Report Project Programming Languages

Rien Maertens

May 2017

Contents

1 Introduction

This report is made on behalf of the course Programming Languages, taught by professor Eric Laermans. There were two assignments that had to be done:

1. A chess-like game had to implemented in the Oz programming language using only the declarative model and the message-passing concurrency model.
2. The first implementation had to be adapted to communicate with other students' implementations and to facilitate an extra rule added to the game.

Development of the assignments was done on a laptop with an Intel i7 Haswell processor running Arch Linux. Neovim was the main editor used to write the source code and this report. An oz.vim plugin for this editor was of great help. The source code was built with the Mozart 2 programming system together with GNU Make to simplify conditional compilation and execution. This report was written in markdown and converted to pdf (trough LaTeX) with Pandoc.

I will begin with describing how I tackled the assignments while explaining some of the design choices I made. Then, a more detailed breakdown of the implementation is given. Finally, I conclude by summarizing the insights I gained by completing these assignments.

2 Assignments

2.1 Assignment 1

The board

I started this assignment by implementing a module to represent and manipulate a board to play the game on. This seemed like a good basis to start with and allowed me to learn the specifics of the Oz language while writing this module.

Initially a board was represented by a list of lists of atoms. Because there were going to be a lot of indexed reads on the board I searched for an alternative to the built-in linked lists, which have linear access time. I looked at arrays (constant, but not declarative), quadtrees (logarithmic, but not built-in and a lot of work to implement) and tuples (constant and built-in).

I did a benchmark accessing a million items in both tuples and lists. Where lists would take five minutes to read all the items, tuples would need less than a second. Needless to say, it was an easy decision to convert the board from a list of lists to a tuple of tuples. You can find this benchmark in the file `List_vs_Tuples.oz`.

The referee

The next step was to write a referee module. Because the referee has to keep track of the board and the progress of the game, it has to have some kind of 'state'. Originally this was done by using `fold` on the list of incoming messages. But as the referee became more complex I wrote a function that recursively iterates. In general the flow of a move is as follows:

1. A message from a player is received.
2. The referee checks whether it is the turn of the player who sent the message.

3. The referee checks whether the move the player sent is valid.
4. The move is executed on the board and the referee checks if the game is finished.
5. If the game isn't finished, the other player is sent the new board and a request for a new move.

At the start of the game, each player is given its own port to communicate with the referee. Two threads proces the contents of these ports. They wrap the message of each player in a record together with an atom indicating the player and send that record to the main referee port. The messages are essentially multiplexed. This way, the referee knows which player is currently talking, preventing a player from pretending to be the other. When such behavior is detected, the player who was pretending to be the other loses.

After each turn, the referee calculates the full board and all the possible next moves. This information is used to decide whether the game has ended or not. To mitigate duplicate effort, a move request sent to a player was accompanied by this information, since a player then simply had to choose which one of those moves he thought was best. This also removes the need for an internal state for a player. This changed when in the protocol we chose for more minimalistic communications.

The player

Finally, a player module was written. The way a player thread and port created is analogous to a referee. The first implementation was very simple: when the next move is requested, the first item in the list of valid moves was sent back to the referee. My plan was to develop a smarter strategy together with the second assignment.

Declarativity

I've found declarative programming and its surrounding concepts to be really powerful. For example, single assignment: the fact that variables can only be bound once and their value never changes makes it simpler to reason about the origins of a certain value found in a variable. The fact that one can trust that functions don't have side effects is a huge advantage over object oriented programming. However, declarative programs tend to be more complex to do simple things like conditionally adding multiple items to a list. There will always be a trade-off between simplicity and complexity, but I personally prefer reasoning above quickly hacking a program together.

Another advantage was the powerful dataflow behavior: you can just trust that the program execution will stop and wait automatically until a variable is bound, instead of worrying about latches, semaphores and other synchronisation techniques. Unfortunately, it is still possible to create deadlocks and the way Oz

handles deadlocks is also very confusing: exiting without an error or an exit status different from 0.

The main drawbacks I experienced with this programming language had more to do with the way the syntax and the compiler than the concepts behind the language. For example, the fact that every variable had to be capitalized, just like functions, was confusing at first. The compiler is also less clever in communicating what exactly goes wrong. As a consequence, searching what causes a particular error can be time consuming.

The fact that Oz is dynamically typed is a curse as well as a blessing. Programs can be written more quickly, but the result is more error prone. I prefer Haskell in that aspect: the (very) strong typing together with a concise type checker has as a consequence that if your program compiles, it is most likely correct. Whereas if your Oz program compiles, there is no guarantee that it is correct.

Another drawback is efficiency. Arbitrarily accessing an item on the board happens in constant time. However, when an item has to be replaced (i.e. executing a move), the whole board has to be copied, taking linear time. When using a nondeclarative language arrays could be used, where replacing items is also constant.

Message-passing Concurrency

The way ports are exchanged between the referee and the players is loosely based on the examples in sections 5.2.1 and 5.2.2 in the book (*Concepts, Techniques and Models of Computer Programming*). First, each player port is created with `Player.createPlayer` and the variable where the port to the referee will be bound to is given as an argument. Then the referee is created with `Referee.createReferee`, this binds the referee port to the variables. The messages between the threads are then processed as if they were normal list items.

This way of information exchange between threads felt very intuitive once I realised that ports are just lists. This way of sending messages also creates the opportunity for a player to immediately start reasoning about which move to take next just after it sent its response to the referee. The communication between the referee and a player also goes trough a single channel. Both implementations are therefore very independent of each other, which eases the integration of other people's code in assignment 2.

Unfortunately, I found out a beter way to communicate with the when it was too late. Instead of creating two ports per player (one in each direction), one can create only the port from the referee to a player and embed a response variable in each request sent by the referee. The referee can then implicitly wait until that response variable is bound.

However, the threads are actually sequential. Each thread had to wait for a previous one to send its next message, which always happens when a function

is finished. There is no real need for concurrency here. The communications could therefore just as well be done by directly invoking a function instead of sending it trough a port.

2.2 Assignment 2

Implementing an extra rule

The extra rule came down to the addition of some extra stages within the game. I had to add the current stage to the state of the referee and the player had to take account for more kinds of requests. I think I solved this cleanly by giving the referee different `Judge`-functions that have the responsibility over different stages of the game. The player solved this in a similar way.

It was easy to extend my own code to facilitate the extra rule. I estimate that it took a day's work to implement it properly. This was partly because the player had a very simple implementation and I only started creating a 'smart' AI when all the necessities were already implemented.

The extra rule of eliminating your own pawns gave an interesting strategic twist to the game. I decided to eliminate my own pawns in such a way that there is almost one empty place between each pawn. If an opposing pawn then reaches the other side, there is always at least one of pawn that can take the adversary.

In my opinion, the declarative model eased extending the existing implementation. Few bugs occurred when changing the code and I think the reason behind this is the fact that functions often do not have side effects. With OO-programming it is not transparent which of the variables change when a function or method is called, while in declarative programming one can trust that bound variables will stay the same while a function is executed.

Integrating other students' code

When the group composition was revealed, we first gathered everyone in Slack (a team collaboration tool) and then agreed to create a protocol which we would use to let out programs communicate with each other. I created a repository to facilitate easy collaboration, but I ended up adapting the protocol I wrote for the first assignment by myself while taking in account the opinions of my group members. Some decisions we made:

- Instead of sending the full board to the player, the response of the other player is sent to the current player. This way the communication is more minimalist and independent of the internal representation of a board.
- The player that chooses how many eliminations have to be done also sends his first elimination in his response.

- Every message sent by the referee is wrapped within a record `r(other:X request:Y)`, where `X` is the other player's answer and `Y` is the request, except of `gameEnded` which tells the players the game is finished.

I included the complete protocol in **Appendix A**.

I was pleasantly surprised that everyone was able to implement this protocol and adjusting my code to work nicely with other players was a breeze.

3 Implementation

I will now give a more concise breakdown of the functions implemented in each functor. If something is not clear, you can always check the source file itself. I try to document my code where needed.

3.1 Helper

This is an extra functor where some I chose to put some small helper functions:

- `IsEmpty` returns `true` when a list is empty, `false` otherwise.
- `OtherPlayer` returns `p1` when `p2` was given as argument and vice versa.
- `DirectionFor` returns `+1` or `-1` depending on what has to be added to a pawn's position to do a move.
- `MakeListWith` returns a list of the given length, where each item is bound to the second argument of this method.
- `MakeTupleWith` the tuple equivalent of the function above.
- `Join` given a list of strings and a separator (string), concatenates each string in the list interleaved with the separator.
- `JoinTuple` the pule equivalent of the function above.

3.2 Board

As previously stated, a board is represented by a tuple of tuples of atoms (`p1`, `p2` or `empty`. Where possible, I tried to use functions like `map` and `fold` to manipulate the board instead of iterating with `for`. The following functions can be found in the `Board` functor:

- `Init`: takes two arguments `N` and `M` and generates a board with `N` rows and `M` columns with the pawns of each player on its starting row.
- `Show`: procedure which prints the board contents to the standard output.
- `Set`: takes a pawn atom, a row, a column and a board as arguments and returns a new board with the new pawn placed on the given row and column.

- `DoMoveFor`: takes a player atom (`p1` or `p2`) a move and a board and returns a new board where this move is performed.
- `ValidMovesFor`: takes a player and a board and returns all the valid moves for this player.
- `Analyze`: takes a board and returns a record with for each player all his valid moves and a boolean whether that player reached the other side or not.

One could argue that the last two functions (`Analyze` and `ValidMovesFor`) should be moved to the `Referee` functor because they essentially dictate who has won and which moves are valid. However, since a player would probably want to use these functions it seemed logic to keep these functions with the board to keep the player independent from its referee.

3.3 Referee

Almost all of the functionality in this functor is within the `CreateReferee` function: it starts its own thread with its initial state and defines the functions who need to access its arguments: the ports from and to each player. The most important subfunctions are:

- `RefereePort` creates the thread for the referee and returns the port to communicate with it.
- `ProcessMsg`: takes a list of message and an initial state, pops the first message and checks if it is message receiver's turn. If that is the case, it the message to a judge function and recurses with a new state and the rest of the messages.
- `Eliminate` eliminates a pawn from the internal board.
- `JudgeSize` checks if the requested size is within the allowed values and creates the initial board.
- `JudgeFirstElimination` checks if the requested amount of eliminations is allowed and if the elimination is not 0, eliminates the first pawn.
- `JudgeElimination` eliminates the next pawns and checks if all eliminations are done.
- `JudgeMove` checks if a requested move is in the list of possible moves for that player. If it is, the move is executed. If it is not, the current player gets a second chance to submit a move. If it was already the current player's second chance, the game is ended.
- `NextRequest` sends a request to a player.
- `EndGame` ends the game and exits the program.

The body of the `CreateReferee` sends the first `size` request to player 2, starts the referee thread and starts the two multiplexing player threads.

3.4 Player

The player works similarly to the referee: there is one function `CreatePlayer` which defines subfunctions that returns the port to that player. Again an overview of the most important functions:

- `SetOwnValues` is a procedure that binds the own values, as soon as it is known which player this thread is representing and what the board dimensions are.
- `NextMove` calculates the next move to send to the referee. A more detailed explanation will follow.
- `NextElimination` picks the next elimination to send to the referee. The strategy is to make sore there is always at most one empty place between two pawns.
- `ApplyOther` takes the current board and the move of the other player, and executes that move on the board.
- `RespondTo` looks at the request sent by the referee and the current board, calculates what the next response has to be and sends that response to the referee. A new board is returned where the player's own action is applied on.

There is also a function `ProcessRequests` which is where player's port is created and the thread is started. It also checks whether a `gameEnded` is sent by the referee and ends the game appropriately.

Calculation of the next move

This is done by getting the list of all possible moves with `Board.validMovesFor` and scoring each move. Then the move with the highest score is sent to the referee.

Scoring is done as follows:

1. The distance a pawn has already covered is calculated. The closer a pawn is to the other side, the higher the score.
2. If the moves eliminates a pawn of the opponent a bonus is added. This bonus is multiplied with the distance the other pawn has covered. That way, when there is a choice between taking a pawn closer to the own side or a pawn closer to the opposing size, the closer pawn is taken because that pawn poses more of a threat to win.
3. If the own pawn becomes vulnerable by making the move, a flat penalty is added so that such a move will always be taken when there is no other option.
4. If the pawn does not have any opponents on its path or the two columns surrounding it, a bonus is added in such a way that this move will be chosen if it is certain to win.

I think this implementation is rather smart and efficient. And I am satisfied with the result I have achieved for the minimal amount of effort I have put in it.

4 Conclusion

4.1 Declarativity

The declarative model can be very powerful and results in programs that are usually more safe than its nondeclarative equivalents. Programming with dataflow variables and higher order programming is very pleasant to work with. Although it is more complex to reason about, once you get the hang of it, the result is very clean. Unfortunately, sometimes you have to sacrifice some performance. Hopefully this can be compensated with the performance gained with declarative concurrency.

4.2 Message-passing Concurrency

This model makes it easy for threads to communicate with each other. Despite the fact that the different threads in my program are not really parallel it was still useful to communicate with the players and referee. An interface had to be defined because all the information is exchanged trough one channel, which made is very easy to integrate other people's code.

4.3 Comparison with other programming models

I did not have the feeling that the declarative model was an obstruction in writing this program. Of course, some aspects could be solved in a better or more efficient way with other models. Something that I missed in this programming language was a strong type system instead of a dynamic one. Another thing that I think the language needed, was a way to enforce the protocol that we defined. In an OO-language, this would be done by defining an interface, while now we had to trust each other to implement everything properly.

5 Appendix A: Agreed communications

5.1 Protocol

Each player has to create a `Port` to which the `Referee` can send its messages to. Each player receives a `Port` to which it can send the answers to the requests.

The referee and the players communicate with each other by sending values trough these ports.

A `Board` is represented by a grid of `N` rows by `M` columns of cells. These rows and columns are 1-indexed. Player 1 starts on the first row, player 2 starts with all its pawns on the last row. We can represent the initial configuration of a 5x6 board as follows:

```
  1     2     3     4     5     6
1 p1    p1    p1    p1    p1    p1
2 empty empty empty empty empty empty
3 empty empty empty empty empty empty
4 empty empty empty empty empty empty
5 p2    p2    p2    p2    p2    p2
```

With `p1` a pawn owned by player 1, `p2` a pawn owned by `player 2` and `empty` a cell without a pawn.

Messages sent by the referee and the players

All the messages the referee sends to the players are records `r(other:X request:REQ)` where `X` is the last response of the other player and `REQ` is the request of the referee. In the next section, `R` and `C` represent a row and a column number. These numbers are indexed by 1.

1. **The referee** sends the record `r(other:nil request:size)` to player 2.
2. **Player 2** responds with a tuple `size(N M)`. Where `N` and `M` are numbers between 5 and 8 (inclusive). If the values are not within these bounds the referee ends the game and player 1 wins.
3. **The referee** sends the record `r(other:size(N M) request:firstElimination)` to player 1.
4. **Player 1** responds with the record `firstElimination(k:K row:R col:C)`. Where `K` is the amount of pawns each player will have to remove (`K` is 0 the `row` and `col` are ignored). If `K` is greater than `M`/2 or the `row` and `col` do not represent a pawn of player 1 the referee ends the game and player 2 wins.
5. The following steps repeat until every player has eliminated `K` pawns:
   - **The referee** sends the record `r(other:X request:elimination(L))` to next player. Here is `L` the amount amount of eliminations left.
   - **The next player** responds with the tuple `eliminate(R C)`. Again, the selected position should have a pawn owned by the current player.
6. When all eliminations are done, the following steps are repeated: (note that player 1 is the first player to move).
   - **The referee** sends the record `r(other:X request:move(A))` to the next player. `A` is a boolean which is `true` when this is the current player's second chance.

- **The next player** responds with a tuple `move(f(R C) t(R C))`. Where `f(R C)` represents the current row and column of a pawn (**f**rom) and `t(R C)` represents the desired next coordinate of that pawn (**t**o).
- If the response was an invalid move, the referee resends the same request to the current player but with `A = true`. If the player then sends another invalid move the game is ended in favor of the opponent.

7. When the game is over, **the referee** sends the record `gameEnded(winner: P)` to both players. The value `winner: P` is `p1` when player 1 has won the game, and `p2` when player 2 is victorious.

Port creation

It should be possible to create a new game like this:

```
functor
import
    Player
    Referee
define
    PR1
    PR2
    P1
    P2
in
    P1 = {Player.createPlayer PR1}
    P2 = {Player.createPlayer PR2}
    {Referee.createReferee P1 P2 PR1 PR2}
end
```

`P1` and `P2` are the ports the referee can use to send his messages to each player. `PR1` and `PR2` are the ports each player gets to send responses to the referee. It is allowed to deviate a little from this standard, as long as it is easily possible to integrate it with code that is conform to this document.

File naming

All the files should be suffixed with your UGent username. For example: `Player_rbmaerte.ozf` `Board_rbmaerte.ozf` `Referee_rbmaerte.ozf`.