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

3rd Year Project Progress Report

Toby Bailey

November 27, 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 more errors at compile time. Features similar to optional types have been added to languages such as Java[3] and C#[4], and languages like Rust have pioneered ways of safely handling dynamic allocation through ownership types[5]. Many compilers now force the developer to handle classes of errors that previously could only be encountered at runtime, such as null pointer exceptions.

However, catching logical errors in imperative languages is almost always done at runtime. Many software systems use runtime features such as exceptions to discover and deal with errors and misuse of APIs. Enforcement of invariants and rules is typically dynamic; if a check fails, an exception is thrown and potentially handled. However, if a programmer forgets to implement such a check, the behaviour is unpredictable. A 2007 study [1] on Java and .NET codebases indicates that exceptions are rarely used to recover from errors, and a 2016 analysis of Java codebases [2] reveals that exceptions are commonly misused in Java.

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 [3], using *type families* [4] [5] that emulate functions at the type-level. These computations run at compile time, before the compiler generates an executable of the source code, allowing programmers to transform logic errors into type errors [6]. The exception misuse described above could be avoided by employing logical invariant checks at the type-level, rather than at runtime.

Since these are relatively recent developments, there are few examples of their usage in complex applications. In this project, we show how to utilise type-level programming features in Haskell in order to model the classic board game Chess in Haskell's type system, ruling out invalid moves at the type-level. A Haskell-embedded Domain-Specific Language (DSL), for describing games of chess, will interact with the type-level model. This Embedded DSL (EDSL) will be modelled on Algebraic Notation, a method of writing down the moves associated with a particular match of chess. We

[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://docs.oracle.com/javase/8/docs/api/java/util/Optional.html`

[4] `https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references`

[5] `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`

implement the full, official FIDE ruleset for chess.

2 Background

In fully dependently-typed languages, such as Idris, there is no clear distinction between types and values. Types' definitions depend on values, and vice versa; so types and values are interchangeable in usage. Functions can be written to compute with types, as well as values.

Haskell was not designed to be such a language. In Haskell, values have types, and types have kinds. Types of kind `*`, also known as `Type`, have term-level values; `Int` has kind `*` because it has the inhabitants `0`, `1`, `2`, etc. The `-XDataKinds` extension allows the programmer to define other kinds [3], but these kinds do not have any term-level values.

While GHC compiles a Haskell program, *type erasure* occurs. At runtime, Haskell types have no representation, and so writing runtime computation for type-level constructs is nigh impossible. Much previous academic work has gone into emulating dependent types in Haskell [7], worKing around type erasure through clever use of GHC extensions. Chesskell uses many of these techniques to blur the line between types and values, allowing compile-time checking of the rules of chess—as we explain below.

2.1 Related Work

There are many publicly available chess-related programs written in Haskell[6,7]. Many of these are chess engines, which take in a board state and output the move(s) which are strongest, and so therefore perform 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[8]. The N-queens problem makes use of some chess rules, including the Queen's attack positions[9]; but as the end goal is not to successfully model a game of chess, it is not a full type-level chess implementation.

An important part of any good chess program is its board representation, since all other parts of the program come from this; move generation, move evaluation, and the entire search space are all defined or influenced by the board representation. A

[6] `https://github.com/mlang/chessIO`
[7] `https://github.com/nionita/Barbarossa`
[8] `https://github.com/ExNexu/nqueens-idris`
[9] A Queen can attack in a straight line in any direction.

great deal of work has gone into defining memory- or time-efficient chess boards [8] [9], including combinations of multiple representations to yield greater speed [10]. While there is value to be gleaned from examining these representations, Chesskell serves a different purpose; it does not need to search through the valid set of moves to determine which are the best, and speed is not its focus. Chesskell's board representation must be relatively efficient, but it would be naive to expect similar levels of performance from type-level constraint solving computation as from optimised term-level code.

Despite the apparent lack of work on Chess at the type level specifically, there is work on Haskell-Embedded DSLs in other domains to enforce certain behaviour at compile time. Mezzo [11] is an EDSL for music composition, which checks if the described piece of music conforms to some musical ruleset during compilation of the program. This EDSL is similar to Chesskell in aim, if not in application domain; performing compile-time checks of rulesets that are commonly checked at runtime.

As another example, BioShake [12] is an EDSL for creating performant bioinformatics computational workflows. The correctness of these workflows is checked during compilation, preventing any from being created if their execution would result in certain errors. For bioinformatics workflows especially, this is ideal since many of these workflows are lengthy. BioShake goes further, however; providing tools to allow parallel execution of these workflows. While it is encouraging to see this and other EDSLs [13] focus on (and achieve) high performance, Chesskell will have no such focus. This is primarily because only certain parts of the rule-checking process can be parallelised; much of the move handling and order of rule checks must be done sequentially.

3 Current Progress

Our EDSL allows us to describe games of chess, move-by-move. For example, we express a simple 3-move checkmate by White as follows in Chesskell:

```
game = chess
             pawn _e2 to _e4
             pawn _f7 to _f5
             queen _d1 to _f3
             pawn _g7 to _g5
             queen _f3 to _h5
         end
```

Note that the White team always begins a game of chess, and moves alternate teams; so there is no need to annotate moves with their team.

Each chess move is described with a type family, which takes as input the current state of the board, and outputs the board after the move has been processed. The core movement function, aptly named `Move`, takes in the position to move from, the position

to move to, and the current state of the board, using this information to produce a new board in which the move has been made:

```
data Move :: Position -> Position -> BoardDecorator -> Exp
   BoardDecorator
```

The EDSL, as we explain in more detail below, uses this `Move` function to perform type-level rule checking of the described chess game. While the continuation-passing style (CPS) structure complicates the relevant types, the intuition of the EDSL is to take in the current board state, as well as the positions to move from and to, and output the new board state generated by that move. A simplified non-CPS example is below, to aid understanding:

```
edslMove :: SPosition from -> SPosition to -> Proxy (b ::
   Board) -> Proxy (Eval (Move from to b))
edslMove (x :: SPosition from) (y :: SPosition to) (z :: Proxy
   (b :: Board)) = Proxy @(Eval (Move from to b))
```

3.1 Chess Types Overview

The software has a full representation of a game of chess at the type-level, as we explain below. As the system checks the game move by move, the only required representation is the board at the time of checking; containing all pieces on that board.

The `Board` type is the most important type in Chesskell, representing the chess board at the type-level. It is a *composite type* containing all pieces in play. However, `Board` types are usually accessed through a type named `BoardDecorator`, which contains more information for ease of use, as we detail below.

We describe Chesskell's types from the bottom up in the following sections, since the types here are composite and require understanding of other types.

3.1.1 Team and PieceName

Both `Team` and `PieceName` are simple algebraic data types, with all constructors defined in code. The `Team` type enumerates all teams a piece can belong to; `Black` and `White`. The `PieceName` type enumerates all possible names of pieces; Pawn, 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 of kind `Team`[10].

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

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

3.1.2 Position

The `Position` type holds the positions of pieces on the chess board. It 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 enumerating all columns that a piece can reside within. The row type is a type-level implementation of Peano natural numbers, named `Nat`. Early versions of Chesskell had a custom implementation, but the current version simply uses definitions provided in `Data.Type.Nat`.

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

Note that the `Position` kind has a potentially infinite number of valid types, but only 64 of these types are valid chess positions. As such, there is a type family, `IsValidPosition`, which outputs `True` if the given position is a valid chess position, and `False` otherwise.

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

3.1.3 The Pieces

Each piece, represented by the `Piece` type, contains information relevant for rule checking: that piece’s team, name, and an information type. The information type, named `PieceInfo`, contains a `Nat` and a `Position`, to represent the number of moves that piece has taken, and its current position on the board respectively. Recording the number of moves the piece has taken is important for several rules in chess, including castling and *En Passant*, and so is included in the `PieceInfo` type.

The `PieceInfo` type was created separate from the plain `Piece` type so that if any further information was needed, it could be added without breaking existing `Piece` pattern-match definitions, though no such additional information was required.

```
data PieceInfo where
    Info :: Nat -> Position -> PieceInfo

data Piece where
    MkPiece :: Team -> PieceName -> PieceInfo -> Piece
```

3.1.4 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 (an 8x8 grid) without a length check each move? This would take at least 56 additions, since list length is computed recursively; as well as 7 more 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 specify the length of a vector, resulting in a type error if the vector is any other length. As with most things in Haskell, we use recursive definitions; 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
    (:->)  :: 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 `Maybe Piece`-s. We use `Maybe Piece` instead of just `Piece` because a board square does not necessarily 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 similar to the decorator design pattern [14], 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.

Previous versions of the program, to find the King positions, would pass repeatedly over the `Board`. Having their positions available in the decorator saves Chesskell from making these passes. 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 decorator.

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

3.2 Checking Chess Rules

We perform all rule-checking with type families that take in the `BoardDecorator`, among other arguments, and output either a correct result, or a type error. Rule-checking occurs during movement, since the chess game effectively hangs while players do not move.

3.2.1 Type families in Haskell

Although Haskell allows type-level computation via type families, these type families can't be partially applied like normal Haskell functions can. However, Li-yao Xia's work on First Class Families[11] defines a type-level interpreter, allowing similar behaviour to partial application.

First Class Families are achieved through the definition of an open type family, `Eval`, and an expression type, `Exp`. In Haskell, type constructors can be partially applied at the type level, where type families cannot. Type-level functions are defined such that their return value is wrapped in an `Exp`. Consider a type-level implementation of addition in First Class Family form, and in type family form:

```
data Plus :: Nat -> Nat -> Exp Nat
type instance Eval (Plus x y) = Plus' x y

type family Plus' (n :: Nat) (m :: Nat) :: Nat where
    Plus' Z     m = m
    Plus' (S n) m = S (Plus' n m)
```

`Plus 4` would be a valid type, with kind `Nat -> Exp Nat`; whereas `Plus' 4` would result in a type error. With a type-level `Map` definition, a first class family of the appropriate type can be applied to every element in a sequence. Consider the below `Vector` instance for `Map`:

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

```
data Map :: (a -> Exp b) -> f a -> Exp (f b)
type instance Eval (Map f VEnd)        = VEnd
type instance Eval (Map f (x :-> xs)) = Eval (f x) :-> Eval
   (Map f xs)
```

If a `Vector` is mappable, then the `Board` is mappable. We can perform computation on the pieces therein without writing a new function each time. Consider the below code snippet, executed in GHC's interpreter, which allows you to fully evaluate type-level computation with the `:kind!` command:

```
$> :kind! Eval (Map (Plus (S Z)) (Z :-> (S Z) :-> VEnd))
Eval (Map (Plus (S Z)) (Z :-> (S Z) :-> VEnd)) :: Vector (S (S
   Z)) Nat
    = S Z :-> (S (S Z) :-> VEnd)
```

As expected, the given First Class Family `Plus (S Z)` has been applied to every element in the `Vector`. Due to the ability to map over the `Board`, 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.

There is one major caveat, though. When pattern-matching with First Class Families, catch-all matches fail to work. That is, the below would fail to compile, since either definition could be valid for `Plus Z 5`.

```
data Plus :: Nat -> Nat -> Exp Nat
type instance Eval (Plus Z n) = n
type instance Eval (Plus m n) = -- ...
```

However, First Class Family-style definitions can be combined with regular type families to enable pattern matching, as with `Plus` and `Plus'` above.

3.2.2 Move Lists

Each piece, depending on its team and name, can move 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 that piece as a `Piece` type, and a `BoardDecorator` representing the current state of the board.

```
data PieceMoveList :: Piece -> BoardDecorator -> Exp [Position]
```

Consider a `PieceMoveList` instance for Bishops:

```
type instance Eval (PieceMoveList (MkPiece team Bishop info)
   boardDec) = Eval (AllReachableDiag team boardDec (Eval
   (GetPosition info)))
```

Bishops can move diagonally in a straight line by any number of spaces. The type family `AllReachableDiag` is used to get a list of all diagonally "reachable" positions. It takes in the `Position` of the relevant piece, that piece's `Team`, and the current state of the board as a `BoardDecorator`. It outputs all diagonal positions that piece can move to.

Reachability for a given direction is defined here as all the empty spaces in that direction, stopping at either the first occupied space or the edge of the board. That space is included or excluded depending on whether that space is occupied by a piece of the opposite team, since an attacking piece could move to that space and take the piece there.

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.

3.2.3 Moving the Pieces

`Move`, as we introduce above, performs a series of checks on the board state, computes a move from an origin `Position` to a destination `Position` with a given `BoardDecorator`, and then performs further checks. The actual movement code is omitted here for brevity; but all moves are present and complete. The checks themselves are implemented as functions that take in a `BoardDecorator`, and output either a `BoardDecorator` if the check passes, or a type error. All of these functions have been given the suffix `-Check`.

```
data Move :: Position -> Position -> BoardDecorator -> Exp
   BoardDecorator
type instance Eval (Move fromPos toPos boardDec) = Eval ((
    CheckNoCheck (GetMovingTeam boardDec) .
    MoveNoChecks fromPos toPos .
        NotTakingKingCheck toPos .
        CanMoveCheck fromPos toPos .
        NotTakingOwnTeamCheck toPos .
        NotSamePosCheck fromPos toPos .
        NotLastToMoveCheck fromPos .
        TeamCheck fromPos) boardDec)
```

3.3 EDSL

3.3.1 Proxies and Singletons

Type-level information is not naturally available to Haskell's term-level facilities, due to type erasure. As such, there are various methods to simulate computation with

types at runtime.

Proxy types are parameterised with kind-polymorphic type variables [3]. A `Proxy` value can have type `Proxy Int` or `Proxy 'Black`, for example. Proxies are used to make runtime board information available at the type level in Chesskell.

Singleton types mimic dependent types [7]; each type has a single, unambiguous value, and vice versa. We use this value to pass around type information at the term-level. Chesskell uses singletons to perform computation with positions and piece names. For example, the value `_a4` is has type `SPosition (At A (S (S (S (S Z)))))`.

3.3.2 EDSL Implementation

A Continuation Passing Style [15] scheme forms the foundation for the EDSL, with inspiration taken from Dmitrij Szamozvancev's Flat Builders pattern [16]. The core idea is value transformation through a series of continuation function applications, until the final continuation function returns a value.

The type `Spec t` is the type of functions which take in a continuation to operate on a value of type `t`. For instance, a function with type `Int -> Spec Int` would take in an integer, and then a continuation to operate on that integer.

```
type Spec t = forall m. (t -> m) -> m
```

A function with type `Int -> Spec Int` can be represented with `Conv Int Int`—the `Conv s t` type represents functions which convert a value of type `s` to a value of type `Spec t`.

```
type Conv s t = s -> Spec t
```

Finally, the `Term t r` type ends the continuation stream by taking no continuations, and simply taking in a value of type `t` and returning a value of type `r`. If `t` and `r` are equal, then an example definition would be `id`.

```
type Term t r = t -> r
```

The above continuation types are combined with type-level rule checking, to create a Chess EDSL that operates through passing continuations. Type-level computations within the EDSL are achieved through *type applications* [17], which allow the programmer to specify the exact type of a kind-polymorphic type variable. Using a combination of singletons, proxies, kind signatures, and type applications, the value-level Haskell code for the EDSL can have specific type variables and involve type family application. Essentially, the term-level EDSL can involve type-level rule checking.

The chess game starts with a `Proxy` value, with its type containing a `BoardDecorator` type. Continuations are applied, transforming that value, until the chess game ends

or a rule is broken. Chess games begin with the board in a set configuration; and so a type `StartDec` of kind `BoardDecorator` was defined to contain all of this information.

```
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy @StartDec)
```

The core continuations are named after the pieces, such as `pawn` and `king`. Each of them takes in an `SPosition`, a singleton version of `Position`. We define a new datatype, `MoveArgs`, in order to simplify the process of passing information between the continuations; `MoveArgs` can be partially applied, whereas a regular type family cannot. The `pawn` continuation is used below as an example; however, all of the piece continuations are similar, and only differ in the `PieceName` type passed to the continuation via `MoveArgs`.

```
data MoveArgs where
    MA :: BoardDecorator -> Position -> PieceName -> Position
   -> MoveArgs

pawn :: Proxy (b :: BoardDecorator) -> SPosition fromPos ->
   Spec (Proxy (MA b fromPos 'Pawn))
pawn (dec :: Proxy b) (from :: SPosition fromPos) cont = cont
   (Proxy @(MA b fromPos Pawn))
```

The next continuation, `to`, takes in another `SPosition` as well as the `MoveArgs`, performs the move computation, puts the resulting board decorator into a `Proxy` type, and passes that `Proxy` into the continuation given.

```
to :: Proxy (MA (b :: BoardDecorator) (fromPos :: Position) (n
   :: PieceName)) -> SPosition toPos -> Spec (Proxy (Eval
   (MoveWithStateCheck n fromPos toPos b)))
to (args :: Proxy (MA (b :: BoardDecorator) (fromPos ::
   Position) (n :: PieceName))) (to' :: SPosition toPos) cont =
   cont (Proxy @(Eval (MoveWithStateCheck n fromPos toPos b)))
```

The final relevant definition is of `end`, which ends the chess game, as well as the continuation stream.

```
end :: Term (Proxy (b :: BoardDecorator)) (Proxy (b ::
   BoardDecorator))
end = id
```

Using the above continuations, we can lay out a chess game, move by move. Consider the game expressed in the EDSL which we describe in section 3. It compiles successfully; but should Black attempt to move after checkmate, an error will be logged, since no moves by Black will be valid:

```
-- Below results in the following type error:
```

```
    -- * The Black King is in check after a Black move. This is
   not allowed.
    -- * When checking the inferred type
    --     game :: Data.Proxy.Proxy (TypeError ...)
game = chess
           pawn _e2 to _e4
           pawn _f7 to _f5
           queen _d1 to _f3
           pawn _g7 to _g5
           queen _f3 to _h5
           pawn _g5 to _g4
       end
```

Or, should White attempt an impossible move in the middle of the game, such as moving a Queen through another piece, a different type error will occur:

```
-- Below results in the following type error:
    -- * There is no valid move from D1 to D3.
    -- The Queen at D1 can move to: E2, F3, G4, H5
    -- * When checking the inferred type
    -- game :: Data.Proxy.Proxy (...)
game = chess
           pawn _e2 to _e4
           pawn _f7 to _f5
           queen _d1 to _d3
           pawn _g7 to _g5
           queen _f3 to _h5
       end
```

3.3.3 EDSL Board Creation

There are additional statements to help create an initial chess board for easy testing. A short example is below, in which White Rooks are placed at a8 and h1, and a Black Pawn is placed at h7 before the game begins.

```
example
  = create
        put _Wh _R at _a8
        put _Wh _R at _h1
        put _Bl _P at _h7
    startMoves
        rook _a8 to _a2
        pawn _h7 to _h6
        rook _h1 to _h2
    end
```

3.3.4 Pawn Promotion

The EDSL function `becomes` handled Pawn promotion in earlier versions of Chesskell, transforming a Pawn into another piece after it has moved to the opposite side of the board. Since the whole chess game is expressed in continuation-passing style, `becomes` simply edits a Pawn's `PieceName` in place.

```
game = chess
           -- ...
           pawn _a7 to _a8 becomes _Queen
           -- ...
       end
```

We can combine this with the board creation functionality to create a simple game which compiles:

```
game = create
           put _Wh _P at _a7
       startMoves
           pawn _a7 to _a8 becomes _Queen
       end
```

However, the below code will also compile, when it should not, since according to the FIDE 2018 Laws of Chess[12], promotion is not optional, and must occur.

```
game = create
            put _Wh _P at _a7
        startMoves
            pawn _a7 to _a8
        end
```

We require additional logic to enforce promotion, in two ways. Firstly, promotion must occur when a Pawn reaches the opposite end of the board. Secondly, if a Pawn reaches the opposite end of the board but is not promoted, then Chesskell should throw a type error.

`ShouldHavePromotedCheck` is a new check for enforcing promotion during movement. It checks if a promotion should have occurred[13] and outputs a type error if so, and the inputted `BoardDecorator` otherwise. Promotion within the EDSL occurs through a new function, `promoteTo`, which uses a type family `PromotePawnMove` to promote a Pawn to a given `PieceName` after movement. The following code snippet compiles, since the White Pawn is promoted to a Queen:

```
game =  create
            put _Wh _P at _h7
```

[12] `https://handbook.fide.com/chapter/E012018`

[13] i.e. if a Pawn reached the opposite end of the board.

```
        startMoves
            pawn _h7 promoteTo _Queen _h8
        end
```

However, the below does not compile, since no promotion occurs even though the White Pawn reaches the opposite end of the board:

```
-- Below results in the following type error:
--      * Promotion should have occurred at: H8. Pawns must be
   promoted when they reach the opposite end of the board.
--      * When checking the inferred type
--          didntPromoteWhite :: Data.Proxy.Proxy (TypeError ...)
game =  create
            put _Wh _P at _h7
        startMoves
            pawn _h7 to _h8
        end
```

3.3.5 EDSL Issues

GHC had issues compiling early versions of the EDSL, due to the initial definition of `chess`; the programmer did not manually create `StartDec`, but pieced it together through a series of expensive type family applications:

```
type StartDec = MakeDecorator (ExpensiveOperation (...))
```

This early version of `StartDec` was used to set up the chess game, with attempts at two definitions of `chess`; one using a type application, and the other using a type signature.

```
-- Early definition of chess with type application
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy @StartDec)

-- Version of chess with type signature
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy :: Proxy StartDec)
```

The version of `chess` that used a type application would compile without issues, but would cause lengthy pauses at runtime when used. Conversely, when compiling the type signature version, GHC would consume high amounts of memory, and would either crash or hang for multiple minutes. This behaviour is unusual, so we filed a GHC bug report[14]. As a work-around until the bug is fixed, `StartDec` is manually

[14] `https://gitlab.haskell.org/ghc/ghc/-/issues/18902`

written out in full in the source Haskell file, allowing compilation and usage of either definition with no issues.

Additionally, adding the move number to the type error[15] causes a spike in compile time; not as excessive as above, as GHC does not crash, but still doubling or tripling compile time. The move number will only be re-added to the error if this problem can be fixed or worked around, since it would greatly slow the testing of a large number of games.

3.4 Testing

The test suite is a mixture of HSpec behaviour-driven tests [18] and module declarations, where the HSpec tests cover many of the declared type families, such as `Move` and `PieceMoveList`.

The original plan was to have all tests be HSpec tests; many of these HSpec tests make use of the type-level testing facilities laid out in other works [16]. However, running the test suite on EDSL statements, as well as some of the expensive type families, such as those involving castling, causes excessive memory usage, exceeding 27 GB in some cases.

Work on a solution is ongoing, but for now, we express EDSL tests as high-level declarations in a Haskell module. The tests are run by seeing whether these declarations compile or not, and if not, what error they throw. This will hopefully only be temporary, but it should suffice as a rudimentary, if slow, method of integration testing.

4 Next Steps

As the project's main feature set is complete, the remaining time will be spent testing, evaluating, and improving the project. Full-length chess games will be expressed in the EDSL, to determine if the program compiles correct chess games. Then, we will introduce subtle errors to those games, to test that incorrect games fail to compile. Of course, the code itself is only part of the project; work on the dissertation should begin early next term/year.

4.1 EDSL

The EDSL is feature-complete, as far as chess rules go. However, a few additional helper functions or syntax changes may be helpful during development or to clarify their purpose, and so the EDSL is subject to change.

[15] e.g. `Error at move 1: Pawn cannot move to...`

4.2 Performance

The current memory usage and compile time of the EDSL is acceptable, but only short (<10 moves) chess games have been described and tested thus far. Memory usage and compile time are likely to rise both with the length of the chess game, and the number of games tested. Memory usage optimisations may become necessary as testing continues.

4.3 Project Extensions

As the project is nearly complete, it may be worth adding several extensions to further explore Chess at the type level. Some potential extensions are described below.

4.3.1 Session Types

Session types are formalisms of communication, between two or more parties, at the type level [19]. Interestingly, chess could theoretically be modelled in terms of communication between two parties, Black and White; they can only send one message at a time to each other, and their communications cannot break the rules of chess. A potential extension would be to conduct more research into this topic, determining how viable a session-typed Chesskell would be, and possibly implementing it, time constraints allowing.

4.3.2 Haskell Symposium Paper

Another potential extension is to write a paper on Chesskell for the Haskell Symposium 2021, either for its early track or its regular track. The Haskell Symposium accepts papers on a variety of topics, including language design, experiences, tools, and so on. This would be a good opportunity to gauge academic interest in Chesskell, and contribute to the wider academia surrounding type-level programming.

4.4 Revised Timetable/Plan

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

These two weeks will be spent performing integration/system testing. A curated data set of chess games will be expressed in the EDSL and put through the GHC compiler. Some of these games will be valid, and some will be invalid. Only valid games should compile, and invalid games should fail to compile with an appropriate type error.

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

This is the first section of allocated empty space. If the test suite is posing any issues with regards to compile time and memory usage then this additional month should help with solving them. But should the testing 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 Chess at the type level, as we discuss above.

4.4.3 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, while gathering any relevant graphs, figures, and citations.

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.

4.4.4 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. We will finish an initial draft by the 7th of March.

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

4.4.5 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 occurs on the 19th March; so this empty space will ideally be spent completing the presentation and rehearsing it.

4.4.6 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.

5 Risks

The biggest risk going into the project was that it was unachievable; however, that risk has not materialised—the implementation of type-level chess rules is complete. The largest project-related risks left are the memory usage and compile time issues.

There are, of course, other risks not directly associated with the project; in 2020, there is a large-scale pandemic going on, which could pose considerable personal risk. However, by following Government and University guidelines, the chances of contracting COVID-19 will be kept to a minimum, and sudden illness should not interrupt the project.

6 References

[1] B. Cabral and P. Marques, "Exception handling: A field study in Java and .NET," in *European Conference on Object-Oriented Programming*, pp. 151–175, Springer, 2007.

[2] M. Asaduzzaman, M. Ahasanuzzaman, C. K. Roy, and K. A. Schneider, "How developers use exception handling in Java?," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pp. 516–519, IEEE, 2016.

[3] 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.

[4] T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty, "Towards open type functions for Haskell," *Implementation and Application of Functional Languages*, no. 12, pp. 233–251, 2007.

[5] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich, "Closed type families with overlapping equations," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 671–683, 2014.

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

[7] R. A. Eisenberg and S. Weirich, "Dependently typed programming with singletons," *ACM SIGPLAN Notices*, vol. 47, no. 12, pp. 117–130, 2012.

[8] E. A. Heinz, "How DarkThought plays chess," *ICGA Journal*, vol. 20, no. 3, pp. 166–176, 1997.

[9] T. Warnock and B. Wendroff, "Search tables in computer chess," *ICGA Journal*, vol. 11, no. 1, pp. 10–13, 1988.

[10] S. K. Bimonugroho and N. U. Maulidevi, "A hybrid approach to representing chessboard using bitboard and compact chessboard representation," in *IOP Con-*

ference Series: Materials Science and Engineering, vol. 803, p. 012018, IOP Publishing, 2020.

[11] D. Szamozvancev and M. B. Gale, "Well-typed music does not sound wrong (experience report)," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, pp. 99–104, 2017.

[12] J. Bedő, "BioShake: a Haskell EDSL for bioinformatics workflows," *PeerJ*, vol. 7, p. e7223, 2019.

[13] A. Ekblad, "High-performance client-side web applications through Haskell EDSLs," in *Proceedings of the 9th International Symposium on Haskell*, pp. 62–73, 2016.

[14] 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.

[15] 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.

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

[17] R. A. Eisenberg, S. Weirich, and H. G. Ahmed, "Visible type application," in *European Symposium on Programming*, pp. 229–254, Springer, 2016.

[18] S. Hengel, "Behavior-driven development in Haskell,"

[19] M. Dezani-Ciancaglini and U. De'Liguoro, "Sessions and session types: An overview," in *International Workshop on Web Services and Formal Methods*, pp. 1–28, Springer, 2009.