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

Toby Bailey

March 4, 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.

(Cardelli, "Type systems")

Type Errors

A type system can prevent certain errors from occurring at all:

```
not 5
```

The above will not compile, preventing an error.

Type Errors cont.

You have a website, where you sell books.

Type Errors cont.

You have a website, where you sell books.

For some reason, you use Java to build the server:

```java
int noOfPages = -1;
```

This is obviously an error. But it compiles!

Type-Level Programming

Recent developments to Haskell have focused on performing computation at the type level with *type families* (Schrijvers et al., “Towards open type functions for Haskell”, Eisenberg, Vytiniotis, et al., “Closed type families with overlapping equations”).

Haskell is NOT a dependently typed language; types and values are separated.

Type Erasure

In fact, Haskell programs undergo *type erasure*.

Type Erasure

In fact, Haskell programs undergo *type erasure*.

```haskell
x :: Int
x = 3
```

Haskell type-level programming involves circumventing type erasure.

Complex Type-Level Computation

There are other attempts at rule enforcement, in Haskell, at the type level.

Complex Type-Level Computation

There are other attempts at rule enforcement, in Haskell, at the type level.

Mezzo is an EDSL for musical composition, which can enforce classical harmony rules at the type level (Szamozvancev and Gale, "Well-typed music does not sound wrong (experience report)").

Complex Type-Level Computation

There are other attempts at rule enforcement, in Haskell, at the type level.

Mezzo is an EDSL for musical composition, which can enforce classical harmony rules at the type level (Szamozvancev and Gale, "Well-typed music does not sound wrong (experience report)").

BioShake is an EDSL for Bioinformatics workflows, whereby only workflows that are configured correctly will compile (Bedő, "BioShake: a Haskell EDSL for bioinformatics workflows").

Why create Chesskell?

These capabilities are intended for use; and so must be tested through use.

What issues do we run into when implementing a complex rule set at the type level?

Why create Chesskell?

These capabilities are intended for use; and so must be tested through use.

What issues do we run into when implementing a complex rule set at the type level?

Is Haskell's type system mature enough for Chess?

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 (Gusev, "Using Chess Programming in Computer Education.", Block et al., "Using reinforcement learning in chess engines");

Why Chess?

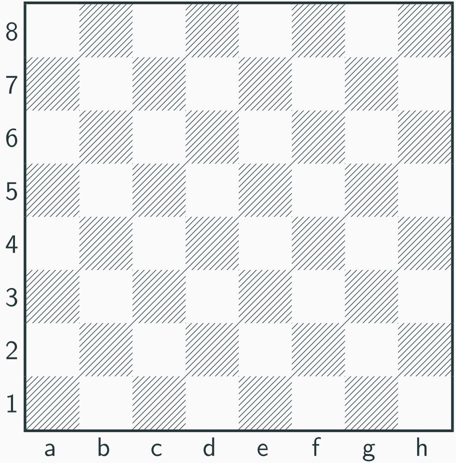
- It's popular and internationally known;
- It's been widely studied in the field of Computer Science (Gusev, "Using Chess Programming in Computer Education.", Block et al., "Using reinforcement learning in chess engines");
- It has a *well-defined ruleset*.

Why Chess?

- It has a *well-defined ruleset*.

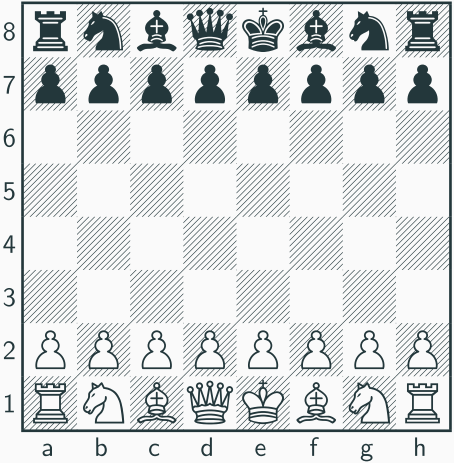
A note on Chess

A Chess game takes place on a board.



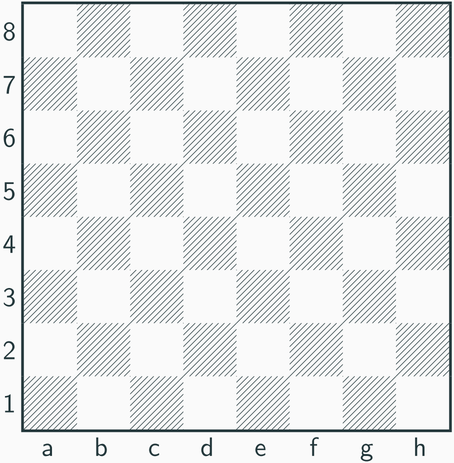
A note on Chess cont.

There are two *Teams*; Black and White.



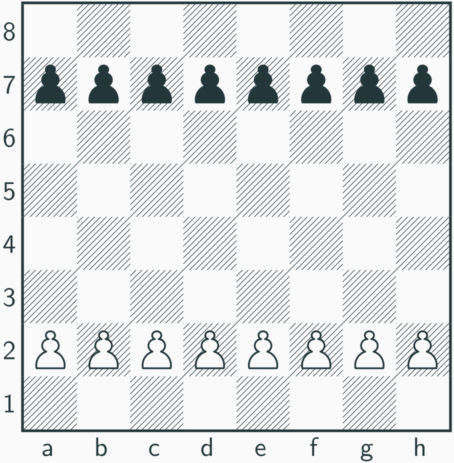
A note on Chess cont.

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



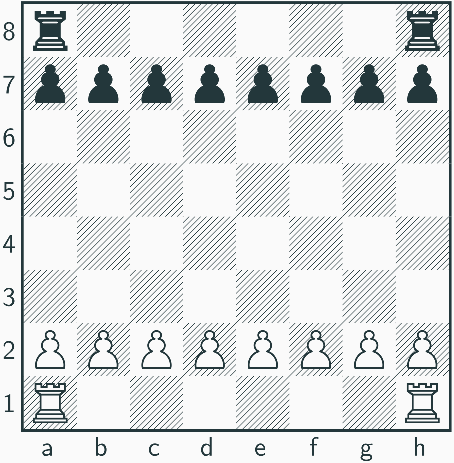
A note on Chess cont.

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



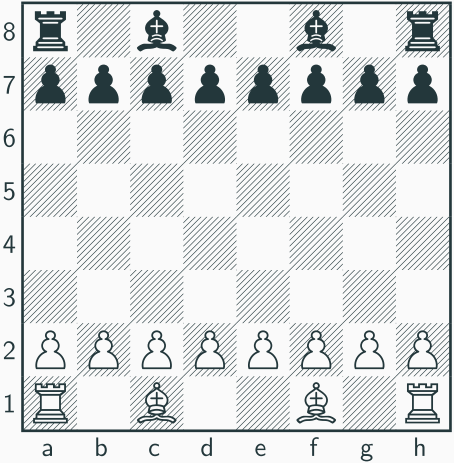
A note on Chess cont.

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



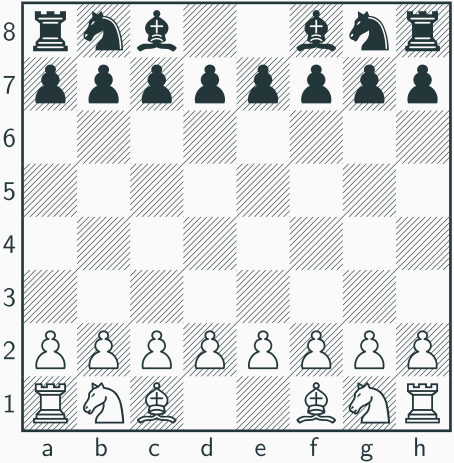
A note on Chess cont.

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



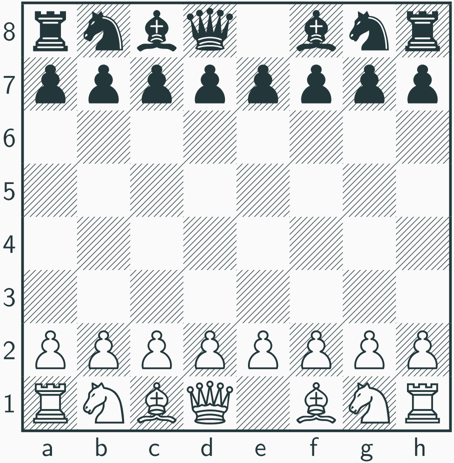
A note on Chess cont.

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



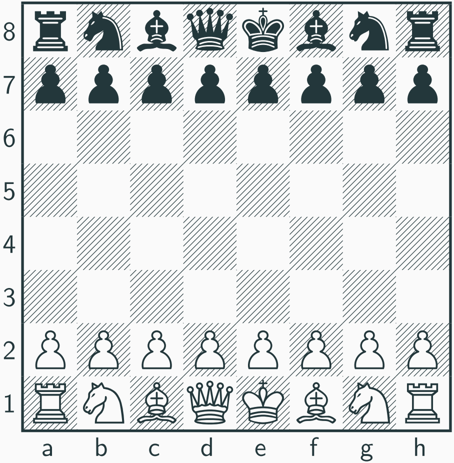
A note on Chess cont.

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



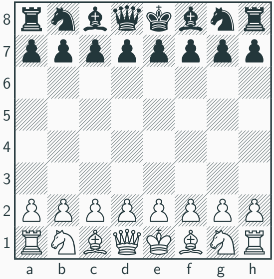
A note on Chess cont.

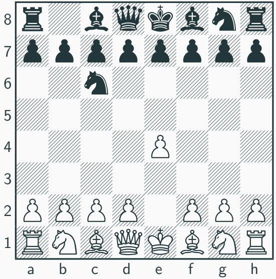
Each Team has 16 *Pieces*; 8 Pawns, 2 Rooks, 2 Bishops, 2 Knights, a Queen, and a King.



A note on Chess cont.

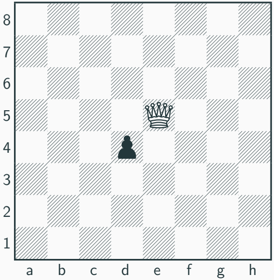
Each piece has different movement rules, allowing them to move around the 8x8 board.

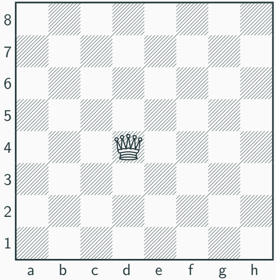




A note on Chess cont.

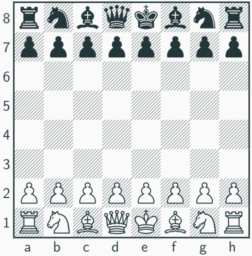
Pieces can remove other pieces from the board via *capture*; which almost always involves moving to the other piece's square.

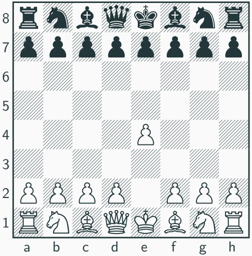




A Short Example

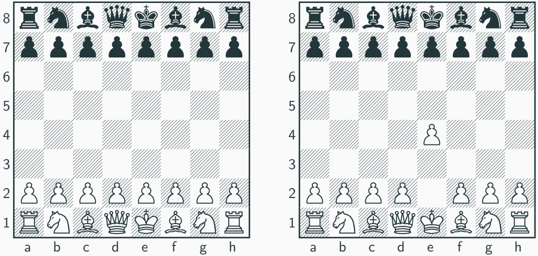
Below is a valid move by a White Pawn:





A Short Example

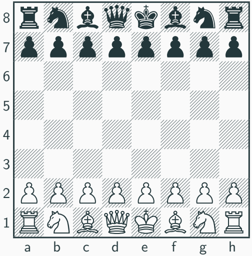
Below is a valid move by a White Pawn:

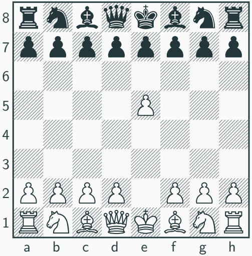



```
chess
    pawn e2 to e4
end
```

A Short Example cont.

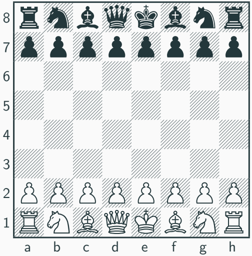
Below is an *invalid* move by a White Pawn:

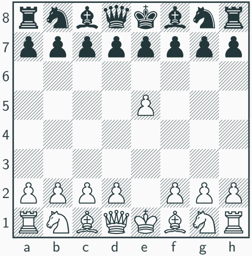




A Short Example cont.

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

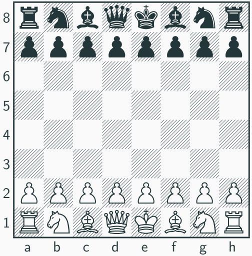


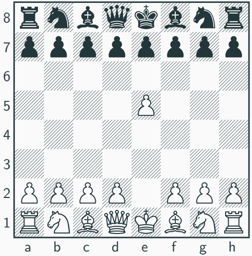


```
chess
    pawn e2 to e5
end
```

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 (Yorgey et al., "Giving Haskell a promotion").

A Little Terminology

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

Luckily, we can *promote* types to kinds with the `-XDataKinds` extension (Yorgey et al., “Giving Haskell a promotion”).

```haskell
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*.

```haskell
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)
```

But you have to use type families to compute on types:

```
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]
-- = [3,4,5]
```

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]
-- = [3,4,5]
```

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
```

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
```

Type-Level Mapping

With the below definition of `Map`:

```
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)
```

And a definition of a type-level `(+)`:

```
data (:+) :: Nat -> Nat -> Exp Nat
type instance Eval (Z     :+ y) = y
type instance Eval ((S x) :+ y) = S (x :+ y)
```

We can now map over a type-level list:

```
Eval (Map (:+ 2) '[1,2,3])
-- = '[3,4,5]
```

Representing Movement

Each turn of movement is expressed as a single First Class Family:

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

Representing Movement

Each turn of 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
```

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
```

In the codebase, we use a wrapper data structure (named `BoardDecorator`) to hold additional useful information.

Using the Type-Level Model

To interact with this type level model, the output of each `Move` call is piped to the next one:

```
x = Move a1 a2 StartBoard
y = Move e3 e4 x
z = -- ...
```

Using the Type-Level Model cont.

Below is a simplified representation of what happens for the game:
`chess pawn a1 to a2 king e2 to e1 end`

```
(MoveWithCheck King e2 e1 . MoveWithCheck Pawn a1 a2)
    StartBoard
```

Using the Type-Level Model cont.

Below is a simplified representation of what happens for the game:

`chess pawn a1 to a2 king e2 to e1 end`

```
(MoveWithCheck King e2 e1 . MoveWithCheck Pawn a1 a2)
    StartBoard

data MoveWithCheck :: PieceName -> Position ->
    Position -> Exp Board
type instance Eval (MoveWithCheck name fromPos toPos
    board)
     -- If there is a piece of that type at fromPos
     = If (IsPieceAt name fromPos board)
         -- then
         (Move fromPos toPos board)
         -- else
         (TypeError -- ...)
```

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
```

Can achieve using Continuation Passing Style, inspired by Dima Szamozvancev's Flat Builders work (Szamozvancev, "Well-typed music does not sound wrong").

Chess Continuations

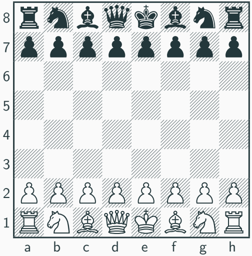
We define a continuation for beginning the stream (named `chess`), another for ending it (`end`), and a series of piece continuations that move a piece of that type; `king`, `queen`, etc.

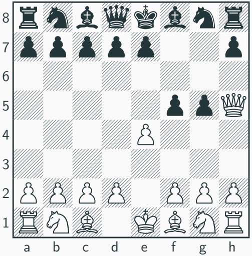
All of the above continuations can be chained together like so:

```
game = chess pawn a1 to a2 bishop e4 to d5 end
```

A Longer Example

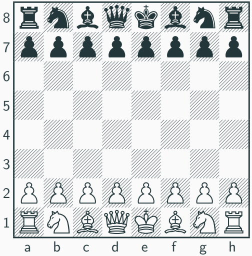
Below is a short game, ending in checkmate by White:

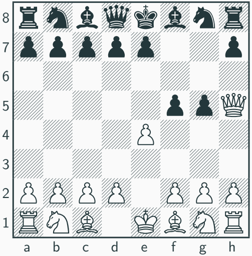




A Longer Example

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!

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
```

Demo

Testing Overview

Combination of:

Testing Overview

Combination of:

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

Testing Overview

Combination of:

- Unit testing with assertions, based on whether a code snippet compiles or fails to compile;
- Unit tests of custom-made board scenarios, to test out specific behaviour;

Testing Overview

Combination of:

- Unit testing with assertions, based on whether a code snippet compiles or fails to compile;
- Unit tests of custom-made board scenarios, to test out specific behaviour;
- EDSL tests of custom board scenarios, for the same purpose;

Testing Overview

Combination of:

- Unit testing with assertions, based on whether a code snippet compiles or fails to compile;
- Unit tests of custom-made board scenarios, to test out specific behaviour;
- EDSL tests of custom board scenarios, for the same purpose;
- EDSL testing with famous Chess games, written out in Chesskell notation.

Unit Testing

Unit tests rely on two main assertions; `shouldTypecheck`, and `shouldNotTypecheck`, which succeed or fail based on whether a specific code snippet fails with a type error or not.

We created unit tests for individual type families, to determine if they have the behaviour they should:

```
oppositeTeamTest1 :: White :~: Eval (OppositeTeam
    Black)
oppositeTeamTest1 = Refl

-- ...

it "1: OppositeTeam Black = White" $
    shouldTypecheck oppositeTeamTest1
```

(Note that a value with type `x :~: y` will only compile if `x` and `y` can be unified.)

Unit Testing cont.

We also created unit tests for every FIDE Law of Chess that could be tested in this manner:

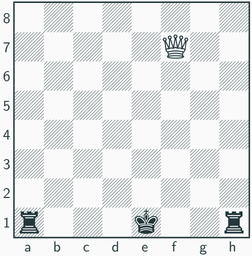
```
whiteBishopCannotTakeOwnTeam :: Proxy (a ::
    BoardDecorator)
whiteBishopCannotTakeOwnTeam = Proxy @(Eval (Move (At
    C Nat1) (At D Nat2) WhiteStartDec))

-- ...

it "1: A White Bishop cannot take a piece on the same
    team" $
     shouldNotTypecheck whiteBishopCannotTakeOwnTeam
```

Scenario Testing

We created custom Chess test boards, paired with unit tests, to model specific behaviour:




```
blackCanCastleLeft :: '(True, False) :~: CanCastle
    Black BlackCastleLeftDec
blackCanCastleLeft = Refl
-- ...
shouldTypecheck blackCanCastleLeft
```

EDSL Scenario

The EDSL was similarly tested with scenarios, to ensure that rule-breaking moves did not compile:

```
didntPromoteBlack = create
        put _Wh _P at h7
        put _Bl _P at a2
    startMoves
        pawn h7 promoteTo _B h8
        pawn a2 to a1
    end
```

EDSL Scenario

The EDSL was similarly tested with scenarios, to ensure that rule-breaking moves did not compile:

```
-- Below fails with the following type error:
    -- * Promotion should have occurred at: a1. Pawns
    must be promoted when they reach the opposite end
    of the board.
    -- * When checking the inferred type:
    --      didntPromoteBlack :: Data.Proxy.Proxy
    (TypeError ...)
didntPromoteBlack = create
        put _Wh _P at h7
        put _Bl _P at a2
    startMoves
        pawn h7 promoteTo _B h8
        pawn a2 to a1
    end
```

EDSL Game Testing

Testing for EDSL correctness primarily consisted of writing out the first few moves of some famous game, and then making variations with errors:

```
loopVsGandalf = chess
      p e4 p c5
      n f3 p d6
      p d4 p d4
      n d4 n f6
      n c3 p a6
   end
```

EDSL Game Testing

Testing for EDSL correctness primarily consisted of writing out the first few moves of some famous game, and then making variations with errors:

```
loopVsGandalfError = chess
        p e4 p c5
        n f3 p d6
        p d4 p d4
        n d4 n f6
        n c3 p a7
    end
```

EDSL Game Testing

Testing for EDSL correctness primarily consisted of writing out the first few moves of some famous game, and then making variations with errors:

```
loopVsGandalfError = chess
      p e4 p c5
      n f3 p d6
      p d4 p d4
      n d4 n f6
      n c3 p a7   -- Pawn moves to same place!
   end
```

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**; while all 6-move games tested have compiled, most 8- and 10-move games do as well.

Compile-time and memory issues cont.

In fact, we discovered a difference in behaviour between type applications (Eisenberg, Weirich, and Ahmed, "Visible type application") and type signatures:

```haskell
-- Compiles, but would hang
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy @StartDec)

-- Would fail to compile
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy :: Proxy StartDec)
```

Compile-time and memory issues cont.

In fact, we discovered a difference in behaviour between type applications (Eisenberg, Weirich, and Ahmed, "Visible type application") and type signatures:

```
-- Compiles, but would hang
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy @StartDec)

-- Would fail to compile
chess :: Spec (Proxy StartDec)
chess cont = cont (Proxy :: Proxy StartDec)
```

So we filed a GHC bug report:
`https://gitlab.haskell.org/ghc/ghc/-/issues/18902`

Project Management

The project was managed successfully, making use of;

Project Management

The project was managed successfully, making use of;

- Spiral methodology;

Project Management

The project was managed successfully, making use of;

- Spiral methodology;
- Git and GitHub for version control;

Project Management

The project was managed successfully, making use of;

- Spiral methodology;
- Git and GitHub for version control;
- Weekly supervisor meetings;

Project Management

The project was managed successfully, making use of;

- Spiral methodology;
- Git and GitHub for version control;
- Weekly supervisor meetings;
- A Trello board to track upcoming tasks and completed features (Figure 1);

Project Management

The project was managed successfully, making use of;

- Spiral methodology;
- Git and GitHub for version control;
- Weekly supervisor meetings;
- A Trello board to track upcoming tasks and completed features (Figure 1);
- Extensive unit and integration testing.

Project Management

The project was managed successfully, making use of;

- Spiral methodology;
- Git and GitHub for version control;
- Weekly supervisor meetings;
- A Trello board to track upcoming tasks and completed features (Figure 1);
- Extensive unit and integration testing.

The implementation of Chesskell was feature-complete by the 4th of December; since then, work has gone into optimisation, testing, and write-ups.

Project Management cont.

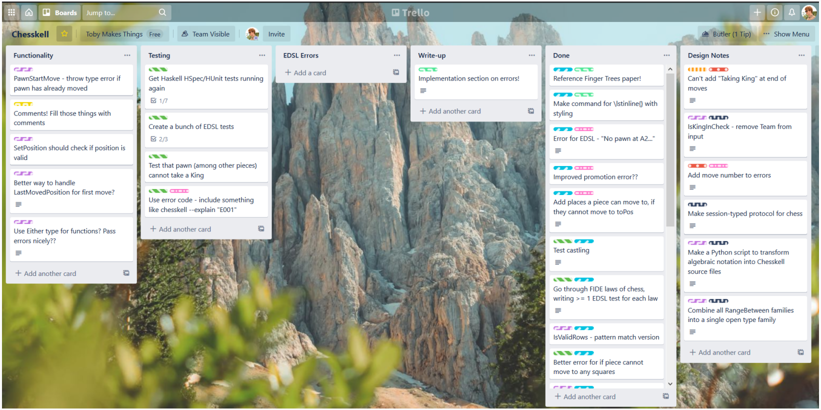


Figure 1: The Trello board used to track development.

Further Work

There is room to expand Chesskell:

Further Work

There is room to expand Chesskell:

- A session-typed version of Chesskell;

Further Work

There is room to expand Chesskell:

- A session-typed version of Chesskell;
- Further optimisations to try and increase the move limit;

Further Work

There is room to expand Chesskell:

- A session-typed version of Chesskell;
- Further optimisations to try and increase the move limit;
- An automated tool to transform from Algebraic Notation into Chesskell notation.

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;
- A first draft of a Haskell Symposium paper about the development of Chesskell, including our 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;
- A first draft of a Haskell Symposium paper about the development of Chesskell, including our 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!

Q&A

References

- Justin Bedő. "BioShake: a Haskell EDSL for bioinformatics workflows". In: *PeerJ* 7 (2019), e7223.
- Marco Block et al. "Using reinforcement learning in chess engines". In: *Research in Computing Science* 35 (2008), pp. 31–40.
- Luca Cardelli. "Type systems". In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 263–264.

Richard A Eisenberg, Dimitrios Vytiniotis, et al. "Closed type families with overlapping equations". In: *ACM SIGPLAN Notices* 49.1 (2014), pp. 671–683.

Richard A Eisenberg, Stephanie Weirich, and Hamidhasan G Ahmed. "Visible type application". In: *European Symposium on Programming*. Springer. 2016, pp. 229–254.

Dmitri A Gusev. "Using Chess Programming in Computer Education.". In: *Association Supporting Computer Users in Education* (2018).

References iii

Tom Schrijvers et al. “Towards open type functions for Haskell”. In: *Implementation and Application of Functional Languages* 12 (2007), pp. 233–251.

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

Dmitrij Szamozvancev. “Well-typed music does not sound wrong”. In: (2017).

References iv

Brent A Yorgey et al. “Giving Haskell a promotion”. In: *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. 2012, pp. 53–66.