.

Based on atomic expressions, more complex expressions can be built by forming tuples and by applying functions or infix operations. In addition to the usual arithmetic operations and comparison operators, infix application also includes the notation for board lookup `board!p` where `p` must be an expression that evaluates to a pair of integers. The purpose of tuple projection is to extract a single value or a sub-tuple of values from a tuple. They are defined separately from the infix operations since they require specific forms as arguments.

Local definitions can be performed using the `let` construct: an expression `let x=e in e'` produces as a result the result of `e'` where all occurrences of `x` in `e'` are substituted by `e`.

[1] This notation mirrors the notation of board lookup, see Section 5.

Syntax of Expressions

aexpr	::=	*int* | *Value* | *name*	(*value, name*)
expr	::=	*aexpr*	(*atomic expression*)
	|	`(`*expr*`, ... ,`*expr*`)`	(*tuple*)
	|	*name*`(`*expr*`, ... ,`*expr*`)`	(*function application*)
	|	*expr binop expr*	(*infix application*)
	|	*expr*`#`*int*	(*single tuple projection*)
	|	*expr*`#(`*int*`, ... ,`*int*`)`	(*tuple projection*)
	|	`let` *name* `=` *expr* `in` *expr*	(*local definition*)
	|	`if` *expr* `then` *expr* `else` *expr*	(*conditional*)
	|	`while` *expr* `do` *expr*	(*while loop*)
binop	::=	`+` | `-` | `*` | `/` | `!` | `==` | `/=` | `>` | `>=` | `<` | `<=`	(*binary operation*)

Finally, expressions include two control structures: conditional expressions and while loops. A while loop `while c do e` is always evaluated in the context of a binding, which we call the *binding context* of the while loop. This can be either a single variable or a tuple of variables, introduced by a `let` expression or as the parameter(s) of a function definition. Let's assume the binding context is `(x1,...xn)`. Both the condition `c` and the body `e` of the while loop can refer to the variables from the binding context, and the expression `e` must either produce a single value or a tuple of values `(v1,...vn)` whose size and type matches that of the binding context. If `c` evaluates to `True`, the variables of the binding context are updated by the values `(v1,...vn)` that results from evaluating the body `e`. If `c` evaluates to `False`, the tuple `(x1,...xn)` is evaluated and the tuple of values of the variables is returned as a result.

6 Built-In Values and Functions

Here is a list of all built-in BoGL values and functions. The name `input` refers to a value of type `Input` (if that type is defined). That value is determined by the user's input provided in response to each call `input` and thus changes over time.

```
input : Input
```

If a program defines a board type, the names `width` and `height` refer to its width and height, respectively. The function call `place(c,b,(x,y))` changes the board `b` by placing the value `c` at the position `(x,y)`. The function call `count(c,b)` returns the number of times the value `c` occurs on the board `b`, and the function call `longestRow(c,b)` returns the length of the longest horizontal, vertical, or diagonal row of `c` values on the board.

```
width      : Int
height     : Int
place      : (Content,Board,(Int,Int)) -> Board
count      : (Content,Board) -> Int
longestRow : (Content,Board) -> Int
```

Finally, the functions `not`, `and`, and `or` perform the logical operations of the same name that operate on the `Bool` values `True` and `False`.

```
True  : Bool
False : Bool
not   : Bool -> Bool
or    : (Bool,Bool) -> Bool
and   : (Bool,Bool) -> Bool
```

Appendix: BoGL Syntax Chart

Complete BoGL Syntax

game	::=	`game` *Name typedef* valuedef**	(*game definition*)
typedef	::=	`type` *Type* `=` *type*	(*general type definition*)
	|	`type Board = Array (`*int*`,`*int*`) of` *type*	(*board type definition*)
btype	::=	*Type* | `(`*btype*`, ... ,` *btype*`)`	(*base type*)
etype	::=	`{`*Value*`, ... ,` *Value*`}`	(*enumeration type*)
type	::=	*btype* | *etype* | *type* `&` *Type* | *type* `&` *etype*	(*type expression*)
ftype	::=	*btype* `->` *btype*	(*function type*)
vtype	::=	*btype* | *ftype*	(*value type*)
valuedef	::=	*signature equation*	(*value definition*)
signature	::=	*name* `:` *vtype*	(*type signature*)
equation	::=	*name* `=` *expr*	(*value equation*)
	|	*name*`(`*name*`, ... ,` *name*`)` `=` *expr*	(*function equation*)
	|	*boardeq ... boardeq*	(*board equations*)
boardeq	::=	*name*`!(`*pos*`,`*pos*`)` `=` *expr*	(*board range definition*)
pos	::=	*int* | *name*	(*board position*)
aexpr	::=	*int* | *Value* | *name*	(*value, name*)
expr	::=	*aexpr*	(*atomic expression*)
	|	`(`*expr*`, ... ,`*expr*`)`	(*tuple*)
	|	*name*`(`*expr*`, ... ,`*expr*`)`	(*function application*)
	|	*expr binop expr*	(*infix application*)
	|	*expr* `#` *int*	(*single tuple projection*)
	|	*expr*`#(`*int*`, ... ,`*int*`)`	(*tuple projection*)
	|	`let` *name* `=` *expr* `in` *expr*	(*local definition*)
	|	`if` *expr* `then` *expr* `else` *expr*	(*conditional*)
	|	`while` *expr* `do` *expr*	(*while loop*)
binop	::=	`+` | `-` | `*` | `/` | `!` | `==` | `/=` | `>` | `>=` | `<` | `<=`	(*binary operation*)