AGPL: A Game Programming Language

Matthew Eads

14 December 2015

Abstract

This document presents AGPL (A Game Programming Language), a simple language that can be used to specify a range of board games. AGPL is embedded in Haskell, and provides a means for a programmer to specify the rules of a given board game and get back a playable binary file from the compiler.

The current lack of a language that provides sufficient flexibility and has sufficient ease of use is first discussed in this paper. As AGPL is introduced, this paper details its implementation–how it is implemented in Haskell, and what it provides to the user.

AGPL provides a solution to the problem of translating the rules of board games, as they are understood by humans, to a format which a computer can use. AGPL succeeds in providing a concise means to describe simple board games, but ultimately more time and work is needed to give it the expressiveness and flexibility to be usable for implementing more complex games.

1. Programming Simple Games

Many people learn to play board games from a very young age, starting with simple games such as Tic Tac Toe and progressing to more complicated games such as Catan and Chess. Board games are an integral part of many a childhood, and many a family gathering even after progressing into adulthood.

Every one of these board games has its own set of rules describing exactly how the game should be played. Naturally the rules for some games are much simpler than for others, but for a game like Tic Tac Toe, the rules are simple enough to be understood by a young child.

Explaining simple concepts to a computer however, is often not as simple; many concepts that are very easy for a human to understand when talking about board games (having x pieces in a diagonal, how to make a 'move'), are not as easy to understand for a computer.

A programming language aims to bridge the gap between what is easy for a computer to understand, and what is easy for a human to understand. For board games, simply programming in traditional languages, whether in C, Python, or Haskell, is insufficient. The abstractions provided by these languages are useful for many purposes, but for board games specifically, there is still a great cognitive load on the programmer who tries to translate the rules of a board game (written by humans for humans) to something the computer can understand and use.

AGPL aims to solve this problem by allowing a programmer to simply enter the rules of a given board game, which AGPL will then translate and use to produce a playable game and AI. Embedded in Haskell, the game designer still needs to understand basic Haskell (more knowledge would be needed for more complex games), but AGPL significantly reduces the amount of boiler-plate code required, and more importantly greatly reduces the cognitive load on the programmer by providing sufficient abstractions to easily implement simple board games.

2. Existing Work on Game Programming

A number of projects currently exist which try to solve the same problems as AGPL. They do these through different means, but ultimately they all suffer from some common issues, namely a limited scope (only supporting a small subset of games),

```
(game
  (title "Tic-Tac-Toe")
  (players X O)
  (turn-order X O)
  (board
     (grid
        (start-rectangle 16 16 112 112)
        (dimensions ;3x3
           ("top-/middle-/bottom-" (0 112))
           ("left/middle/right" (112 0)))
        (directions (n -1 0) (e 0 1) (nw
           -1 -1) (ne -1 1))))
  (piece
  (image X "x.bmp" O "o.bmp")
     (drops (add-to-empty)))
  (draw-condition (X O) stalemated)
  (win-condition (X O)
    (or (relative-config man n man n man)
        (relative-config man e man e man)
        (relative-config man ne man ne man)
        (relative-config man nw man nw
           man))))
```

Figure 1. Implementation of Tic Tac Toe in Zillions of Games, adapted from example file TicTacToe.zrf

and limited support for customization which leads indirectly to not supporting more complex games.

2.1 Zillions of Games

Zillions of Games, created by Mark Lefler and Jeff Mallett in 1998, is a Lisp-like language that provides a simple way to define a variety of board games. However, although Zillions of Games can support a huge number of games, it does not offer support for card games or more complicated board games (involving pieces that can occupy multiple positions for example). Figure 1 shows a truncated implementation of Tic Tac Toe, and demonstrates the conciseness and relatively straight-forward structure of a game definition in Zillions of Games.

The challenge that Zillions of Games faces in card games specifically, is the conflict in supporting both complete information games and limited information games. In complete information games, everyone 'knows' everything; there is no hidden information. The is opposed to limited information games, in which different players can 'know' different things, such as the contents of their hand. Aside from a number of tricky implementation details (how do you display hidden information to only some users), the biggest issue faced is writing an AI that can play limited information games without cheating and looking at information which is supposed to be hidden.

2.2 Joker, GDL, and Libraries

Joker and GDL (Game Description Language) are two other languages that can be used to specify simple games. Joker is implemented in Java, and supports writing simple card games, whereas GDL is a variant of Datalog, and can be used to specify a similar set of games as Zillions of Games.

GDL is significantly less user-friendly than Zillions of Games, and again suffers from limitations in scope of supported games, and ability for the programmer to add complex custom declarations. Joker creates a number of useful abstractions for card games, but has absolutely no support for board games.

Various libraries exist in existing general-purpose languages, providing some abstractions, but are often more focused on the front-end of game design, and do not provide sufficient abstractions to reduce the cognitive load of the programmer, instead only reduce the amount of boilerplate code needed.

3. AGPL

3.1 Goals

As mentioned earlier, the key idea behind AGPL is providing the ability for users to create playable games by only having to provide a description of the rules; limiting the amount of boilerplate code and the cognitive load of translating human rules to computer instructions as much as possible.

More specifically, for simple games such as Tic Tac Toe and Connect Four, the programmer should only need to know a limited amount of Haskell. But by allowing the programmer to use Haskell, and by providing the tools to customize and expand the language, a more experienced Haskell programmer would be able to program much more complicated games. Balancing the need to allow for as much flexibility as possible so that a wide range of games is supported, but not giving too much flexibility such that the language is too difficult to use or does not offer significant gains over simply coding in Haskell or Python is an important decision in

```
⟨game⟩ ::= 'Gamestate:{' [⟨gamestate⟩] '}'
      'Player:{' ⟨Dec⟩ '}'
      'Move:{' ⟨Dec⟩ '}'
      'isVailid:{' ⟨Exp⟩ '}'
      'possMoves:{' ⟨Exp⟩ '}'
      'outcome:{' ⟨Outcome⟩ '}'
      'initialState:{' ⟨InitState⟩ '}'
      'fromString:{' ⟨Exp⟩ '}'
      '$' [⟨Dec⟩] '$' (Custom declarations)

⟨gamestate⟩ ::= 'Board:{' ⟨BoardDec⟩
   | 'Piece:{' ⟨Dec⟩
   | 'Hand:{' ⟨Dec⟩
   | 'Turn:{' ⟨Dec⟩
   | ⟨string⟩':{' ⟨Dec⟩

⟨BoardDec⟩ ::= '{Matrix['⟨int⟩'][' ⟨int⟩']}'
   | '{Array['⟨int⟩']'
   | '<<' ⟨Dec⟩ '>>' (Custom Type)

⟨Outcome⟩ ::= '{winCondition:{' ⟨Exp⟩ '}'
      'tieCondition:{' ⟨Exp⟩ '}'
      'else:{' ⟨Exp⟩ '}'
   | '<<' ⟨Exp⟩ '>>'

⟨InitState⟩ ::= 'Board:' ⟨BoardInitDec⟩
      'Turn:{' ⟨Exp⟩ '}'

⟨BoardInitDec⟩ ::= '{ all' ⟨Exp⟩ '}' (Initialize board to
      piece)
   | '{' ⟨Exp⟩ '}' (Initialize board to List literal)
   | '<<' ⟨Exp⟩ '>>' (Custom initialization func.)

⟨Exp⟩ ::= ⟨Template Haskell Expression⟩

⟨Dec⟩ ::= ⟨Template Haskell Declaration⟩
```

Figure 2. AGPL Syntax

designing the features in AGPL. Using escape syntax has been a simple way to get the best of both worlds, allowing users to use the simplified AGPL syntax and structure, or using Haskell directly to write more complex code if necessary.

With a valid AGPL file, the compiler will be able to produce a playable binary file, an example use of which can be seen in appendix A. The language also has a framework for creating an AI, although this is not yet implemented (this will be discussed in more detail in section 5).

3.2 Grammar

The formal AGPL syntax can be seen in figure 2. An AGPL program contains exactly one game. The game itself is made up of a combination of declarations which correspond closely to the rules of the game. The Gamestate definition defines the properties of the game: what the board is, what a piece is, etc. The Player and Move declarations define how each is represented. The isValid function and outcome functions are used to test if a given move is allowed, and what its outcome is (a win, tie, or another gamestate), respectively. The fromString function takes a string and returns a move, this is used to parse input from the command line. The possMoves function should return a list of all the possible moves, which is needed to build an AI, although if this is left undefined, then no AI will be produced, but the code will still work. Finally, any code between two dollar signs will be parsed as pure Haskell code and spliced in with everything else, this is useful for declaring helper functions.

Diving into some additional detail, the board declaration can be either a matrix or an array (specifying the dimensions for each), or a custom structure. These two structures represent most simple board games, and we are able to provide a number of helper functions if we know that either a matrix or an array will be used, and their sizes. The outcome function declaration is also simplified into specifying the win and tie conditions, and then what to do if neither are met. This is a more natural way of thinking about an outcome, and simplifies the code from the programmers perspective. Lastly, the initialization is also simplified by expressing whose turn it is to begin, and what the initial state of the board is. The initial board can be set to be all one piece, a 2D or 1D list literal (for matrices and arrays, respectively), or a custom initialization function that returns a new board.

3.3 Tic Tac Toe Example

Tic Tac Toe is used as an example throughout this paper due to its simplicity, as well as its popularity. As a reminder, here is an overview of the rules of Tic Tac Toe:

- Tic Tac Toe is played on a 3x3 board.
- There are two players, X and O.
- A player can make a move by placing a piece (X or O) on an empty position on the board.
- A player wins by having 3 pieces in a row, column, or diagonal.
- A tie occurs when the board is full, and no player has won.

```
import Agpl_lib
TicTacToe: Game
Gamestate: {
        Board: {Matrix[3][3]}
        Piece: {X | O}}
Player: {PX | PO}
Move: {(Int, Int)}
isValid: { (\(Move (i, j)) -> ((inBounds
    (i,j)) && (isEmpty (i,j) game)))}
possMoves:{
mfold (\((i, j), piece, a) ->
       if piece == Nil then (Move(i, j):a)
       else acc) [] (board game)}

outcome: {
  winCondition: { (inRowColOrDiag 3 PX
      (board game)) || (inRowColOrDiag 3 PO
      (board game))}
  tieCondition: {isFull (board game)}
  else: {place game (playerToPiece
      (currentTurn game)) (mtoc move)}}

initialState:{Board: {all Nil}
          Turn: {PX}}
--Board: <<matrix 3 3 (\(i,j) -> Nil)>>
--Board: {[[Nil, Nil, Nil],
--       [Nil, Nil, Nil],
--       [Nil, Nil, Nil]]}

fromString: {\(s) -> let move = L.map read
    (splitOn "," s) in (Move ((L.head
    move), (L.head (L.tail move))))}

$
pieceToPlayer :: Piece -> Player
pieceToPlayer X = PX
pieceToPlayer O = PO
playerToPiece :: Player -> Piece
playerToPiece PX = X
playerToPiece PO = O
mtoc :: Move -> (Int, Int)
mtoc (Move c) = c
$
```

Figure 3. Example implementation of Tic Tac Toe in AGPL

The goal of AGPL has been to provide a way to write these rules in as concise a manner as possible. Figure 3 shows the implementation of these rules in AGPL.

Each field translates directly to a field specified in the grammar, described in the previous section. AGPL borrows heavily from Haskell, so most of the code written is valid Haskell code. There are a number of helper functions that are used here, such as inRowColOrDiag, and inBounds. These two represent the two 'classes' of helper functions; those which are imported from the standard library (or other libraries), and those which have been generated at compile time.

The function inRowColOrDiag is a simple example of the imported class of helper functions, and its implementation can be found in `Agpl_lib.hs`. The inBounds function on the other hand is generated at compile time. Using the board declaration, the compiler will either generate a function which takes in two integers and checks them against the matrix size specified in the board declaration, or takes in one integer and checks it against the array size.

We can also see how the rules specified earlier translate to AGPL code. The first rule is seen in the 3x3 matrix board declaration, the second rule follows in the declarations of a player and a piece. The move declaration shows that a player has to specify a coordinate on the board. The 'isValid' function and the 'else' condition in the outcome declaration implements the rest of the rule – a move has to be in bounds, and the position has to be empty, if it is, then the piece corresponding to the player is placed on the board. The fourth and fifth rules are then implemented in the win and tie conditions of the outcome declaration.

4. Implementation Details & Compilation Process

AGPL is implemented as an embedded domain specific language in Haskell. AGPL code can be quasiquoted into a Haskell file, either directly with `[agpl|<agpl code>|]` or by using a file with `[agpl_f|file.agpl|]`. AGPL can only be quasiquoted as declarations; attempting to use the AGPL quasiquoter in place of a pattern, expression, or type will cause an error. The compilation pro-

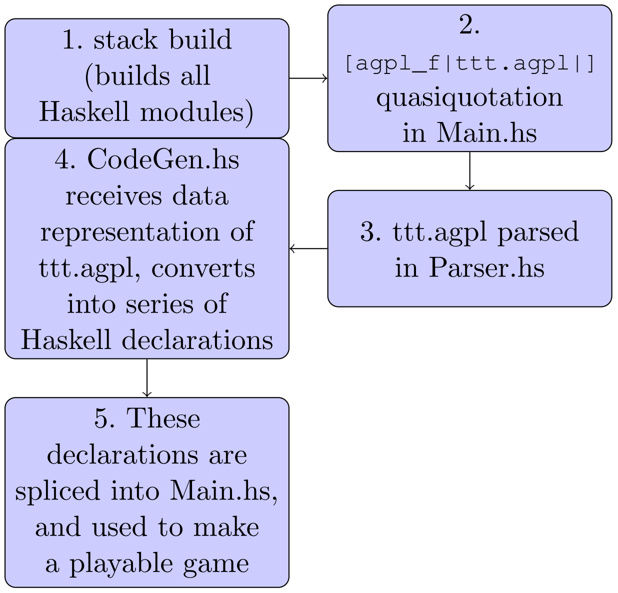
1. stack build (builds all Haskell modules)
2. [agpl_f|ttt.agpl|] quasiquotation in Main.hs
3. ttt.agpl parsed in Parser.hs
4. CodeGen.hs receives data representation of ttt.agpl, converts into series of Haskell declarations
5. These declarations are spliced into Main.hs, and used to make a playable game

Figure 4. Compilation Process

cess is shown in figure 4. After creating an AGPL program (say ttt.agpl), the programmer compiles with ghc to make the binary game file. Currently AGPL uses stack as its package manager, so running `stack build` will compile all the necessary modules (further documentation on using stack can be found in the readme). GHC will compile all the Haskell code, but once it reaches the agpl quasiquotation, it will call the quasiquoter, splice in the declarations provided by the quasiquoter, and move on.

The quasiquoter first parses the string given to it by ghc; using the Parsec parsing monad, the AGPL parser will convert a valid AGPL file into a data structure representing an AGPL program. This data structure is defined in `Agpl_syntax.hs`, and mirrors the structure of the formal grammar.

In the fourth step, the parsing monad sends the data structure representation of the program to the code generation module, `CodeGen.hs`, which is responsible for converting an AGPL program into Haskell declarations. Much of an AGPL program can be converted simply into Haskell declarations, for example the isValid declaration on line 8 in figure 3 is valid Haskell code, and only needs to be wrapped in a declaration assigning it to a function name. However some sections are slightly more complicated; depending on the type of structure given, the code generation module will produce different declarations. The outcome declaration is one of the more complicated functions, in which the three separate parts (win/tie/else conditions) have to be combined to make one outcome function. Information from the AGPL file, such as the type of board being used (matrix or array), can also be used to generate additional declarations to help the AGPL programmer. The inBounds and isEmpty functions for example both rely on knowing what type of board is being used.

Once the code generation module finishes, it returns a list of Haskell declarations, which are spliced directly into `Main.hs`, where the original quasiquotation occurred. The main file itself is very simple, it asks for a move from stdin, which is converted into the move data structure with the provided fromString function. It then checks if the move is valid, and if it is it processes the outcome of the move, with the functions provided from AGPL. If someone won, or the game ended in a tie, the program exits, otherwise it continues.

5. Evaluation

5.1 Ease of Use and Functionality

The main goal of AGPL has been to provide a way to concisely declare board games, and to do so without having too large a cognitive burden translating the regular rules of the game to AGPL code. Both Tic Tac Toe (figure 3) and Connect Four (appendix B), have been implemented in fewer the forty lines of AGPL code. Furthermore, the structure of each follows closely a standard definition of the rules of each; the abstractions of the gamestate, isValid, and outcome functions, are the core of the game, and very closely follow how the rules are defined. Additional declarations such as the initialState and fromString are relatively simple and are necessary in the current implementation of AGPL.

These two examples are relatively simple, but the abstractions provided by AGPL, along with the ability to use Haskell and declare helper functions and libraries, should make defining many other board games fairly straightforward. Chess for example is significantly more complicated to play than Tic Tac Toe, but other than having many cases for all the different pieces, there is nothing fundamentally more challenging about Chess that would prevent it from being implemented in AGPL.

5.2 Future Goals

AGPL is the product of only a few months of work, and is thus still very limited and in practice is not as complete or as usable as Zillions of Games and other languages. Although a number of abstractions and helper functions have been provided, AGPL still relies heavily on Haskell, and the programmers ability to use Haskell.

A number of planned features of AGPL have also not yet been implemented: the current graphical interface is incredibly minimal, and needs significant work to be usable for more complex boards with more complex pieces. More useful error reporting would be a very useful tool for AGPL programmers, who currently have to use a bit of trial and error to find out where errors (that are reported in respect to Haskell code) are coming from. An AI would also be an important part of AGPL, not only providing the ability to play against the computer, but a functioning AI could also play itself to generate some simple statistics on the game. Although the AI itself would take time to implement fully, the abstractions from AGPL provide sufficient information and functionality to build an AI around. An AI mainly needs to be able to query for a list of possible moves, calculate the 'value' or outcome of each move, and then make said move; all these requirements are already met from the language perspective, so implementing an AI would not require any significant change to the language itself.

Alongside ease of use, AGPL has also aimed to support as many games as possible, however, due to time restrictions it currently has no real support for card games, particularly complex boards, or other 'limited information' games (in which certain players know things others do not, such as the contents of their hand).

Adding support for more complex games and different types of games is a long-term goal for AGPL. Adding helper functions and additional syntactic sugar is a relatively simple process that will make defining a range of games simpler. Adding support for new features, such as cards and dice, is a notably more complex process, but given sufficient time is perfectly achievable; the structure and flexibility already provided by AGPL should allow such new features to be added without breaking existing parts, slotting easily into the existing framework of the language.

6. Conclusions

It should be clear from the example implementations of Tic Tac Toe and Connect Four that AGPL succeeds in providing a framework to concisely define board games. Although AGPL is far from a finished product, with more time and work to improve existing aspects of the language, as well as add new features, AGPL would be a much more complete language.

Although it is perfectly possible to implement games in C++, Haskell, Zillions of Games, or any other language, AGPL not only reduces the amount of boilerplate code needed (as demonstrated by the conciseness of the example implementations), but it also allows the programmer to explain the game to a computer in a similar way as one would explain to a friend or a child.

7. Appendix

A Example use of Tic-Tac-Toe

```
matt:agpl$ stack build
matt:agpl$ ./main
Player PX's turn.Please enter a move:
1,1
(  X Nil Nil )
( Nil Nil Nil )
( Nil Nil Nil )

Player PO's turn.Please enter a move:
1,2
(  X  O Nil )
( Nil Nil Nil )
( Nil Nil Nil )

Player PX's turn.Please enter a move:
2,2
(  X  O Nil )
( Nil X Nil )
( Nil Nil Nil )

Player PO's turn.Please enter a move:
2,3
(  X  O Nil )
( Nil X  O )
( Nil Nil Nil )

Player PX's turn.Please enter a move:
3,3
Player PX has won!
```

B Connect Four AGPL Implementation

```
import Agpl_lib

ConnectFour : Game

Gamestate: {
          Board: {Matrix[7][6]}
          Piece: {X | O}
}

Player: {PX | PO}

Move :{(Int)}
isValid: {\(Move i) -> slot (board game) i /= (-1, -1)}

possMoves: {undefined}

outcome:{
       winCondition: { (inRowColOrDiag 4 PX (board game) || inRowColOrDiag 4 PO (board game))}
       tieCondition: {(isFull (board game))}
       else: {place game (playerToPiece (currentTurn game)) (slot (board game) (mtoc move))}}

initialState:{Board: {all Nil}
           Turn:{PX}}
fromString: {\(s) -> Move (read s)}

$
slot ::Board -> Int -> (Int, Int)
slot b i = let c = getRow i b
        in case V.elemIndex Nil c of
            (Just j) -> (i, j+1)
            (Nothing) -> (-1, -1)
mtoc :: Move -> Int
mtoc (Move c) = c
playerToPiece :: Player -> Piece
playerToPiece PX = X
playerToPiece PO = O
$
```

C Parser Module

The parsing module is responsible for converting the AGPL file, passed to it as a string, into a data structure representing the AGPL program. First I have included the declaration of the data structure representing a AGPL program, from `Agpl_syntax.hs`. This data structure mirrors the formal grammar.

```
data Game = Game (GameID, GameState, Move, IsValidFun, PossMovesFun,
                  OutcomeFun, InitState,
                     Player, FromString, [Dec], [Dec])
          | NIL deriving Show


data GameState = GameState {board :: Board, piece :: Dec, hand :: Dec, turn :: Dec}
                  deriving Show
data Board = Matrix (Dec, (Integer, Integer))
          | Array (Dec, Integer)
          | Board (Dec) deriving Show

data OutcomeFun = CustOutcomeFun (Exp)
              | OutcomeFun {wincon :: Exp,
                 tiecon :: Exp, elsecon ::
                 Exp} deriving Show

type GameID = String
data Move = Move (Dec) deriving Show
data Player = Player (Dec, Int) deriving Show
data IsValidFun = IsValidFun (Exp) deriving Show
data PossMovesFun = PossMovesFun (Exp) | PMNil deriving Show
data FromString = FromString (Exp) deriving Show
data CustomDataType = CustomDataType (Dec) deriving Show
data InitState = InitState {boardInit :: Exp, turnInit :: Exp} deriving Show
```

The entire parsing module is over 500 lines of code, so only a snippet is included here. The code below is responsible for parsing the initial board state; the board initialization can be one of three things, setting the board to be initialized to one piece, a list literal, or a custom function. The first two are also dependent on the previous declaration of the type of the board (matrix or array).

```
parseInitBoard :: Board -> Parser Exp
parseInitBoard board = try (parseBoardAll board) <|> try (parseBoardLit board) <|> parseCustomBoard

parseCustomBoard :: Parser Exp
parseCustomBoard =
 do {
  ws; m_reservedOp "<<";
```

```
    e <- (manyTill anyChar (try (string
        ">>")));
     case parseExp e of
     (Left err) -> do {trace ("error1 " ++
        e ++ "\nerror: " ++ err) (return
        undefined)}
     (Right exp) -> do {return exp}
  }

parseBoardAll :: Board -> Parser Exp
parseBoardAll (Matrix (d, (i, j))) =
  let s = "matrix " ++ (show i) ++ " " ++
      (show j) ++ " (\\(i,j) -> "
  in do {
     ws; m_reservedOp "{"; ws;
    m_reserved "all"; ws;
    piece <- (many (noneOf "}"));
    m_reservedOp "}";
    case parseExp (s ++ piece ++ ")") of
      (Right e) -> do {return e}
      (Left err) -> do {(trace ("error: " ++
         err)(return undefined))}}

parseBoardAll (Array (d, i)) =
  let s = "generate " ++ (show i) ++ " (\\i
       -> "
  in do {
    ws; m_reservedOp "{"; ws;
    m_reserved "all"; ws;
    piece <- (many (noneOf "}"));
    m_reservedOp "}";
    case parseExp (s ++ piece ++ ")") of
      (Right e) -> do {return e}
      (Left err) -> do {return undefined}}
parseBoardAll _ = do {m_reserved "ERROR";
    return (VarE (mkName "nil"));}

parseBoardLit :: Board -> Parser Exp
parseBoardLit (Matrix _) =
  do {
    ws; m_reservedOp "{"; ws;
    lists <- (many (noneOf "}"));
    m_reservedOp "}";
    case parseExp ("M.fromLists " ++ lists)
       of
      (Right e) -> do {return e}
      (Left err) -> do {return undefined}}
parseBoardLit (Array _) =
  do {
    ws; m_reservedOp "{"; ws;
    list <- (many (noneOf "}"));
    m_reservedOp "}";
    case parseExp ("V.fromList " ++ list) of
      (Right e) -> do {return e}
      (Left err) -> do {return undefined}}
parseBoardLit _ = do {m_reserved "ERROR";
    return (VarE (mkName "nil"))}
```

D Code Generation Module

This first function takes in the structure produced by the parsing module, it then converts it to a series of Haskell declarations, and returns a list of Haskell declarations.

```
makeAGPLDecs :: Game -> Q [Dec]
makeAGPLDecs (Game (id, gs, m, ivf, pmf,
    ocf, is, p, fs, cd, imports)) =
   do {
     gsdecs <- gamestateDec gs;
     ttype <- turnTypeDec;
     gsdec <- gsDec [boardT,turnT];
     initStateDecs <- initStateDec is;
     move <- moveDec m;
     player <- playerDec p;
     isValid <- isValidDec ivf;
     possMoves <- possmovesDec pmf;
     outcome <- outcomeDec ocf;
     fromS <- fromStringDec fs;
     inBounds <- inBoundsDec (board gs);
     isEmpty <- emptyDec (board gs);
     place <- placeDec;
     return (imports ++ gsdecs ++
         initStateDecs ++ ttype ++ move ++
         player ++ isValid ++ gsdec ++
         outcome ++ possMoves ++ fromS ++ cd
         ++ inBounds ++ isEmpty ++ place);
   }
```

This is an example of one of the functions that produces Haskell code from the AGPL program. It is responsible for creating the inBounds function, which checks if a given position is within the declared bounds of the board. Naturally, the definition is dependent on the board being either a matrix or an array, and the distinction between the two is clear here.

```
inBoundsDec :: Board -> Q [Dec]
inBoundsDec (Matrix (d, (x, y))) =
   [d| inBounds (x1, y1) = ((x1 <= x) &&
       (x1 > 0) && (y1 <= y) && (y1 > 0))|]
inBoundsDec (Array (d, x)) =
   [d| inBounds x = (x <= x && x > 0) |]
inBoundsDec _ = do {return []}
```

E Using the Declarations

Finally, these declarations are used by `Main.hs` to play the game. Currently, this is a very simple process, and the entire file is copied below.

```
[agpl_f|ttt.agpl|]

getMove :: GameState -> IO Move
getMove gs = do {
   putStrLn ("Player " L.++ (show
       (currentTurn gs)) L.++ "'s turn."
       L.++ "Please enter a move:");
   m <- getLine;
   return (fromString m);
   }

won :: Player -> IO ()
won p = do {
   putStrLn ("Player " L.++ (show p) L.++ "
       has won!");
   return ();
}

tie :: IO ()
tie = do {
   putStrLn "The game has ended in a tie.";
   return ();
}

playGame :: GameState -> IO ()
playGame gs = do {
   m <- getMove gs;
   if isValid gs m then
      let (result, i) = outcome gs m
      in (case result of
         (Win p) -> (won p)
         (Tie) -> tie
         x -> (trace (show (board x))
             (playGame x)))
   else do {
      putStrLn "Invalid move, try again.";
      (playGame gs);
   }
}

main :: IO ()
main =
   let gs = GameState{board = (boardInitF),
       currentTurn=turn} in
      do{
         playGame gs;
         return ();
      }
```

References

N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth General Game Playing: Game Description Language Specification. Stanford Logic Group, Stanford University, Stanford, CA. 2008. http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf

Zillions Development Corp. Supported FAQ Page. http://www.zillions-of-games.com/supportedFAQ.html, 1998