Chesskell: Modelling a Two-Player Game at the Type-Level

Toby Bailey

March 2, 2021

Department of Computer Science

Why do type systems exist?

Type systems exist because:

Why do type systems exist?

Type systems exist because: we want to avoid errors [1].

Why do type systems exist?

Type systems exist because: we want to avoid errors [1].

But not domain-specific logical errors?

Why do type systems exist?

Type systems exist because: we want to avoid errors [1].

But not domain-specific logical errors?

Solution: model your domain in the types!

Why Chess?

- It's popular and internationally known;

Why Chess?

- It's popular and internationally known;
- It's been widely studied in the field of Computer Science [2] [3];

Why Chess?

- It's popular and internationally known;
- It's been widely studied in the field of Computer Science [2] [3];
- It has a *well-defined ruleset*.

Why Chess?

- It has a *well-defined ruleset*.

A note on Chess

There are two *Teams*; Black and White.

A note on Chess

There are two *Teams*; Black and White.

Each Team has 16 *Pieces*; 8 Pawns, 2 Rooks, 2 Bishops, 2 Knights, a Queen, and a King.

A note on Chess

There are two *Teams*; Black and White.

Each Team has 16 *Pieces*; 8 Pawns, 2 Rooks, 2 Bishops, 2 Knights, a Queen, and a King.

Each piece has different movement rules, allowing them to move around the 8x8 board.

A note on Chess

There are two *Teams*; Black and White.

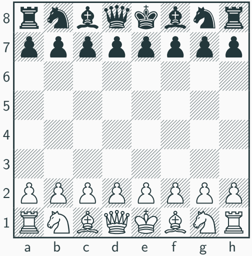
Each Team has 16 *Pieces*; 8 Pawns, 2 Rooks, 2 Bishops, 2 Knights, a Queen, and a King.

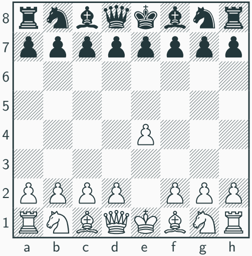
Each piece has different movement rules, allowing them to move around the 8x8 board.

Pieces can remove other pieces from the board via *capture*; which almost always involves moving to the other piece's square.

A Short Example

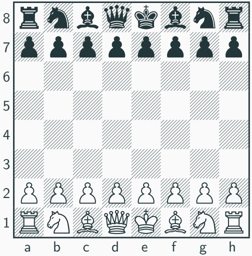
Below is a valid move by a White Pawn:

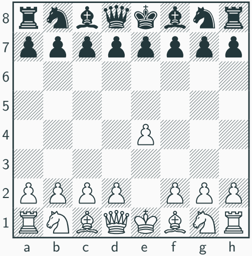




A Short Example

Below is a valid move by a White Pawn:

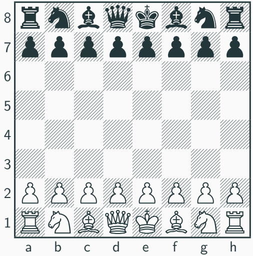


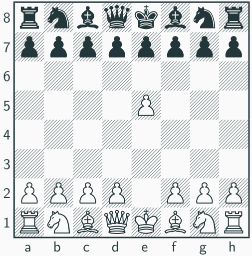


```
chess
    pawn e2 to e4
end
```

A Short Example cont.

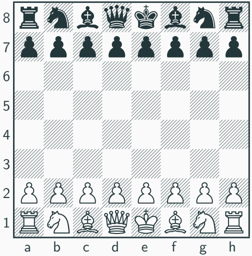
Below is an *invalid* move by a White Pawn:

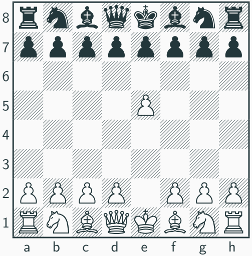




A Short Example cont.

Below is an *invalid* move by a White Pawn:

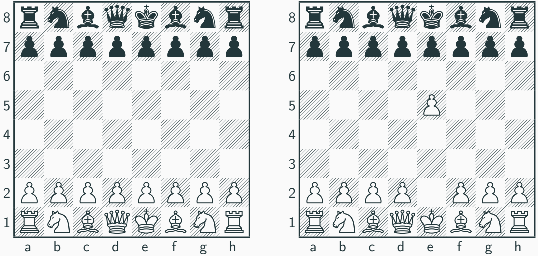





```
chess
    pawn e2 to e5
end
```

A Short Example cont.

Below is an *invalid* move by a White Pawn:




```
-- Fails to compile with type error:
--      * There is no valid move from E2 to E5.
--      The Pawn at E2 can move to: E3, E4
chess
    pawn e2 to e5
end
```

A Little Terminology

In Haskell, values have *types*, and types have *kinds*.

A Little Terminology

In Haskell, values have *types*, and types have *kinds*.

Luckily, we can *promote* types to kinds with the `-XDataKinds` extension [4].

A Little Terminology

In Haskell, values have *types*, and types have *kinds*.

Luckily, we can *promote* types to kinds with the `-XDataKinds` extension [4].

```
data Book = Fiction | NonFiction
```

With the extension, this creates the values `Fiction` and `NonFiction` of type `Book`, and also the *types* `'Fiction` and `'NonFiction` of kind `Book`.

A Little Terminology cont.

In Haskell, you compute on values with *functions*.

```
factorial :: Int -> Int
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

A Little Terminology cont.

In Haskell, you compute on values with *functions*.

```
factorial :: Int -> Int
factorial 0 = 1
factorial x = x * factorial (x - 1)
```

You can't compute on types with functions; instead, you have to use *type families* [5] [6].

```
type family Factorial (x :: Nat) :: Nat where
    Factorial 0 = 1
    Factorial x = Mult x (Factorial (x - 1))

type family Mult (x :: Nat) (y :: Nat) :: Nat where
    Mult 0 y = 0
    Mult 1 y = y
    Mult x y = y + (Mult (x - 1) y)
```

Problems with Type Families?

Lots of idiomatic Haskell code relies on functions being *first-class*; partial application, mapping, etc.

```
x = map (+ 2) [1,2,3]
```

Problems with Type Families?

Lots of idiomatic Haskell code relies on functions being *first-class*; partial application, mapping, etc.

```
x = map (+ 2) [1,2,3]
```

But type families can't be partially applied!

```
-- Type error: type family (+) was expecting 2
    arguments, got 1
type X = Map (+ 2) '[1,2,3]
```

Introducing First Class Families

Thanks to Li-yao Xia, we have First Class Families!

It relies on a data type `Exp`, and a type family `Eval`, to create a type-level interpreter:

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

Making a First Class Family

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

Making a First Class Family

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

becomes:

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

Creating the Types for Chess

Using promotion (as we explain earlier), we define appropriate types for use with type families in Chess. We give some examples below, in both regular and GADT [7] syntax:

```
data Team = Black | White

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

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

The Board type

To avoid repeated length checks, we use *length-indexed vectors* with a type-level implementation of Peano natural numbers:

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

The Board type

To avoid repeated length checks, we use *length-indexed vectors* with a type-level implementation of Peano natural numbers:

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

Since a Chess board is always an 8x8 grid, we use vectors of vectors:

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

The BoardDecorator type

In the codebase, we use a wrapper data structure to hold helpful information along with the `Board` for rule checking:

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

Representing Movement

Movement is expressed as a single First Class Family:

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

Representing Movement

Movement is expressed as a single First Class Family:

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

Thanks to First Class Families, we can extend this with rule-checking naturally; using a type-level version of the function composition operator, `(.)`:

```
PostMoveCheck2 . PostMoveCheck1 . Move fromPos toPos
    . PreMoveCheck2 . PreMoveCheck1
```

Interacting with Type-Level model at the value level

The core idea is wrapping the BoardDecorator type in a Proxy, so that it can be passed around within a value by functions:

```
data Proxy a = Proxy

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

Interacting with Type-Level model at the value level

The core idea is wrapping the `BoardDecorator` type in a `Proxy`, so that it can be passed around within a value by functions:

```
data Proxy a = Proxy

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

But this would still look similar to Haskell syntax; we need a new approach.

Creating the EDSL

Ideally, the EDSL should look like existing chess notation:

```
1. e4 e5 2. Nf3 Nc6 3. Bb5 a6
```

Creating the EDSL

Ideally, the EDSL should look like existing chess notation:

```
1. e4 e5 2. Nf3 Nc6 3. Bb5 a6
```

Can achieve using Continuation Passing Style, inspired by Dima Szamozvancev's Flat Builders work [8].

Making continuations for Chess

The core continuations are `chess`, the piece continuations, and `to`:

Making continuations for Chess

The core continuations are `chess`, the piece continuations, and `to`:

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

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

Making continuations for Chess

The core continuations are `chess`, the piece continuations, and `to`:

```
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))
```

Making continuations for Chess

The core continuations are `chess`, the piece continuations, and `to`:

```
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)))
```

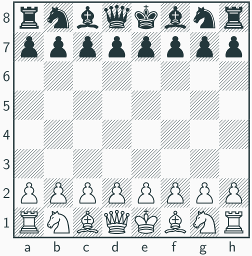
Using the Chess continuations

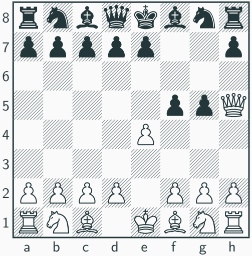
All of the above continuations can be chained together (along with `end` which ends the continuation stream) like so:

```
game = chess pawn a1 to a2 end
```

A Longer Example

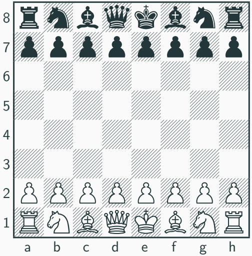
Below is a short game, ending in checkmate by White:

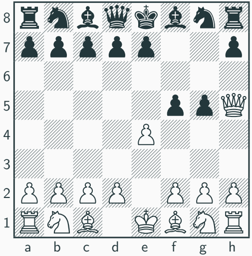




A Longer Example

Below is a short game, ending in checkmate by White:





```
game = chess
    pawn e2 to e4
    pawn f7 to f5
    queen d1 to f3
    pawn g7 to g5
    queen f3 to h5
end
```

A Longer Example

What about a piece trying to move after Checkmate, when the game ends?

```
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
```

A Longer Example

What about a piece trying to move after Checkmate, when the game ends?

```
-- 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
```

A Longer Example cont.

What about if the White Queen tries to move through another piece, mid-game?

```
game = chess
    pawn e2 to e4
    pawn f7 to f5
    queen d1 to d3   -- Invalid move
    pawn g7 to g5
    queen f3 to h5
end
```

A Longer Example cont.

What about if the White Queen tries to move through another piece, mid-game?

```
-- 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  -- Invalid move
    pawn g7 to g5
    queen f3 to h5
end
```

A Longer Short Example

We also developed a shorthand syntax!

A Longer Short Example

We also developed a shorthand syntax!

The below game:

```
game = chess
    pawn e2 to e4
    pawn f7 to f5
    queen d1 to f3
    pawn g7 to g5
    queen f3 to h5
end
```

A Longer Short Example

We also developed a shorthand syntax!

The below game:

```
game = chess
    pawn e2 to e4
    pawn f7 to f5
    queen d1 to f3
    pawn g7 to g5
    queen f3 to h5
end
```

becomes:

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

Testing

Combination of:

Testing

Combination of:

- Unit testing with assertions, based on whether a code snippet compiles or fails to compile;

Testing

Combination of:

- Unit testing with assertions, based on whether a code snippet compiles or fails to compile;
- EDSL testing with famous Chess games, written out in Chesskell notation.

Compile-time and memory issues

Compile-time and memory issues came up time and again throughout development; putting a hard limit on the length of Chesskell games.

With some games, GHC will run out of memory (>25GB) and crash.

Through testing, it seems the upper limit is **12 moves maximum**; some 12-move games tested compile, most 10-move games compile, and all 8-move games compile.

Further Work

Further Work

- More domains could be modelled at the type level, not just Chess;

Further Work

- More domains could be modelled at the type level, not just Chess;
- A session-typed version of Chesskell;

Further Work

- More domains could be modelled at the type level, not just Chess;
- A session-typed version of Chesskell;
- Further optimisations to try and increase the move limit.

Conclusions

We have created:

Conclusions

We have created:

- A full type-level model of Chess, which enforces all rules in the FIDE 2018 Laws of Chess;

Conclusions

We have created:

- A full type-level model of Chess, which enforces all rules in the FIDE 2018 Laws of Chess;
- An EDSL for describing Chess games and creating custom chess boards, which uses the type-level model for rule-checking;

Conclusions

We have created:

- A full type-level model of Chess, which enforces all rules in the FIDE 2018 Laws of Chess;
- An EDSL for describing Chess games and creating custom chess boards, which uses the type-level model for rule-checking;
- Interesting findings on compile-time and memory usage issues.

Conclusions

We have created:

- A full type-level model of Chess, which enforces all rules in the FIDE 2018 Laws of Chess;
- An EDSL for describing Chess games and creating custom chess boards, which uses the type-level model for rule-checking;
- Interesting findings on compile-time and memory usage issues.

Furthermore, Chesskell is unique and has never been done before. Though there is room for further work and improvement, Chesskell is a success!

References i

L. Cardelli, "Type systems," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 263–264, 1996.

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

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.

- 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.
- 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.
- 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.

References iii

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn, "Simple unification-based type inference for GADTs," *ACM SIGPLAN Notices*, vol. 41, no. 9, pp. 50–61, 2006.

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

Demo

Q&A