BoGL Syntax

Martin Erwig

February 20, 2020

We generally adopt the Haskell rules for lexical syntax, in particular, *name* is used for function and parameter names and stands for alphanumeric strings starting with a lowercase letter. Similarly, *Name* is used for symbols (which are basically nullary constructors) and stands for alphanumeric strings starting with an uppercase letter. It is also used for the names of games. The nonterminals *int* and *string* stand for integer and string constants, respectively. Comments are also handled as in Haskell.

A game definition consists of a name, two mandatory type definitions for the board and player inputs, and any number of value and function definitions. The board definition determines the board dimensions and the type of values contained on a board.

Games and Type Definitions

game	::=	`game` *Name board input* $\mathit{typedef}^*$ $\mathit{valuedef}^*$	(*game definition*)
board	::=	`type Board = Array (`*int*`,`*int*`) of` *type*	(*board type*)
input	::=	`type Input =` *type*	(*input type*)
typedef	::=	`type` *Name* `=` *type*	(*type definition*)

Value definitions (for values and functions) require a signature declaring their type plus an equation for defining the value. If a value is parameterized, that is, if it is followed by a parameter list, it represents a function.

A board definition (for a variable *name* with declared type `Board`) is equivalent to an array definition and may contain a number of definitions for individual positions (given by pairs of integers) and for sets of positions, which can be given by using a variable for either or both coordinates. The semantics of a definition such as `board(x,2) = Empty` is to set all fields in the second row to `Empty`.

Value Definitions

valuedef	::=	*signature equation*	(*value definition*)
signature	::=	*name* `:` *type*	(*type signature*)
equation	::=	*name* = *expr*	(*value equation*)
	|	*name parlist* = *expr*	(*function equation*)
	|	$\mathit{boarddef}^*$	(*board definition*)
boarddef	::=	*name* `!` `(`*pos*`,`*pos*`)` = *expr*	(*board equation*)
pos	::=	*int* | *name*	(*board positions*)
parlist	::=	$(\mathit{par}_1, \ldots, \mathit{par}_k)$	(*parameter list,* $k \geq 1$)
par	::=	*name* | *parlist*	(*parameter*)

The types system builds on a collection of basic types, which include predefined numbers and booleans as well as the type `Symbol`, which is the type for all symbol values (*Name*). The predefined symbol values `A` and `B` also have the more specific type `Player`. The special type `Input` is a synonym for the type of information that is gathered from the user in each move during an execution of the game. Basic types also include types specific to the domain of board games. Among those, the type `Position` is a synonym for `(Int,Int)`. The type `Positions` is an abstract data type for collections of positions, which can be though of as sets or lists of positions. However, the underlying representation is not exposed to the programmer. A value of type `Positions` can only be generated and manipulated by a number of predefined operations.

Extended types extend a base type by one or more symbols, which are values and can be thought of nullary data constructors. For example, the Haskell type `Maybe Int` can be represented by the extended type `Int|Nothing`. Extended types facilitate the extension of types by values for representing "special" situations. For example, `Player|Empty` is a type typically used in a board type definition.

A tuple type can contain extended types, and a plain type is either a tuple type or an extended type. The language has first-order function types whose argument and result types can be any plain type.

Types

$$\begin{array}{rcll}
\textit{btype} & ::= & \texttt{Bool | Int | Symbol | Input} & (\textit{atomic type}) \\
 & | & \texttt{Board | Player | Position | Positions} & (\textit{game type}) \\
\textit{xtype} & ::= & \textit{btype}(\texttt{|}\textit{Name})^* & (\textit{extended type}) \\
\textit{ttype} & ::= & \texttt{(}\textit{xtype}_1\texttt{, ... ,}\textit{xtype}_k\texttt{)} & (\textit{tuple type, } k \geq 2) \\
\textit{ptype} & ::= & \textit{xtype} \texttt{ | } \textit{ttype} & (\textit{plain type}) \\
\textit{ftype} & ::= & \textit{ptype} \texttt{ -> } \textit{ptype} & (\textit{function type}) \\
\textit{type} & ::= & \textit{ptype} \texttt{ | } \textit{ftype} & (\textit{type})
\end{array}$$

Atomic expressions are either basic integer or string values or symbols. Note that symbols include the predefined values `A` and `B` of type `Player` plus a number of predefined operations. Not that the case for infix application includes the Haskell notation for array lookup `board!(x,y)`.

Expressions

$$\begin{array}{rcll}
\textit{expr} & ::= & \textit{int} & (\textit{integer}) \\
 & | & \textit{string} & (\textit{string}) \\
 & | & \textit{Name} & (\textit{symbol}) \\
 & | & \textit{name} & (\textit{variable}) \\
 & | & \texttt{(}\textit{expr}\texttt{)} & (\textit{parenthesized expression}) \\
 & | & \texttt{(}\textit{expr}_1\texttt{, ... ,}\textit{expr}_k\texttt{)} & (\textit{tuple, } k \geq 2) \\
 & | & \textit{name}\texttt{(}\textit{expr}_1\texttt{, ... ,}\textit{expr}_k\texttt{)} & (\textit{function application}) \\
 & | & \textit{expr binop expr} & (\textit{infix application}) \\
 & | & \texttt{let } \textit{name} \texttt{ = } \textit{expr} \texttt{ in } \textit{expr} & (\textit{local definition}) \\
 & | & \texttt{if } \textit{expr} \texttt{ then } \textit{expr} \texttt{ else } \textit{expr} & (\textit{conditional}) \\
 & | & \texttt{while } \textit{expr} \texttt{ do } \textit{expr} & (\textit{while loop}) \\
\textit{binop} & ::= & \texttt{+ | - | == | ! | ...} & (\textit{binary operation})
\end{array}$$

Here are several predefined operations and values and their types. This list is preliminary and is likely to change.

```
-- User input
input : Input

-- Board and Player updates
place : (Symbol,Board,Position) -> Board
next  : Player -> Player
```

```
-- Board predicates
free   : Board -> Positions
isFull : Board -> Bool
isFull(b) = countBoard(Empty,b) == 0

inARow      : (Int,Symbol,Board) -> Bool
countBoard  : (Symbol,Board) -> Int
countRow    : (Symbol,Board,Int) -> Int
countColumn : (Symbol,Board,Int) -> Int
```