Chesskell: A Two-Player Game at the Type Level

Anonymous

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.

Keywords: Haskell, Chess, First Class Families

ACM Reference Format:

Anonymous. 2021. Chesskell: A Two-Player Game at the Type Level. In *Proceedings of ACM SIGPLAN Haskell Symposium 2021 (Haskell'21).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

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 [1, 2]. 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*.

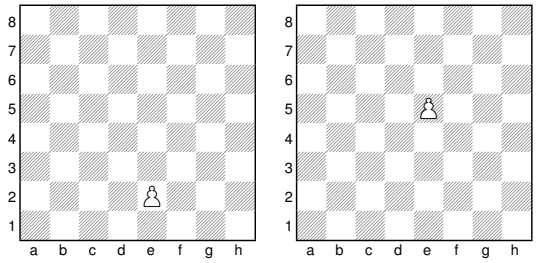

Figure 1. A White Pawn moves from e2 to e5: an invalid move.

Chess is a popular game with a well-defined ruleset [3] 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.
```

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 contributions are:

Haskell'21, August 26-27, 2021, Virtual

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

- We provide an overview of Li-yao Xia's *First Class Families* technique (Section 2), which we utilise extensively in Chesskell for first-class functions at Haskell's type-level;
- 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 [1, 2] 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.

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*[1]. 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
```

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

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) -> f a -> Exp (f 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 ]`.

It is worth noting 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:

```
data Or :: Bool -> Bool -> Exp Bool
type instance Eval (Or True _) = True
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 section 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.

3 Chess Model

Each move in the Chess game 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 First Class Family, 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 return a new board state in which the move has been made. Additionally, it updates relevant piece information for the pieces that have moved, which we further detail below:

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

The core Chess model within Chesskell is expressed through this central `Move` First Class Family. The model is checked move-by-move, with the current board state, as well as some further information, carried between moves via a `BoardDecorator`. This `BoardDecorator` contains all information necessary to encapsulate the current state of a game of chess; in other words, Chesskell does not rely on any global state.

3.1 Chess Types

This section details the types involved, including the board representation, and explains how Chess rules are implemented and enforced. We describe Chesskell's types from the bottom up, 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. Thanks to promotion [4], these definitions are immediately available for use at the type level.

```
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 bottom-left of the board, and "h8" is the top-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`, using 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. We define a type family `IsValidPosition` to ensure correctness, which outputs **`True`** if the given position is a valid chess position and **`False`** otherwise. We give the definition of the `Position` type below:

```
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* (as we discuss in Sections 3.3.1 and 3.3.2), and so is included in the `PieceInfo` type.

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

```
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 of a fixed size and must remain that size; it must always be an 8x8 grid of 64 squares. Using a Haskell type-level list would require checks on its size before each move—instead, we use length-indexed vectors to represent a collection that must be a certain size:

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

We express the Chess board using these vectors—a vector of 8 vectors of 8 `Maybe Piece`-s. We use `Maybe Piece` instead of `Piece` because a board square does not necessarily contain a piece (in fact, at least half the board squares are always empty):

```
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 [5], with the exception that subclassing and superclassing are not features of Haskell. We use `BoardDecorator` by default 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.

We give the definition of `BoardDecorator` below:

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

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 rule-check type family has the suffix `-Check`, such as `NotTakingKingCheck`. These checks are broadly split into pre-move checks, and post-move checks. Each check has a custom error message to make clear to the user where the rule violation occurs.

We give the definition of `NotSamePosCheck` below as an example, which checks that the move is between two distinct positions, and generates a type error otherwise:

```
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)
```

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. The `PieceMoveList` First Class 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.

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` with kind `Position -> Position -> BoardDecorator -> Exp BoardDecorator`, which checks if there is a piece at the first position that can move to the second position. Additionally, for 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. `PieceMoveList` cannot be used to determine which squares a piece can attack; so 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 when 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`.

Early versions of Chesskell naively computed and combined all attack lists for all pieces, and simply checked if the King's position was an element of that combined list. A more efficient approach, found in Chesskell today, is to emulate other pieces' movement code 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—Kings. This is deliberate; 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 below for understanding. 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 Exceptional Rules

There are a few Chess rules that are dissimilar from all other Chess rules; and implementing these rules requires a different approach from other rule implementations. Since they are of particular interest, the implementation of these rules is detailed here.

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 tests before it can proceed.

There are two varieties of Castle; Queen-side Castle and King-side Castle, depending on the direction that the King moves in (either left or right). Essentially, the King

moves either 2 or 3 spaces towards the Rook, and the Rook wraps around to the other side of the King.

In Chesskell, we model castling as a move by the King; valid castling positions are added to the King's move list. A type family, `CanCastle`, is responsible for checking if the King of a certain team can indeed perform castling in either direction, returning a pair of Booleans to state whether the King can castle left or right. The below code snippet illustrates a part of this process:

```
type family CanCastle (t :: Team) (b ::
    BoardDecorator) :: (Bool, Bool) where
    CanCastle team boardDec = If' (Not'
    (HasKingMoved team boardDec))
        (ID (CanCastleToEitherRook team boardDec))
        (ID '(False, False))
```

The above code first checks if the King has moved; if they have not, then it checks if both Rooks have not moved. If they have not moved either, then it determines if any of the spaces the King would move through are in check, and then if there are any pieces between the King and the Rook. These checks must pass for the King to be able to castle in that direction; and a pair of Booleans is returned signifying if the King can castle in either direction. For instance, if the King can castle left but not right, then `CanCastle` will return `'(True, False)`.

This extended Castling check illustrates why it is useful to have each piece's move count in the `PieceInfo` type; it enables quickly determining 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.

There are no circumstances under which they are obligated to castle, and so there is no pre- or post-move castle check. The castling positions are included in each King's move list. However, a King cannot castle to capture another piece; so these positions are not a part of the King's attack list.

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, they cannot capture a piece this way—they can only capture one diagonal space in front of themselves.

This means that a Pawn can indeed 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; that of *en passant*.

A Pawn can perform an *en passant* capture if a Pawn of the opposite team has moved forwards by two spaces last turn (which can only occur if that was the opposite Pawn's first move), and ended up next to the attacking Pawn. In this situation, the original attacking Pawn can move diagonally to the empty space behind the opposite team's Pawn, and capture it.

This capture rule is dependent on several factors; the last move made, as well as the relative positions of the pieces. Furthermore, the capture move does not result in the attacking piece landing on the square of the piece being captured; making it distinct from all other capture rules.

The majority of the *en passant* logic is implemented via a First Class Family, `GetEnPassantPosition`, which is responsible for determining if an *en passant* capture is a valid move for a pawn at a given position:

```
data GetEnPassantPosition :: Position
                          -> BoardDecorator
                          -> Exp [Position]
type instance Eval (GetEnPassantPosition pos
    boardDec)
    = If'
    -- condition
    (Eval ((GetLastPosition boardDec) `In` Eval
    (GetLeftRightPositions pos)))
    -- then
    (FromMaybe '[] (EnPassantPosition
    (GetMovingTeam boardDec) . PiecePosition)
        (Eval (GetPieceAtWhichDec boardDec
    (GetLastPosition boardDec) (IsPawn .&.
    PawnMovedTwoLast))))
    -- else
    (ID '[])
```

Firstly, it checks if the position moved to last turn (fetched from the `BoardDecorator` with `GetLastPosition`) is either to the left or the right of the given Pawn position. This is the check to determine whether any piece moved last turn to the left or right of the current Pawn. If this check passes, then there is another check; whether the piece that made that last move was a Pawn, and whether it moved two spaces. If that check passes, then the piece's position is fetched and the row is either incremented or decremented (depending on the attacking team) by `EnPassantPosition` to get the single space either above or below that piece—the target square to perform *en passant* capture.

We use First Class Families to simplify these logical checks. The First Class Family `GetPieceAtWhichDec` returns a `Maybe Piece` depending on whether there is a piece at a given position which fulfils a given predicate, returning `Nothing` if the predicate does not evaluate to true. Additionally, a type-level First Class Family version of `FromMaybe` is used to either transform the `Nothing` type into an empty list `'[]`, or to transform the wrapped value into a singleton list containing the *en passant* capture position.

Implementing *en passant* captures was the driving factor that prompted the creation of the `BoardDecorator` type, since the last position moved to was required as part of the process. Ultimately, *en passant* captures are implemented in Chesskell, as we demonstrate with the successful compilation of the below game:

```
enPassant = chess
    p d4 p a6
    p d5 p e5
    p e6  -- En Passant capture!
end
```

4 DSL for Describing Chess Games

The final product of Chesskell 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
    p e4 p f5
    q f3 p g5
    q h5
end
```

Note that the spacing is purely for style reasons; the above game could just as easily be written as:

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

The EDSL, as we explain in more detail below, uses the aforementioned `Move` function to perform type-level rule checking of the described chess game. A Continuation Passing Style (CPS) [6] scheme forms the foundation for the EDSL, with inspiration drawn from Dmitrij Szamozvancev's Flat Builders pattern [7]. While the 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))
```

4.1 EDSL Types

The chess game starts with a `Proxy` value, its type parameterised with 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. We use type application [8] to set the type variable for the `Proxy` type:

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

The core continuations are named after the pieces, such as `pawn` and `king`. We use singleton types [9] rather than `Proxy` types in case of any future extensions requiring term-level knowledge of the types. 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 and the continuation stream.

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

Using the above continuations, we can describe a chess game, move by move. Consider the game expressed in the EDSL which we describe in Section 4. It compiles successfully; but should Black attempt to move after checkmate, GHC produces a type error, 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
```

4.1.1 Shorthand syntax.

While the above continuations allow the user to fully describe a game of Chess, the resulting notation is considerably more lengthy than Algebraic Notation or other comparable chess notations. As such, a shorthand syntax was introduced towards the end of development, to allow more concise description of Chess games. (This is the syntax we demonstrate in Section 4.)

Consider the original continuation for moving a Pawn, named `pawn`. To move using `pawn`, both the origin and destination squares are required, as well as the use of the continuation `to`. The new shorthand continuation is

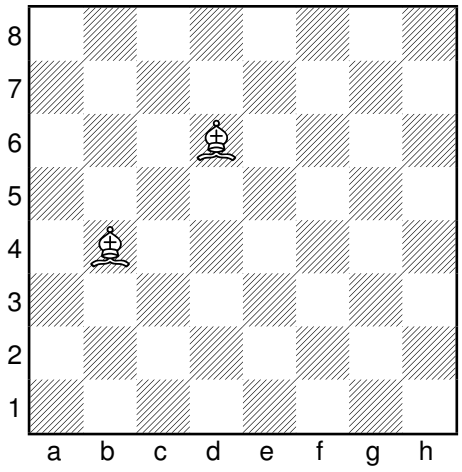

Figure 2. A board where two White Bishops can move to c5.

a single letter, `p`, which takes in the destination position and a continuation and performs the move immediately; calculating the origin square is left to the type-level model, via a type family `MoveTo`:

```
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 square 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 square a to square b, then it can also move from b to a.) For example, to determine the potential origin squares for a Bishop moving to destination square c5, the Bishop's movement rules are applied to an empty board to see the squares that the Bishop can move to from c5.

Then, this list of squares 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 resulting filtered list has length 1 (i.e. it contains a single piece), then the position of that single piece is extracted from its `PieceInfo` type. Otherwise, there are either no valid origin squares, which means the user has made a mistake; or multiple valid origin squares, 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 square c5, and as such Chesskell will not be

able to tell which bishop should move to that location, and will fail to compile with a type error:

```
-- 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
```

Despite the late introduction of this shorthand syntax, it fits into Chesskell as a form of Chess notation as well as a demonstration of type-level modelling. In fact, as we discuss in Section 5.5, its addition resulted in unexpected performance improvements.

4.2 Creating Chess Boards

Chesskell can also be used to create Chess boards, either through a modified EDSL form of Forsyth-Edwards Notation (FEN), or by individually placing pieces onto the board. These features were added to simplify testing, allowing creation of arbitrary board states to test specific movement rules. An example of each is below, demonstrating the creation of 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
```

5 Compile Time and Memory Usage

For us, a key observation while developing Chesskell has been unpredictable compile-time and memory usage issues. These issues demonstrate a lack of transparency from GHC as to the performance and behaviour of type-level computation. In this section we outline the issues we faced, and describe our approaches in solving or avoiding them.

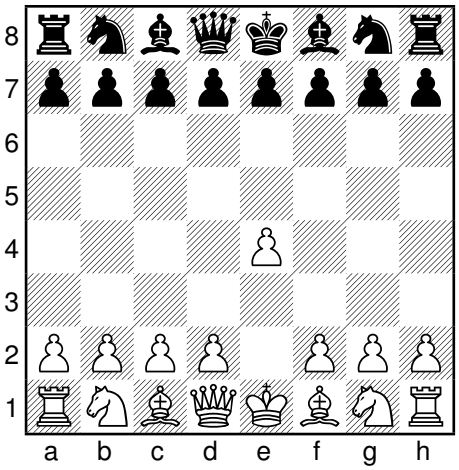

Figure 3. A complex board to demonstrate the EDSL form of FEN notation.

We first observed these issues while adding to a test suite for Chesskell. The test suite is primarily HSpec behaviour-driven tests [10], so as to test the outputs of type families and ensure adherence to the FIDE ruleset.

Compiling this test suite causes memory usage to exceed 27 GB. These memory issues are not specific to HSpec testing, and can be reproduced in longer Chesskell games. Due to these issues, Chesskell games are limited to 12 moves maximum, before GHC crashes and runs out of memory (on all systems tested).

Below, 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. This game was chosen as the benchmark for compile-time comparison (though others are also used to test the functionality of Chesskell), both King- and Queen-side castling.

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 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.

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 performance-heavy parts of Chesskell involves detecting when check has occurred within the game. As we explain in Section 3.2.2, an initial naive 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.

The pre- and post-move checks were composed together within a single type family, along with the actual movement of the pieces themselves, like so:

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

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" crash, we created the new ray implementation we detail 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) = Eval (
    (MoveNoChecks fromPos toPos .
        CanMoveCheck fromPos toPos .
        NotTakingKingCheck toPos .
        NotTakingOwnTeamCheck toPos .
        NotSamePosCheck fromPos toPos .
        NotLastToMoveCheck fromPos .
        TeamCheck fromPos) boardDec)
```

While the cause for this increase in performance is unknown, we suspect it is due to the requirement for unification of many type variables when performing type-level function composition. We believe that more clarity from GHC, with regards to its performance and behaviour, would aid in diagnosing the causes for these differences.

5.3 Type Signatures vs Type Applications

We observed a difference in behaviour between type signatures and type applications during development of the EDSL. This was due to the initial definition of `chess`; we did not manually create the starting board configuration (stored in a `BoardDecorator` type named `StartDec`), but instead pieced it together through a lengthy series of type family applications:

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

We used this initial version of `StartDec` 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. These definitions, given below, should be equivalent in behaviour:

```
-- 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 (sometimes unrecoverable) pauses at runtime when used. Conversely, when compiling the type signature version, GHC would either fail to terminate within a reasonable time frame, or crash due to a lack of memory. This difference in behaviour between the definitions is unusual; so we filed a GHC bug report[2]. As a work-around until the bug is fixed, `StartDec` is manually written out in full in the Haskell source file, allowing compilation and usage of either definition with no issues.

5.4 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, Haskell's singly-linked lists take $O(n)$ time for append operations, as there is no quick way to reach the end of the list. Since this operation is common, we considered the use of an alternative data structure with faster amortized append time; 2-3 Finger trees [11], used in `Data.Seq`.

We developed a full type-level implementation of 2-3 Finger Trees, in an attempt to reduce memory usage and compile time. All append and mapping operations over lists were replaced with corresponding operations over Finger Trees, and we conducted performance testing. Below, we include the initial data type definition:

```
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
```

Additionally, we show parts of the definition for an append operation:

```
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
```

The gains measured due to these 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. Due to the lack of noticeable gains in compile time or memory usage, we ceased work on the Finger Tree branch of the Chesskell codebase.

5.5 Chesskell Shorthand Syntax

One of the most dramatic optimisations in terms of compile-time and memory usage was unexpected and accidental. As development of Chesskell drew to a close, we decided to shorten Chesskell's syntax and develop the shorthand we detail in Section 4.1.1. This was not intended as an optimisation, but rather as a minor extension to the project.

At the time, any Chesskell game longer than 10 moves would cause GHC to crash (on the author's 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 provably higher amount of work required from GHC to compile these short-hand Chesskell games, there were notable and significant decreases in average compile time and memory usage.

Previously, with the longer 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. But with the new shorter syntax, a 10-move Chesskell game compiles in around 1 minute 20 seconds, using 24GB of memory, and a 12-move Chesskell game

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

compiles in an average of 1 minute 50 seconds, using 25GB of memory.

The shorthand syntax allows for us to express Chess games which are longer by 2 moves, and yet incur no additional penalties on compile time or memory usage. After measuring these differences, we tested the shorthand syntax on a branch using `Proxy` types instead of singletons; and the performance differences were negligible. The improvement is not directly related to singletons or proxies, but to the number of type variables present. A clearer notion of how GHC tackles type-level computation would give insight into performance, and shed light on further optimisation strategies.

6 Related Work

Chesskell is, at the time of writing, unique; we are aware of no other type-level Chess implementations. There have been allusions to Chess at the type-level through solving the N-queens problem in dependently typed languages, such as Idris[3]. The N-queens problem makes use of some Chess rules, including the Queen's attack positions[4]; but as the end goal is not to successfully model a game of Chess, it is not a full type-level Chess implementation.

However, Chesskell draws from, and owes much to, many well-established research areas, including type-level rule checking, EDSLs, and Chess programming in general. This section of the report will detail related work, and how Chesskell differs from existing literature.

6.1 Haskell-Embedded Domain-Specific Languages

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. DSLs exist for the purpose of modelling some domain in a language; so Haskell-Embedded DSLs are a natural use case for domain-specific modelling with types. If an EDSL comes with the guarantee that all compiling programs written in that language will not exhibit invalid behaviour, then the EDSL becomes an attractive way to interact with that domain.

Mezzo [7] is an EDSL for music composition, which checks if the 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 you compose would not go against the rules of the musical period. This EDSL is similar to Chesskell in aim, if not in application domain; performing compile-time checks of rulesets that are commonly checked dynamically. Mezzo is an example of a complex domain with complex rules (classical harmony) being modelled and enforced at the type-level. This is similar to Chesskell's objectives, and was a direct inspiration for the project.

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 BioShake and other EDSLs [13] focus on (and achieve) high performance, Chesskell has no such focus. This is primarily because very few parts of the rule-checking process can be parallelised; much of the move handling and order of rule checks must be done sequentially.

6.2 Chess in Computer Science

Chess has a rich history as a study area of Computer Science. Getting computers to play Chess was tackled as far back as 1949 [14], and since then many developments have been made in the field. Chess has been used to educate [15], to entertain, and to test out machine learning approaches [16]. 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 [17], relies on the rules of Chess.

Many of these Chess-related programs are written in Haskell, and are publicly available[5,6]. 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.

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

Reporting on our work in implementing such a complex set of rules encoded in Haskell's type system serves multiple purposes:

- It provides a stress test for GHC's type system that we hope can be used to build regression tests for the compiler's type checking performance.
- We report on areas of friction in developing such type-level models of complex rule sets where improvements could be made to the compiler.
- We identify some techniques for optimising and troubleshooting type-level code.

While our implementation can be considered idiomatic in the sense that the rules are expressed in a human-readable manner and use intuitive representations, the resulting performance does not allow us to describe longer games where memory usage becomes the limiting factor. In our testing, describing a game of around twelve moves will 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 which we expected to make no difference ended up improving performance. These surprising findings make it clear to us that, in order for developers to be able to encode complex business logic 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 allow developers to profile and debug their encodings.

7.1 Future Work

While our implementation uses a board representation based on two-dimensional, length-indexed vectors and we investigated an equivalent 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 [18] using type-level `Nat`-s to potentially reduce the amount of memory and, importantly, number of type variables that must be unified.

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

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

[11] R. Hinze and R. Paterson, "Finger trees: a simple general-purpose data structure," *Journal of functional programming*, vol. 16, no. 2, pp. 197–218, 2006.

[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] C. E. Shannon, "Xxii. programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, no. 314, pp. 256–275, 1950.

[15] D. A. Gusev, "Using chess programming in computer education.," *Association Supporting Computer Users in Education*, 2018.

[16] M. Block, M. Bader, E. Tapia, M. Ramírez, K. Gunnarsson, E. Cuevas, D. Zaldivar, and R. Rojas, "Using reinforcement learning in chess engines," *Research in Computing Science*, vol. 35, pp. 31–40, 2008.

[17] I. P. Gent, C. Jefferson, and P. Nightingale, "Complexity of n-queens completion," *Journal of Artificial Intelligence Research*, vol. 59, pp. 815–848, 2017.

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