Chesskell: A Two-Player Game at the Type Level

Toby Bailey
Toby.Bailey@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Michael B. Gale
M.Gale@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Abstract

Extensions to Haskell's type system, as implemented in GHC, have given developers more tools to express the domain-specific rules and invariants of their programs in types. For these extensions to see mainstream adoption, their use in complex applications has to be practical. We present Chesskell, an EDSL for describing Chess games where a type-level model of the full FIDE ruleset prevents us from expressing games with invalid moves. Our work highlights current limitations when using GHC to express such complex rules due to the resulting memory usage and compile times, which we report on. We further present some approaches for working around those limitations.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; **Functional languages**; **Constraints**.

Keywords: Haskell, Chess, First Class Families

ACM Reference Format:
Toby Bailey and Michael B. Gale. 2021. Chesskell: A Two-Player Game at the Type Level. In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3471874.3472987

1 Introduction

By encoding invariants in types, programmers can leverage their compiler to check those invariants for them, leading to greater safety guarantees. To facilitate this, the Glasgow Haskell Compiler (GHC) implements extensions to Haskell's type system, such as allowing programmers to define functions on types [4, 11]. For developers to adopt these techniques to express sophisticated invariants about their programs, such extensions should not negatively impact development experience, particularly compile time. To test how GHC copes with an implementation of a complex set of invariants, we implement the rules of Chess in Haskell as a typed Chess EDSL: *Chesskell*[1].

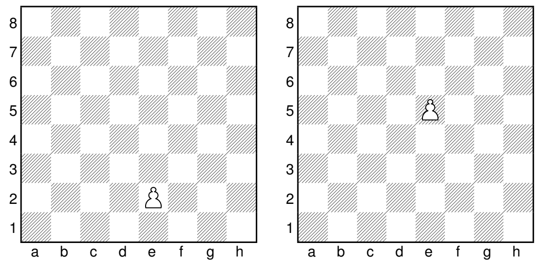

Figure 1. A White Pawn moves from e2 to e5: an invalid move.

Chess is a popular game with a well-defined ruleset [9] as specified by FIDE – the International Chess Federation. Chess games are played by two players who take turns moving individual pieces belonging to their team around a board, until a winning condition is met. Each piece can be moved according to its rules and the current state of the board. Indeed, due to the complex rules governing the game it is easy for human players to make mistakes that inadvertently lead to an invalid move.

Consider a Pawn (one of the pieces) which moves from square e2 to square e5, as shown in Figure 1. This move is not allowed in Chess, but a human could accidentally miss such an invalid move. In Chesskell, we express this move as `pawn e2 to e5` and, if we try to compile it, we receive the following type error from GHC:

```
* There is no valid move from E2 to E5.
  The Pawn at E2 can move to: E3, E4.
```

[1]All of our code can be found at https://github.com/Proxymoron461/chesskell.

In fact, if any FIDE 2018 Chess rule is violated, the described game will simply not compile. The type error produced by Chesskell also explains the cause of the problem to the user and suggests corrective action: the Pawn moved to a square that it cannot reach. To rectify the problem, we can move the Pawn to e3 or e4 instead, one of the squares mentioned by the error message. The resulting program then compiles successfully.

Our work utilises Li-yao Xia's *First Class Families* technique extensively, for which we provide an overview in Section 2 before building on it in our contributions:

- A full type-level model of Chess according to the FIDE 2018 Laws of Chess, which allows us to detect games which contain invalid moves (Section 3);
- An Embedded Domain Specific Language for describing Chess games in Haskell (Section 4), resembling typical Chess notation, which is typed using our type-level model of the FIDE 2018 Laws of Chess;
- We report on our experience with GHC in developing Chesskell with respect to compile time and memory usage, including a discussion of techniques which improved on this (Section 5).

2 First Class Families

In functional programming languages, the benefits of having functions as first-class values are well known: it allows us to abstract over common patterns in our programs with higher-order functions and to partially apply functions to some, but not all, of their arguments. This is useful because we can, for example, define a higher-order function `map` which applies some other function to all the elements of a list.

However, Haskell's type system does not currently permit us to do the same with type families: neither closed nor open type families [4, 11] can be partially applied – they must instead always be fully applied to all of their arguments. This means that it is, for example, not possible to make use of a type family similar to `map`, which would apply some other given type family to all the elements of a type-level list in Haskell. Indeed, many abstractions such as functors, applicative functors, and monads which we take for granted at the value-level are unavailable to us in Haskell's type system.

Although a change to Haskell's type system which removes this restriction has been described [10] and provisionally accepted for inclusion in GHC, it is not available at at the time of writing. Meanwhile, a workaround for this limitation of Haskell's type system has been developed by Li-yao Xia, who refers to the technique as *First Class Families*[2]. The key idea of this approach is the observation that, while type families cannot be partially applied, type constructors can be. Therefore, by defining type constructors along with a type-level interpreter for them, we can simulate partial type application. The interpreter is an open type family named **`Eval`**:

```
type Exp a = a -> *
type family Eval (e :: Exp a) :: a
```

The definition of **`Exp`** is a shorthand for the kind of type constructors from some kind a to kind `*`. As an example of a type-level function implemented in this style, let us first consider a definition of an ordinary, closed type family for logical OR, named `Or`:

```
type family Or (x :: Bool)
               (y :: Bool) :: Bool where
    Or 'True x = 'True
    Or x 'True = 'True
    Or x 'False = 'False
```

This type family can be defined in the First Class Family style by defining a new data type to hold all the arguments, paired with an **`Eval`** instance to state how it should be evaluated:

```
data Or :: Bool -> Bool -> Exp Bool
type instance Eval (Or True True) = True
type instance Eval (Or True False) = True
type instance Eval (Or False True) = True
type instance Eval (Or False False) = False
```

We can also define a type-level equivalent of `map` using this technique:

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

Combined, we can now partially apply *e.g.* `Or`, give it as an argument to `Map` along with a list of type-level booleans, and then use **`Eval`** to compute a result: for example, the type `Eval (Map (Or 'False) '[ 'False, 'True ])` evaluates to `'[ 'False, 'True ]`.

We note that, since we use an open type family for the interpreter, there cannot be any overlapping cases since the instances are not ordered. For instance, GHC fails to compile the following definition because `x` could be **`True`** or **`False`** and therefore, if `x` were **`True`**, either the first or second instance shown below could apply:

```
-- Error: Conflicting family instance declarations
data Or :: Bool -> Bool -> Exp Bool
type instance Eval (Or True _) = True
```

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

```
type instance Eval (Or x True) = True
type instance Eval (Or x False) = False
```

When using a closed type family (or a value-level function), the patterns of the equations are matched against in the order provided, so if there is overlap, the first case that matches is used. In Chesskell, we leverage this capability and combine First Class Families with closed type families, where the latter provide the actual implementation of instances of the `Eval` type family. This allows us to have both partial application and the benefits of ordered definitions:

```
data Or :: Bool -> Bool -> Exp Bool
type instance Eval (Or x y) = Or' x y

type family Or' (x :: Bool) (y :: Bool) :: Bool
    where
    Or' 'True _ = True
    Or' x 'True = True
    Or' x 'False = False
```

Here we have the benefit of having a more concise definition for `Or'` which provides the implementation for an instance of `Eval` for `Or`, along with the ability to partially apply `Or`.

As part of the development of Chesskell, we have re-implemented many functions from Haskell's standard library using the First Class Family technique. If a First Class Family in the following sections has the same name as a function from Haskell standard library, the reader may assume that the First Class Family has similar behaviour as the value-level equivalent, unless explicitly stated otherwise. It is worth noting that such a re-implementation of standard library functions at the type-level would not be necessary in other languages which do not distinguish between types and values in the same way Haskell does. Indeed, Haskell has different syntax for value-level functions and type-level functions (type families). Eisenberg and Stolarek propose a workaround for this using Template Haskell which allows both to be derived from one definition [3].

3 Chess Model

We describe each move in Chess with a corresponding type family, which takes the current state of the board as input and outputs the board after the move has been processed. All of these movement rules are combined into a single First Class Family, named `Move`, which takes in the position to move from, the position to move to, and the current state of the board. It then uses this information to return a new board state in which the move has been made, determining what sort of move is being made in the process. Additionally, it updates relevant piece information for the pieces that have moved, which we detail later on. `Move` has the following kind:

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

Fundamentally, Chesskell works by repeatedly applying this First Class Family to every move that is made in a given game where types of kind `BoardDecorator` represent the game state which is threaded from move to move. The `BoardDecorator` kind is promoted from the following data type definition [15]:

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

Types of this kind store the following information:

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

3.1 Chess Types

Chesskell involves a number of types and kinds to represent various components of Chess. In the following sections, we describe them from the bottom up.

3.1.1 Team and PieceName. To enumerate the two teams a piece can belong to, we define a type `Team`. To enumerate all pieces in the game, such as pawns, rooks, and so on, we define `PieceName`:

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

3.1.2 Position. The `Position` type represents 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: *i.e.* "a1" refers to the bottom-left of the board, and "h8" to the top-right. While the `Column` type is another simple algebraic data type enumerating all columns, rows are just represented as a standard natural number:

```
data Column = A | B | C | D | E | F | G | H

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

Note that the Position kind has a potentially infinite number of valid types, but only 64 of these types are valid chess positions. We check for valid positions with a type family named IsValidPosition which results in **True** if a given position is a valid chess position and **False** otherwise.

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 (Section 3.3.1) and *en passant* (Section 3.3.2), and so is included in the PieceInfo type:

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

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

3.1.4 The Board. In Chess, the board is an an 8x8 grid of 64 squares. We encode this using length-indexed vectors to enforce a fixed size. Our Vec type is defined in the usual way as:

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

The board is therefore a vector of size eight containing vectors of size eight.Furthermore, each square on the board may either contain a piece or be empty. We represent this by wrapping the Piece kind in **Maybe**. The full definition of a Board kind is therefore:

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

This is then stored in types of the BoardDecorator kind we have already introduced.

3.2 Chess Rules

In Chesskell, the rules of Chess are expressed as First Class Families that either return a BoardDecorator or a type error. Each such First Class Family has the suffix Check, such as NotTakingKingCheck. These checks are broadly split into two categories: checks which are performed before a move is made and checks which are performed after a move is made. Each check uses custom error messages to communicate what rule is violated and what valid moves may exist.

As an example, the definition of NotSamePosCheck is given below, which checks that a move is between two distinct positions and otherwise results in a type error. Custom type errors are implemented using the ErrorMessage kind[3] from GHC's standard library. The TE' and ID types are First Class Families which return the results as **Exp** BoardDecorator types:

```
data NotSamePosCheck :: Position -> Position
                     -> BoardDecorator
                     -> Exp BoardDecorator
type instance Eval (NotSamePosCheck fromPos toPos
    boardDec)
    = If' (Eval (fromPos :==: toPos))
        (TE' (TL.Text ("Moves␣from␣a␣position␣to␣
    that␣same␣position␣are␣not␣allowed.")))
        (ID boardDec)
```

The movement of different pieces usually involves the combination of multiple common checks. We combine multiple rule checks using a First Class Family version of the function composition operator, (.):

```
ExampleCheck2 . ExampleCheck1 . NotSamePosCheck
```

3.2.1 Movement Rules. Each piece's movement is subject to a specific set of rules for that piece. For instance, a King can move a single space in any direction, but never into the attack path of another piece. To formalise this, we define the PieceMoveList First Class Family, which given a piece of kind Piece and a BoardDecorator representing the current state of the board as arguments, returns a list of positions that a piece can move to.

```
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.

[3] https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-TypeLits.html#t:ErrorMessage

We define reachability for a given direction as all the empty spaces in that direction, stopping at either the first occupied space or the edge of the board. That occupied space is included or excluded depending on whether it is occupied by a piece of the opposite team, since an attacking piece could move to that space and take the piece there. If it is occupied by a piece of the same team, then it is not included.

We ensure correctness of movement with a pre-move rule-check named `CanMoveCheck`:

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

This checks if there is a piece at the first position that can move to the second position. Additionally, in order to produce more specific error messages, there exist a few additional checks, such as `TeamCheck`, which ensures that the same team does not move twice in a row.

3.2.2 Attack/Capture Rules. The list of spaces that a piece can attack and the list of spaces that a piece can move to are not the same. For example, pieces can attack the King of the opposite team, but cannot directly move to that King's position and capture it. Therefore, `PieceMoveList` cannot be used to determine which squares a piece can attack. We define another type family, `PieceAttackList`, which gives the list of all squares that a piece can attack.

Checking for Check. One of the most important rules in Chess, that of placing the opposite team's King in check, cannot be expressed solely through move and attack lists. Any movement can place either King in check and it is not always the case that a movement by a piece places the opposite King in check. A move may be ruled as invalid because it places that piece's King into check. For instance, if a Black Rook stands between a White Queen and a Black King, the Rook cannot move out of the Queen's attack path, since such a move would put the Black King into check.

However, the only time that check is relevant is after each move. A move by a piece is invalid if it places that piece's King in check, or if it leaves that piece's King in check. We can express this rule as a post-move check, implemented as a First Class Family `CheckNoCheck`.

A naïve approach to detect whether a King is in check is to compute and combine all attack lists for all pieces in order to check if the King's position is in that list. However, we use a more efficient approach in Chesskell where we emulate other pieces' movement from the King's position. It is worth noting that, for all pieces except Kings and Pawns, if they can move from a to b, then they can also move from b to a. As an illustration, if a Queen at the King's position (of the same team as the King) would be able to reach a Queen of the opposite team, then the King would be in check.

We leverage this behaviour to detect when a King is in check. Several "rays" are sent out from the King's position in horizontal, vertical, and diagonal directions (8 in total). These rays detect Queens of the opposite team, as well as Bishops (for diagonal rays) and Rooks (for horizontal and vertical rays). Attacking Pawns are also checked here, for the immediate diagonal positions either above the King (if the King is White) or below the King. If an attacking piece is reached, the ray function returns true; otherwise, it returns false.Additionally, Knight movement rules are applied to check if there are any Knights reachable from the King's position and, if so, then the King is in check.

There is one last piece type not handled by the above method: the other King. This is deliberate as it would be illegal for a King to move within attacking distance of the opposite King, since then the moving King would be in check.

A code snippet for determining if the King is in check, which checks if any of the above conditions are true, is shown below. Each of the `-Ray` functions returns true if a piece could place a King in check from that direction, and the `IsKnightAttacking` function returns true if any Knights of the given team are reachable from the given position:

```
data IsKingInCheck :: Position
                   -> Team
                   -> BoardDecorator
                   -> Exp Bool
type instance Eval (IsKingInCheck kingPos team
    boardDec)
    = Eval (Any '[
    SendLeftRay kingPos team boardDec,
    SendRightRay kingPos team boardDec,
    -- ...
    -- Send rays above, below, and in all 4
    -- diagonal directions
    -- ...
    IsKnightAttacking kingPos team boardDec ])
```

3.3 Special Rules

There are a few Chess rules that are different from all other Chess rules. Therefore, implementing these rules requires different approaches from the implementations of other rules and we describe the interesting aspects of them in detail below.

3.3.1 Castling. Most Chess rules move a single piece and can capture another piece to remove it from play. However, the *Castling* move involves the movement of two pieces: the King and one of their Rooks. Castling can only occur if neither the King nor the Rooks have moved, as long as none of the positions the King would move through are under check, and there are no other pieces between the King and the Rook. It is one of the most complex rules of Chess, and requires many checks before it can be permitted.

Castling illustrates why it is useful to have each piece's move count in the `PieceInfo` type: it enables us to quickly determine if a King or either of the Rooks have moved. Simply checking if the King or Rooks are in their starting positions is not enough, since they could have just moved back to those positions.

3.3.2 Pawn Movement and En Passant. Pawns have the most complex movement rules out of any piece: their attack patterns are different from their movement patterns. Pawns can move one vertical space forwards, but on their first move can move two spaces instead of one. For a White Pawn, "forwards" means towards a row of higher number and, for a Black Pawn, it means towards a row of lower number. However, Pawns cannot capture a piece this way – they can only capture one diagonal space in front of themselves.

This means that a Pawn can make a diagonal move, but only if there is a capturable piece there. For instance, a Black Pawn could move downwards diagonally by a single space to capture a White Bishop, but could not move to that square if it were empty. Additionally, Pawns have one more special capture rule: *en passant*. Implementing *en passant* captures was the driving factor that motivated the `BoardDecorator` type, since the last position moved to was required as part of the process.

4 DSLs for Describing Chess Games

Chesskell is comprised of multiple EDSLs for describing Chess games and Chess boards. We describe games move-by-move and similarly to ordinary Chess notation. For example, in the Cheskell shorthand notation, we can express a simple 3-move checkmate by White as follows:

```
game = chess
    p e4 p f5
    q f3 p g5
    q h5
end
```

Note that the spacing is included purely for readability. The above game could just as easily be written as:

```
game = chess p e4 p f5 q f3 p g5 q h5 end
```

The functions that make up the EDSL are typed using the `Move` first class family to perform type-level rule checking of the chess games that we describe. A Continuation Passing Style (CPS) [13] scheme forms the foundation for the EDSL, inspired by the "Flat Builders" pattern [14]. While a CPS structure complicates the types involved, intuitively we are just describing a sequence of moves, each of which takes in an initial board state as well as the positions to move from and to, and outputs the resulting board state generated by that move. To better illustrate this, a simplified, non-CPS example of a general move function is given below:

```
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))
```

The `from` and `to` type variables represent the positions to move from and to, respectively. Type variable `b` represents the initial state of the board and the resulting board is computed by **`Eval`** `(Move from to b)`.

4.1 Longhand EDSL

The Chesskell longhand EDSL provides a notation that is easier to read than the shorthand EDSL. For example, a game where a Pawn moves from e2 to e4 is written as:

```
game = chess pawn e2 to e4 end
```

The chess game starts with a `Proxy` value, parametrised over the initial `BoardDecorator`. Continuations are applied to this initial value, whose types transform the `BoardDecorator` until the game ends or a rule is broken. Games begin with the board in a set configuration, which is represented by a type `StartDec` of kind `BoardDecorator`. The `chess` functions from the examples shown is a function which takes a continuation and applies it to a `Proxy StartDec` value:

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

It is possible to define a custom starting board instead of the default, which we discuss further in Section 4.3.

Possible continuations are named after the pieces, such as `pawn` and `king`. Each of them expect a position, such as e4 as argument. All positions on the board have corresponding definitions which are typed with the respective Position.

We define a new data type, MoveArgs, to simplify the process of passing information between the continuations in the EDSL. The pawn continuation is used below as an example. All of the other piece continuations are similar and only differ in the PieceName type passed to the continuation via MoveArgs:

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

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

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

```haskell
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 definition is for end, which serves as a terminator for the continuations:

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

Using the above definitions, we can describe a chess game, move by move. Any invalid moves will result in a type error. Consider a modified version of the game we describe at the start of Section 4 in which Black now attempts to move after checkmate. This results in the following type error:

```haskell
-- 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 :: 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:

```haskell
-- 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
```

4.2 Shorthand EDSL

While the longhand notation allows users to fully describe games of Chess, it is considerably more lengthy than Algebraic Notation or other comparable chess notations. Therefore, we also provide a shorthand notation – the one shown at the beginning of Section 4.

Consider the original continuation for moving a Pawn, named pawn. To describe a move using pawn, both the origin and destination squares are required, as well as the use of the continuation to. The shorthand version of pawn is a single letter, p, which takes in only the destination and a continuation and performs the move immediately. The origin position is inferred via a type family MoveTo:

```haskell
p :: Proxy (b :: BoardDecorator)
  -> SPosition toPos
  -> Spec (Proxy (MoveTo Pawn toPos b))
p (dec :: Proxy b) (to :: SPosition toPos) cont
    = cont (Proxy @(MoveTo Pawn toPos b))
```

MoveTo knows the destination square, and knows the PieceName of the piece that moves there. As such, it can calculate the origin for that move using the piece type's movement rules in reverse. Remember, for all pieces except Pawns and Kings, if the piece can move from a to b, then it can also move from b to a. For example, to determine the potential origins for a Bishop moving to destination c5, the Bishop's movement rules are applied to an empty board to see the squares that the Bishop can move to from c5. The resulting list of positions is filtered based on whether there is a valid piece in the original BoardDecorator of the correct team in any of those squares. If the filtered list contains a single

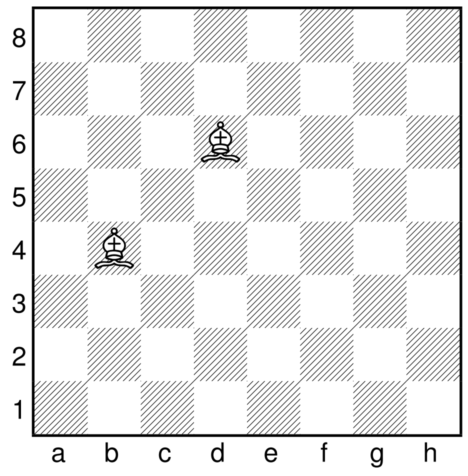

Figure 2. Two White Bishops can move to c5.

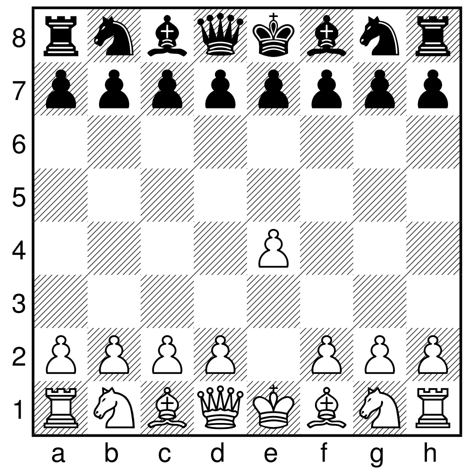

Figure 3. The board resulting from `fenExample`.

piece, then the position of that single piece is extracted from its `PieceInfo` type. Otherwise, there are either no valid origins, which means the user has made a mistake, or multiple valid origins, in which case the longer Chesskell syntax should be used.

As an example of the latter case, consider the board state in Figure 2. There are two bishops who could potentially move to c5, and as such Chesskell will not be able to tell which bishop should move to that location, resulting in a type error. Note that the board is described using the EDSL introduced in the next section:

```
-- Below results in the following type error:
-- * There is more than one White Bishop which
    can move to: C5.
--   Consider using the long-form Chesskell
    syntax instead.
-- * When checking the inferred type
--   twoBishops :: Data.Proxy.Proxy (...)
twoBishops = create
    put _Wh _B at d6
    put _Wh _B at b4
startMoves
    b c5
end
```

An unexpected result of implementing the shorthand syntax is an increase in performance, which we discuss in Section 5.6.

4.3 Creating Chess Boards

Chesskell can also be used to describe Chess boards, either in a modified form of Forsyth-Edwards Notation (FEN) or by individually placing pieces onto the board. This aids the creation of arbitrary board states to test specific movement rules. An example of each is below, resulting in the boards in Figures 2 and 3 respectively:

```
figure2Example = create
    put _Wh _B at d6
    put _Wh _B at b4
end

fenExample = create
    fen1 (ff bR bN bB bQ bK bB bN bR fn0)
    fen2 (ff bP bP bP bP bP bP bP bP fn0)
    fen3 (fn8)
    fen4 (fn8)
    fen5 (ff fn4 wP fn3 fn0)
    fen6 (fn8)
    fen7 (ff wP wP wP wP fn1 wP wP wP fn0)
    fen8 (ff wR wN wB wQ wK wB wN wR fn0)
end
```

With this, we are able to fully describe Chess boards and games, where the types prevent us from describing games that contain illegal moves.

5 Compile Time and Memory Usage

Key observations for us while developing Chesskell have been unpredictable compile-time and memory usage. While the extensions to Haskell's type system as implemented in GHC are very exciting and, as demonstrated, allow us to implement complex rule systems, it is difficult to debug them – GHC's performance with respect to type-level computation is opaque to developers. In this section we outline the issues we faced, and describe approaches we found to solve or avoid them.

The Chesskell library is accompanied by an extensive test suite, implemented using HSpec behaviour-driven tests and deferrable type errors to test the outputs of type families adhere to the FIDE ruleset. Compiling this test suite causes GHC memory usage to exceed 27 GB. These memory issues are not specific to the tests, and can be reproduced in longer Chesskell games. Due to these issues, Chesskell games are practically limited to 12 moves before GHC crashes and runs out of memory (on all systems tested), although there is no theoretical limit to the number of moves.

To investigate the compile times and memory usage further, we developed a GHC benchmark based on Chesskell which comprises 15 different games or scenarios that are compiled with varying numbers of moves, varying versions of GHC (8.8.3 and 8.10.3), and varying compiler options – a total of 356 combinations. We ran these benchmarks on a machine with an Intel i7-10750H CPU (2.60GHz) and 32GB of memory. Out of the 356 individual benchmarks, 163 were never able to run to completion. The most immediate result of this exercise is the observation that, for around 3/4 of the terminating benchmarks, GHC spent most of its time in garbage collection: an average of 4.4 times the time that as was not spent in garbage collection. In the worst case, GHC spent 29.35 times as much time in garbage collection.

In our comparison between GHC 8.8.3 with the "old" garbage collector and GHC 8.10.3 with the "new" garbage collector enabled, GHC 8.8.3 comes out on top. GHC 8.10.3 with the new GC typically requires twice as much memory as GHC 8.8.3 to compile the same game and crashes on more games. This would suggest that GHC 8.10.3 is a step backwards over GHC 8.8.3 with respect to type system performance.

Our results also show that our implementation performs worse on emptier boards. That means moves which take place later on in a game take longer to compile. Indeed, we have not been able to successfully compile *e.g.* a Queen movement on an otherwise empty board without crashing the compiler. It seems that the performance penalty of longer searches across the board for other pieces is significant.

One of our benchmarks compares a game in which a White Bishop moves, followed by a Black Bishop. This game always crashes the compiler. If we replace the White Bishop move with a White Pawn move, the compiler still crashes. If we replace the Black Bishop with a Black Pawn instead, the game compiles using 17GB of memory. Replacing both Bishop movements with Pawn movements leads the game to compile with 16GB of memory. This suggests that the White Bishop move is fairly cheap, while the Black Bishop move causes significant memory usage to the point of crashing the compiler. This might be related to the origin of our board representation and the significant performance impact of having to *e.g.* traverse further down into this structure for one team than the other based on their respective starting positions. While we might expect a similar performance difference to exist in a comparable value-level implementation, we would not expect the difference to be noticeable, let alone be this significant.

For the remaining parts of this section, when we state that a Chesskell game takes *n* seconds to compile, we mean that it takes *n* seconds to compile the Chesskell description of the 1964/65 USSR Championship game between Ratmir Kholmov and David Bronstein. We chose this particular game from our benchmark for the following sections, since most of our baseline benchmarks for this game successfully compiled.

5.1 More Descriptive Error Messages

We intended for Chesskell's type errors to include the number of the move which resulted in a rule violation, to make errors as clear as possible. Ideally, this would result in error messages such as the one below:

```
-- Below results in the following type error:
-- * There is no valid move from E2 to E5.
--   The Pawn at E2 can move to: E3, E4
--   At move: 1
-- * When checking the inferred type
--   game :: Data.Proxy.Proxy (...)
game = chess pawn e2 to e5 end
```

However, adding the move number taken from the current `BoardDecorator` to Chesskell's error messages in this manner causes spikes in compile time and memory usage: a game consisting of a single erroneous move, such as the above, results in nearly 26GB of memory usage, and takes close to 2 minutes to compile. For reference, the average compile time and memory usage for that single move game are under 20 seconds and 4.5GB respectively if the move number is not included.

Taking other information from the `BoardDecorator` type, such as the position of one of the Kings, and putting it into the error message does not result in similar spikes. Due to its effect on compile time and memory usage, Chesskell error messages do not include the move which broke the rule. Instead, we ensure that Chesskell error messages are detailed enough that the user should be able to find the location of the error.

5.2 Checking for Check

One of the most complex parts of Chesskell involves detecting when a King is placed in check. As we explain in Section 3.2.2, an initial naïve implementation of testing for check (named `CheckNoCheck`) involved assembling all possible moves by pieces of the opposite team, and checking if the King position was in that list. Removing the `CheckNoCheck` post-move check entirely (from the codebase using the old implementation) reduced memory usage of an 8-move Chesskell game from 22-23GB to 7.5-8GB, and average compile time from 1 minute 37 seconds to 24 seconds, proving that testing for check was a performance bottleneck.

Attempting to optimise the codebase and reduce memory usage as the result of an "out of memory" error, we developed the ray implementation discussed in Section 3.2.2. Despite the reduced computation performed by GHC with the new implementation, there was no noticeable decrease in memory usage when compiling Chesskell games – GHC continued to run out of memory and crash.

However, splitting out the pre-move rule checks and the post-move rule checks into separate type families helped GHC to terminate, no longer running out of memory. The new implementation below takes less than 1 minute and 30 seconds to compile, and uses an average of 25GB of memory:

```
data Move :: Position -> Position
        -> BoardDecorator -> Exp BoardDecorator
type instance Eval (Move fromPos toPos boardDec)
    = Eval ((ShouldHavePromotedCheck toPos .
    CheckNoCheck)
        (Eval (MoveWithPreChecks fromPos toPos
    boardDec)))

data MoveWithPreChecks :: Position -> Position
                       -> BoardDecorator
                       -> Exp BoardDecorator
type instance Eval (MoveWithPreChecks fromPos
    toPos boardDec) =
    -- most checks are performed here
```

While we do not know the cause for this increase in performance for certain, we speculate it may be related to the number of type variables that must be unified at a given time. In any case, this example shows that sometimes improvements to algorithmic complexity of type-level programs can make little difference, while simply restructuring the type families can. Therefore, we believe that there is a need for better tooling as part of GHC to understand what causes such spikes in compile-time and memory usage.

5.3 Rays for movement

Tweaking our implementation to use the ray-based approach not just for checking whether the King is in check, but also for movement improves the memory use for one game we tested from 30GB to 15GB and reduces the compile time from 400 seconds to around 100 seconds, but increases memory usage by 5GB in another game while reducing the compile time by 50 seconds.

5.4 Type Signatures vs Type Applications

We observed a difference in behaviour between type signatures and type applications. Initially, we did not define `StartDec` by hand for use in the definition of `chess` and instead pieced it together through a lengthy series of type family applications:

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

With this initial version of `StartDec` to set up the game, we attempted two definitions of `chess`: one using a type application and the other using a type signature. These definitions, given below, should be equivalent:

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

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

The version of `chess` using a type application compiles without issues, but causes lengthy (sometimes unrecoverable) pauses at runtime when used. Conversely, when compiling the type signature version, GHC either fails to terminate or crashes due to a lack of memory. This difference in behaviour between the definitions is unexpected and we reported it as a GHC bug[4]. As a workaround, we have written out `StartDec`'s definition in full in the Haskell source file, allowing compilation and usage of either definition with no issues.

5.5 Finger Trees

A common operation in Chesskell is creating lists of potential positions (i.e. move lists and attack lists), and combining them with the append (`++`) operator. However, appending singly-linked lists takes $O(n)$ time. Since this operation is common, we implemented 2-3 Finger trees [8] to take advantage of better amortized append:

```
data FingerTree (a :: Type) where
    Empty  :: FingerTree a
    Single :: a -> FingerTree a
    Deep   :: Digit a -> FingerTree (Node a)
           -> Digit a -> FingerTree a

data Node a = Node2 a a | Node3 a a a
data Digit a = One a | Two a a | Three a a a |
    Four a a a a
```

To illustrate how appending two type-level finger trees works, extracts of the implementation are given below. However, most cases are omitted for brevity:

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

```
type instance ((Deep leftL leftM leftR) ++ (Deep
    rightL rightM rightR))
    = Deep leftL (AddTree1Digit leftM (ToNode
    leftR rightL) rightM) rightR
-- ...

type family AddTree1Digit (t1 :: FingerTree a)
                          (d1 :: Digit a)
                          (t2 :: FingerTree a)
                          :: FingerTree a where
    AddTree1Digit Empty dig rightTree
        = AddDigitLeft dig rightTree
    AddTree1Digit (Single x) dig rightTree
        = x :< AddDigitLeft dig rightTree
    -- ...
    AddTree1Digit (Deep leftL leftM leftR) dig
    (Deep rightL rightM rightR)
        = Deep leftL (AddTree1Digit leftM (ToNode
    leftR rightL) rightM) rightR
```

We replaced all append and mapping operations over lists with corresponding operations on Finger Trees. To our disappointment, changes were negligible: compile time went down by an average of 5 seconds and memory usage was reduced by an average of 0.5 GB. Although this is an improvement, it again shows that algorithmic complexity is not responsible for the compiler performance we observe.

5.6 Chesskell Shorthand Syntax

One of the most dramatic optimisations in terms of compile-time and memory usage was unexpected and accidental: it resulted from the implementation of the shorthand syntax (Section 4.2).

With the longhand notation, any Chesskell game longer than 10 moves would cause GHC to crash (on the authors' machines). We predicted that the shorthand syntax would degrade performance and reduce this number, since the type-level model of Chess would not only have to perform all of the move checking, but also determine which piece(s) could move to the destination square. However, since it allowed more concise description of Chess games in line with existing Chess notation, we deemed this trade-off acceptable.

We developed a working version of the shorthand syntax, without going through extensive optimisations or performance testing. Despite the greater amount of work required from GHC to compile these short-hand descriptions, there were notable and significant decreases in average compile time and memory usage: with the longhand syntax, compiling a 10-move Chesskell game took an average of 3 minutes and 25GB of memory, and a 12-move Chesskell game would crash every time. With the shorthand syntax, a 10-move Chesskell game compiles in around 1 minute 20 seconds, using 24GB of memory, and a 12-move Chesskell game compiles in an average of 1 minute 50 seconds, using 25GB of memory.

The shorthand syntax allows for us to express games which are longer by 2 moves, and yet incur no additional penalties on compile time or memory usage. We again speculate that the improvement is due to fewer type variables that need to be unified.

6 Related Work

Chesskell is, to the best of our knowledge, unique: while there are value-level implementations of Chess in many programming languages, we are not aware of other implementations of Chess rules in a *type system*. There have been allusions to Chess at the type-level through solving the N-queens problem in dependently typed languages, such as Idris[5]. The N-queens problem makes use of some Chess rules, including the Queen's attack positions[6], but it is not the full set of Chess rules.

However, Chesskell draws from, and owes much to, other work related to type-level rule checking, EDSLs, and Chess programming in general. We discuss some of the relations and differences below.

6.1 Haskell-Embedded Domain-Specific Languages

There is work on Haskell EDSLs in other domains to enforce certain behaviours at compile-time. Mezzo [14] is an EDSL for music composition, which checks if a described piece of music conforms to a given musical ruleset during compilation of the program. For instance, one can apply classical harmony rules to ensure that the piece of music we compose would not go against the rules of that musical period. This EDSL is similar to Chesskell in the aim of performing compile-time checks of rulesets that are usually checked dynamically, although not in the same application domain. Mezzo is an example of a complex domain with complex rules (classical harmony) being modelled and enforced at the type-level in Haskell.

BioShake [1] 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 are lengthy. BioShake goes further, however, providing tools to allow parallel execution of these workflows.

[5]https://github.com/ExNexu/nqueens-idris
[6]A Queen can attack in a straight line in any direction.

6.2 Chess in Computer Science

Chess has a rich history in Computer Science. Getting computers to play Chess was tackled as far back as 1949 [12], and since then much progress has been made in the field. Chess has been used to educate [6], to entertain, and to test out machine learning approaches [2]. Due to its status as a widely known game of logic, with a well-defined rule set, it is a prime candidate to act as the general setting for programming problems. Indeed, the famous NP-Complete problem referenced above, the N-Queens problem [5], relies on the rules of Chess.

Many of these Chess-related programs are written in Haskell, and are publicly available[7,8]. A large number 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 does not output a list of strong moves, and does not mediate an ongoing game – it simply takes in the moves performed, and states whether they are valid Chess moves or not.

7 Conclusions

We have presented Chesskell, a Haskell EDSL for describing Chess games where a full encoding of the FIDE 2018 Laws of Chess in the types rules out illegal moves. In addition to presenting an EDSL and type-level model, reporting on our work in implementing such a complex set of rules encoded in Haskell's type system serves multiple, additional purposes. Firstly, it provides a stress test for GHC's type system that we hope can serve as a benchmark for the compiler's type checking performance going forward.

More importantly, we reported on areas of friction in developing such type-level models and identified where improvements could be made to the compiler. Despite significant work in the Haskell community to extend Haskell's type-level programming capabilities and us identifying techniques for optimising and troubleshooting type-level code, a number of workarounds are still required in practice. The same task in a dependently-typed programming language would have have been easier as such workarounds would not have been required. Furthermore, during development, we found type errors to be so long that we had to work out types by hand to troubleshoot what went wrong.

[7]https://github.com/mlang/chessIO
[8]https://github.com/nionita/Barbarossa

While our implementation can be considered idiomatic in the sense that the rules are expressed in a human-readable manner and use representations which are intuitive to functional programmers, the resulting performance prevents us from describing longer games as memory usage becomes the limiting factor. In our testing, describing a game of around twelve moves can consume more than 20GBs of memory.

Reasoning about our definitions' impact on compile-time performance and memory usage has proved difficult: optimisation techniques we expected to improve performance did not and changes we expected to make no difference ended up improving performance. These findings make it clear to us that, in order for developers to be able to encode complex rules in Haskell's type system, it must become more transparent how the compiler will react to a given definition or there must be tools which assist developers with this.

7.1 Future Work

While our implementation uses a board representation based on two-dimensional, length-indexed vectors and we investigated a representation using type-level finger trees, neither presented any significant improvements in terms of compile-time over the other. However, we believe it would be worthwhile to examine further representations, such as a Bitboard [7] using type-level `Nat`-s to reduce the amount of memory and, importantly, number of type variables that must be unified.

Although the long compile times and significant memory usage, as well as the associated crashes of the compiler, currently prevent us from exploring practical applications of Chesskell, we believe it to be a suitable system for at least three potential applications:

- Building type-safe AI rules for a Chess engine. By suitably typing or constraining the types of functions which implement the AI behaviours using Chesskell's types, we could *e.g.* ensure that the AI does not exhibit invalid behaviours.
- Combined with *e.g.* session types, Chesskell could be used to type the communication between two player processes to ensure no errors are introduced into the game state.
- Typesetting Chess games, such as the board diagrams in this paper, in a programmatic approach using a library such as HaTeX[9] could be typed using Chesskell to prevent us from introducing mistakes and ensuring that several states of a given game follow from each other.

[9]https://hackage.haskell.org/package/HaTeX

Acknowledgments

We thank the anonymous reviewers for their constructive feedback on both review versions of this paper. Their feedback encouraged us to develop more extensive benchmarks based on Chesskell and a suggestion from one of the reviewers helped us improve Chesskell's performance further.

References

[1] Justin Bedő. 2019. BioShake: a Haskell EDSL for bioinformatics workflows. *PeerJ* 7 (2019), e7223.

[2] Marco Block, Maro Bader, Ernesto Tapia, Marte Ramírez, Ketill Gunnarsson, Erik Cuevas, Daniel Zaldivar, and Ral Rojas. 2008. Using reinforcement learning in chess engines. *Research in Computing Science* 35 (2008), 31–40.

[3] Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361

[4] Richard A Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed type families with overlapping equations. *ACM SIGPLAN Notices* 49, 1 (2014), 671–683.

[5] Ian P Gent, Christopher Jefferson, and Peter Nightingale. 2017. Complexity of n-queens completion. *Journal of Artificial Intelligence Research* 59 (2017), 815–848.

[6] Dmitri A Gusev. 2018. Using Chess Programming in Computer Education. *Association Supporting Computer Users in Education* (2018).

[7] Ernst A Heinz. 1997. How DarkThought plays chess. *ICGA Journal* 20, 3 (1997), 166–176.

[8] Ralf Hinze and Ross Paterson. 2006. Finger trees: a simple general-purpose data structure. *Journal of functional programming* 16, 2 (2006), 197–218.

[9] International Chess Federation (FIDE). 2020. FIDE Handbook. https://handbook.fide.com/.

[10] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-order type-level programming in Haskell. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–26.

[11] Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, and Manuel Chakravarty. 2007. Towards open type functions for Haskell. *Implementation and Application of Functional Languages* 12 (2007), 233–251.

[12] Claude E Shannon. 1950. XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41, 314 (1950), 256–275.

[13] Gerald Jay Sussman and Guy L Steele. 1998. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation* 11, 4 (1998), 405–439.

[14] Dmitrij Szamozvancev and Michael B Gale. 2017. Well-typed music does not sound wrong (experience report). In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*. 99–104.

[15] Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. 53–66.