Chesskell: Embedding a Two-Player Game in Haskell's type system

3rd Year Project Progress Report

Toby Bailey

November 19, 2020

Contents

1 Introduction

In 2020, video games are more popular than ever. In the US alone, an ESA report[1] estimates that there are more than 214 million individuals who play games. Considering this, it's surprising how many games are released with major bugs in their software—some of which end up being so notable that news and footage of them appear on mainstream media[2].

As programming languages have evolved, many have begun to address errors at compile time. Features like optional types are being introduced to languages such as Java and C#, and languages like Rust have pioneered ways of safely handling dynamic allocation through the use of owner types[3]. Many language compilers now force the developer to handle classes of errors that previously could only be encountered at run-time, such as null pointer exceptions. However, the most common programming language used for game development is C++[1], despite its' lack of automatic memory management and allowing unsafe pointer arithmetic. C++ is chosen for its' speed, but as computers (and gaming consoles) get more powerful, speed becomes less important than developer productivity. Research into bringing more type-safety into game programming environments is rare, and is the niche this project attempts to fill.

Recent versions of the *Glasgow Haskell Compiler* (GHC) support programming at the type level, allowing programmers to compute with types in the same way that languages like C or Python compute with values[2], using *type families* that emulate functions at the type-level. These computations run at compile time, before an executable of the source code is generated, allowing programmers to transform logic errors into type errors[3].

The aim of this project is to demonstrate a proof-of-concept; that it is possible to model Chess at the type-level, and that compiled programs comply with the rules.

2 Background

The project, nicknamed Chesskell, has the main aim of modelling the classic board game Chess in Haskell's type system. This type-level model will be interacted with via a Haskell-embedded Domain-Specific Language (DSL), for describing games of chess. This Embedded DSL (EDSL) will be modelled on Algebraic Notation, a method of writing down the moves associated with a particular match of chess.

[1]`https://www.theesa.com/wp-content/uploads/2020/07/Final-Edited-2020-ESA_Essential_facts.pdf`
[2]`https://www.bbc.co.uk/news/technology-50156033`
[3]`https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`

3 Related Work

There are many publicly available chess engines and chess-related programs written in Haskell[4][5], including a Haskell wiki article teaching the programming about Haskell using chess as an example[6]. Many of these are chess engines, which take in a board state and output the move (or set of moves) which are strongest, and so therefore performs move checking at the value-level to ensure that the moves that it outputs are valid. Chesskell differs from these in function, in that the end software will not output a list of strong moves; it will simply take in the moves performed, and state whether they are valid chess moves or not. We are not aware of any such type-level chess implementations in Haskell.

There have been allusions to chess at the type-level through solving the N-queens problem in dependently typed languages, such as Idris[4][7]. The N-queens problem makes use of some chess rules, including the Queen's attack positions (a straight line in any direction); but as the end goal is not to successfully model a game of chess, it is not a full type-level chess implementation.

Despite the apparent lack of work on Chess at the type level specifically, there has been work on Haskell-Embedded DSLs in other domains to enforce certain behaviour at compile time. Mezzo[5] is an EDSL for music composition, which checks if the created piece of music conforms to some musical ruleset before compilation of the program. WASH/CGI[6] is an EDSL for type-safe server-side web scripting, which ensures thread-safe server operations and valid XHTML generation.

4 Project Functional Requirements

1. The system will contain a type-level model of chess, which will:
    a) Support all rules within the FIDE official chess rules.
    b) Have an internal representation of the board, the pieces, and other information relevant to the state of play.
    c) Be able to detect when a rule of chess has been broken, and give out a compile-time error to explain which rule was broken and how.
2. This type-level system will run on the Glasgow Haskell Compiler.
3. This type-level system will be accessible via an Embedded Domain-Specific Language, which will:
    a) Describe a game of chess, move by move.

[4]`https://github.com/mlang/chessIO`
[5]`https://github.com/nionita/Barbarossa`
[6]`https://wiki.haskell.org/Learning_Haskell_with_Chess`
[7]`https://github.com/ExNexu/nqueens-idris`

b) Be written in Haskell source files, and be valid Haskell according to the Haskell grammar.

c) Be type-checked by the type-level model, and will cause errors if the game described breaks the established rules of chess.

5 Current Progress

5.1 Chess Types Overview

The software has a full representation of a game of chess at the type-level, explained in more depth below. As the game is checked move by move, the only representation required is the board at the time of checking; containing all pieces on that board.

The most important types are the board, and the individual pieces on it. Both of these types are composed of other types, making them *composite types*; the board is comprised of pieces, and the pieces contain that piece's team, name, and position.

5.1.1 Team and PieceName

Both of these types are simple algebraic data types, with all inhabitants defined in code. The team defines whether a piece is black or white, and the piece's name states which chess piece it is; a pawn, a rook, and so on.

The declarations below, combined with the `-XDataKinds` extension, create two things. The first is a value-level representation of `Team` and `Piecename`, where `Black` is a value of type `Team`. The second is a *type-level* representation, where `'Black` is a type and `Team` is a kind[8].

```
data Team = Black | White
data PieceName = Pawn
            | Bishop
            | Knight
            | Rook
            | King
            | Queen
```

5.1.2 Position

The type for holding a piece's position makes use of two more types, one for columns and the other for rows. In chess, columns are labelled with letters and rows are labelled with numbers; "a1" is the top-left of the board, and "h8" is the bottom-right. The column type is another simple algebraic data type with inhabitants listed. The row type is a type-level implementation of Peano natural numbers.

[8]Note that `'Black` has no term-level values, since it does not have kind `*`

```
data Column = A | B | C | D | E | F | G | H
data Nat where
    Z :: Nat
    S :: Nat -> Nat
```

The Position type itself is a *product type*, a type whose cardinality[9] is equal to the product of the types it takes in. Since `Position` takes in a type of cardinality 8 (`Column`) and a type of infinite cardinality (`Nat`), it has infinite cardinality, even though only 64 of them are valid chess positions.

```
data Position where
    At :: Column -> Nat -> Position
```

5.1.3 The Pieces

Each piece contains information relevant for the rule check: that piece's team, name, and an information type. The information type contains a `Nat` and a `Position`, to represent the number of moves that piece has taken, and its current position on the board. Recording the number of moves the piece has taken is important for several rules in chess, including castling and *En Passant*. As such, it's included in the `PieceInfo` type.

```
-- Pieceinfo type, containing general information
data PieceInfo where
    Info :: Nat -> Position -> Bool -> PieceInfo

-- Piece type, containing the most important information for a piece
data Piece where
    MkPiece :: Team -> PieceName -> PieceInfo -> Piece
```

5.1.4 The Board

At the type level, the most common way of storing n elements is with the built-in list type (similar to the value-level). However, a singly-linked list has issues; how can we ensure that the chess board is the appropriate size without a length check each move? This would take at least 64 operations, since the length must be computed recursively; as well as 7 addition operations to put together the list lengths.

Performing such a check each move would be wasteful. Instead, length-indexed vectors are used. Nat values are used to specify the length of a vector, and it would generate a type error if the vector was any other length. As with most things in Haskell, they are defined recursively; an empty vector has length 0, and you express a vector of length (n + 1) by pushing an element to the front of a vector of length n.

```
data Vec (n :: Nat) (a :: Type) where
    VEnd   :: Vec Z a
```

[9]A type's cardinality is the number of inhabitants it has.

```
        (:->)  :: a -> Vec n a -> Vec (S n) a
```

For example, a vector of 3 boolean types has the type `Vec (S (S (S Z))) Bool`. The board can be expressed using these vectors—a vector of 8 vectors of 8 pieces. (`Maybe Piece`s are used instead, since a board square does not necessary contain a piece.)

```
    type Eight = (S (S (S (S (S (S (S (S Z))))))))
    type Row   = Vec Eight (Maybe Piece)
    type Board = Vec Eight Row
```

Although this is the main board type, it is augmented with a `BoardDecorator`, so named because the intention is to be similar to the decorator design pattern[7] (with the exception that subclassing and superclassing are not features of Haskell). `BoardDecorator` should be used instead of `Board`, since it provides additional information:

- The last team to move;
- The last position moved to;
- The White and Black king positions, stored as a tuple;
- The number of moves in the game thus far.

This information prevents repeated passes over the `Board` which existed in previous versions of the program, to find the last piece to move as well as king positions. While there is the overhead that the `BoardDecorator` must be updated at each move, the code is much conceptually clearer with the use of the additional information.

```
data BoardDecorator where
    Dec :: Board -> Team -> Position -> (Position, Position)
        -> Nat -> BoardDecorator
```

5.2 Checking Chess Rules

All of the rule-checking is done via type families that take in the `BoardDecorator` and output either a correct result, or a type-level error.

5.2.1 Type families in Haskell

Although Haskell allows type-level computation via type families, these type families must be fully saturated. That is, they can't be partially applied like normal Haskell functions can. However, Li-yao Xia's work on First Class Families[10] defines a kind of type-level interpreter, allowing similar behaviour to partial application.

This means that the board can be mapped over, performing computation on the pieces therein without writing a new function for each map. Incrementing a piece's move count, or changing the name of a piece, is much easier with First Class Families than it would otherwise be with regular type families.

[10]https://github.com/Lysxia/first-class-families

5.2.2 Move Lists

Each piece, depending on its team and name, can move to a certain number of spaces. For instance, a King can move a single space in any direction. The `PieceMoveList` type family formalises this, returning a list of spaces that a piece can move to (given a piece and board decorator).

```
data PieceMoveList :: Piece -> BoardDecorator -> Exp [Position]
-- If it's the pawn's first move, move two spaces, otherwise move one
type instance Eval (PieceMoveList (MkPiece team Pawn info) boardDec)
    = Eval (If (Eval ((IsZero . GetMoveCount) info)) (PawnStartMove (MkPiece

-- Bishops move diagonally
type instance Eval (PieceMoveList (MkPiece team Bishop info) boardDec)
    = Eval (AllReachableDiag team boardDec (Eval (GetPosition info)))

-- Knights move in an L-shape
type instance Eval (PieceMoveList (MkPiece team Knight info) boardDec)
    = Eval (AllReachableGivenList team boardDec (Eval (GetAllKnightPositions

-- Rooks move in straight lines
type instance Eval (PieceMoveList (MkPiece team Rook info) boardDec)
    = Eval (AllReachableStraightLine team boardDec (Eval (GetPosition info))

-- Queens can move in any direction
type instance Eval (PieceMoveList (MkPiece team Queen info) boardDec)
    = Eval (AllReachableLineAndDiag team boardDec (Eval (GetPosition info)))

-- Kings can move a single space in any direction
type instance Eval (PieceMoveList (MkPiece team King info) boardDec)
    = KingMoveList (MkPiece team King info) boardDec
```

There is another function, `PieceAttackList`, which is similar to `PieceMoveList` but gives the list of spaces that a piece can attack. This is different from a piece's move list in a handful of cases (for instance, pieces can attack the opposite team's King, but cannot move to its position and take it).

5.2.3 Moving the Pieces

The core movement function, aptly named `Move`, performs a series of checks on the board state, moves the piece, and then performs further checks. The actual movement code is omitted here for brevity; but all moves except for pawn promotion are complete.

```
-- MoveNoChecks moves the piece depending on its' type,
-- handling any relevant side effects
data Move :: Position -> Position -> BoardDecorator -> Exp BoardDecorator
type instance Eval (Move fromPos toPos boardDec) = Eval ((
    CheckNoCheck (GetMovingTeam boardDec) . -- Post-move checks
    MoveNoChecks fromPos toPos .            -- Move the piece
        NotTakingKingCheck toPos .          -- Pre-move checks
        CanMoveCheck fromPos toPos .
        NotTakingOwnTeamCheck toPos .
        NotSamePosCheck fromPos toPos .
        NotLastToMoveCheck fromPos .
        TeamCheck fromPos) boardDec)
```

5.3 EDSL

5.3.1 Proxies and Singletons

Value-level information is not naturally available to Haskell's type-level facilities, due to type erasure. As such, there are various methods to pass such information up to the type level.

Proxy types are parameterised with kind-polymorphic type variable. A `Proxy` value can have type `Proxy Int` or `Proxy 'Black`, for example. Proxies are used to make runtime board decorator information available at the type level in Chesskell.

Singleton types mimic dependent types in that

5.3.2 EDSL Implementation

The EDSL is implemented through a Continuation Passing Style[8] scheme, with inspiration taken from the Flat Builders pattern developed by Dmitrij Szamozvancev[5]. The core idea is that a value is transformed via a series of continuation functions, until the final continuation function returns a value.

```
-- Starts with a value of type t, gets a continuation
-- of type (t -> m), and applies it to the value
type Spec t = forall m. (t -> m) -> m

-- Converts a value of type s into a value of type t,
-- passing that t-value into a continuation
type Conv s t = s -> Spec t

-- Ends the continuation stream
type Term t r = t -> r
```

The chess game starts with a `Proxy` value, with its type containing a `BoardDecorator` type. Continuations are passed, transforming that value, until the chess game ends (or until a rule is broken). The core continuations defined are named after the pieces, such as `pawn` and `king`. Each of them takes in...

5.4 Testing

6 Next Steps

6.1 Revised Timetable/Plan

6.1.1 Weeks 9-10: 30th November to 13th December

This space is set for testing of the (near-complete) system against a curated data set, and fixing any potential problems that arise. The test will involve piping the data set to the type-level model via the EDSL, and as such will be integration and system testing, with unit testing taking place during development.

6.1.2 Weeks 11-14: 14th December 2020 to 10th January 2021

This is the first section of allocated empty space; should some of the risks in the below section materialise, this is when the project may be caught up on. Should the project be going well, then this extra time will be spent either starting early on the dissertation, or adding extensions to the project to further explore the modelling of games at the type-level.

6.1.3 Weeks 15-18: 11th January to 7th February

Writing the dissertation is planned for this stretch of time; it will be planned out, section by section, with any relevant graphs, figures, and citations gathered.

The module CS324 Computer Graphics has a coursework due on the 20th January; as such, planning the dissertation could take longer than expected. However, since a month is set aside for just planning, delays are unlikely.

6.1.4 Weeks 19-22: 8th February to 7th March

Once the previous detailed planning stage is complete, writing shall begin; with a detailed enough plan, this section should not take longer than a month. An initial draft will be completed by the 7th of March.

The project itself will be evaluated during this period, examined for any subtle bugs, and the code will be finalised and completed. Work on the project presentation should also start in earnest, writing out a script.

6.1.5 Weeks 23-24: 8th March to 21st March

These two weeks are more empty space; set aside to act as a buffer for delays in dissertation writing. The project presentation is expected on the 19th March; so this empty space will ideally be spent completing the presentation and rehearsing it.

6.1.6 Weeks 25-29: 22nd March to 25th April

Drafting, re-drafting, and refining the dissertation (with the help of the supervisor) will take place during this time. This is the final stretch, and will be spent ensuring that the final piece of writing is as good as it can be.

Revision for examinations is key during this period; however, since the dissertation should be complete and this period is for evaluation and not the main bulk of writing, there should be ample time.

References

[1] R. Nystrom, *Game Programming Patterns*, p. 6. Genever Benning, 2014.

[2] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães, "Giving haskell a promotion," in *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, pp. 53–66, 2012.

[3] S. Maguire, *Thinking With Types: Type-Level Programming In Haskell*. 2018.

[4] E. Brady, "Embedded domain specific languages in idris," 2015. `https://www.cs.ox.ac.uk/projects/utgp/school/idris-tutorial.pdf`, pg. 51.

[5] D. Szamozvancev, "Well-typed music does not sound wrong," 2017.

[6] P. Thiemann, "An embedded domain-specific language for type-safe server-side web scripting," *ACM Trans. Internet Technol.*, vol. 5, p. 1–46, Feb. 2005.

[7] H. Mu and S. Jiang, "Design patterns in software development," in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pp. 322–325, 2011.

[8] G. J. Sussman and G. L. Steele, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405–439, 1998.