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

Toby Bailey

February 4, 2021

Abstract

Type-level programming, a relatively recent phenomenon, allows programmers to express computation during the compilation of their programs. Through the use of type-level constructs, rules can be imposed on code to ensure that if it compiles, then it behaves in a certain way. However, there is still plenty of room to push the boundaries of what can be achieved with type-level programming.

Chess has a well-defined ruleset, and has not been expressed at the type level before. This dissertation describes the development of Chesskell, a Haskell-Embedded Domain-Specific Language to notate Chess games within. If the Chesskell code compiles, then the match described obeys the full International Chess Federation ruleset for Chess. Despite difficulties during development, including memory issues, the final version of Chesskell is feature-complete and supports Chess games of up to 10 moves.

Keywords: Type-level Programming, Haskell, Chess, EDSL.

Contents

Abstract 2

Contents 3

1 Introduction 5
- 1.1 Related Work . . . 7
  - 1.1.1 Type-level Rule Checking . . . 7
  - 1.1.2 Haskell-Embedded Domain-Specific Languages . . . 8
  - 1.1.3 Chess in Computer Science . . . 9
  - 1.1.4 Why Haskell? . . . 9
- 1.2 Objectives . . . 10

2 Background 11
- 2.1 Types in Haskell . . . 11
- 2.2 Type-level Programming . . . 13
  - 2.2.1 Kind Promotion . . . 13
  - 2.2.2 Type Families . . . 14
  - 2.2.3 First-Class Families . . . 15
  - 2.2.4 Proxies and Singletons . . . 16

3 Design 17
- 3.1 The Basics of Chess . . . 17
  - 3.1.1 The Board . . . 17
  - 3.1.2 The Game . . . 17
  - 3.1.3 Chess Notation . . . 17
- 3.2 Type-Level Data Structures . . . 17
  - 3.2.1 Chess Data Structures . . . 17

4 Implementation 18
- 4.1 Type-Level Chess . . . 18
  - 4.1.1 Chess Types and Kinds . . . 18
  - 4.1.2 Chess Rules . . . 18
- 4.2 The EDSL . . . 18
  - 4.2.1 Minimum Viable Product . . . 18
  - 4.2.2 Flat Builders . . . 18
  - 4.2.3 Moving the pieces . . . 18

4.2.4 Setting up a board 18

5 Evaluation 19
5.1 Testing 19
5.1.1 Type-level Unit Testing 19
5.1.2 Testing Chesskell Games 19
5.2 Compile Time and Memory Usage 19
5.2.1 Optimisation Attempts 19
5.2.2 GHC Bug Report 19
5.2.3 Descriptive Error Messages 19
5.3 Chesskell EDSL vs Other Chess Notations 19

6 Conclusions 20
6.1 Results and Accomplishments 20
6.2 Future Work 20
6.2.1 Session-typed Chesskell 20

7 Bibliography 21

8 Appendix 23

1 Introduction

The study of programming languages in Computer Science involves, in large part, the study of type systems. Many of the interesting differences between programming languages lie not in their syntax, but in their semantics; in their behaviour. Since types govern the behaviour of languages, it is fair to say that the difference in type systems between languages forms the basis of what individuals like or dislike about programming in a specific language. Part of why assembly language can be so difficult to reason about at scale is because it is untyped; everything is a byte. C, and other higher-level languages, have been introduced for the programmer's benefit. With higher levels of abstraction, and more complex type systems, can come more safety, as well as clearer program behaviour.

Programming languages have *type systems* for the main purpose of avoiding errors [1]. A *type error* is an instance of attempting to perform a computation on something which does not support that computation. For example, it makes no logical sense to add the number 3 to a dog. This stems from the fact that "3" and "dog" support different behaviours[1]. Therefore, in a programming context, "3" and "dog" have distinct types; 3 is a number, and a dog is an animal. By assigning a type to values, programmers and the languages they use have an easier way to determine the valid operations on a value, and avoid type errors through misuse.

A notable area in which languages differ is *when* they detect type errors. A *static* type system is one in which type errors are detected before the program is run (during compilation), and a *dynamic* type system is one in which type errors are detected while the program is running. A static type system is preferable for runtime safety, since it ensures that any running program will avoid (at least some) type errors.

Of course, as programming languages evolve, many have begun to address more and more errors at compile time (i.e. through the type system). Features similar to optional types have been added to languages such as Java[2] and C#[3], and languages like Rust have pioneered ways of safely handling dynamic allocation through ownership types[4]. Many compilers now force the developer to handle classes of errors that previously could only be encountered at runtime, such as null pointer exceptions.

[1] For instance, dogs can bark, but the number 3 cannot.
[2] `https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html`
[3] `https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references`
[4] `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`

However, one type of error that typically evades the type system is a *logical error*—some (typically domain-specific) behaviour that is not guarded against by a language's type system. Catching logical errors in imperative languages is almost always done during execution. Many software systems use runtime features such as exceptions to discover and deal with errors and misuse of APIs. Enforcement of invariants and rules is typically dynamic; if a check fails, an exception is thrown and potentially handled. However, if a programmer forgets to implement such a check, the behaviour is unpredictable. A 2007 study [2] on Java and .NET codebases indicates that exceptions are rarely used to recover from errors, and a 2016 analysis of Java codebases [3] reveals that exceptions are commonly misused in Java.

Recent versions of the *Glasgow Haskell Compiler* (GHC) support programming at the type level, allowing programmers to compute with types in the same way that languages like C or Python compute with values [4], using *type families* [5] [6] that emulate functions at the type-level. These computations run at compile time, before the compiler generates an executable of the source code, allowing programmers to transform logic errors into type errors [7]. The exception misuse described above could be avoided by employing logical invariant checks at the type-level, rather than at runtime.

Since these are relatively recent developments, there are few examples of their usage in complex applications. It is worth pushing the boundaries of existing type systems, and seeing what kind of logical behaviour can be modelled (and enforced) through type-level checks. In this project, we show how to utilise type-level programming features in Haskell in order to model the classic board game Chess in Haskell's type system, ruling out invalid moves at the type-level. A Haskell-Embedded Domain-Specific Language (DSL), for describing games of Chess, will interact with the type-level model. This Embedded DSL (EDSL) will be modelled on Algebraic Notation, a method of writing down the moves associated with a particular match of Chess. We implement the full, official International Chess Federation (FIDE) ruleset for Chess.

A growing number of new languages have type systems which support *Dependent types*, in which the types themselves depend on runtime values, and can be treated as values. The programming language Idris is similar to Haskell, but allows the programmer to pass around types at runtime, and write functions which operate on those types. Many of Haskell's language extensions have been adding to its type system, moving the language closer and closer towards dependently typed programming [8]. Such a type system has various benefits, since constraining the types means constraining the values without dynamic runtime checks. (For example, in a dependently typed environment, runtime array bounds checks can be eliminated at runtime through being expressed solely in the type system [9].)

Chess was chosen as a suitable game to model at the type-level due to its well-defined ruleset. Programming language type systems will evolve through usage, and so programs will and should be written to test what's possible to express at the type level. Chess is a widely understood, popular, and rigorously documented game, making it a natural fit to help push the boundaries of type-level programming. Simulating, and checking for rule violations within, a Chess game has a much wider scope than using type-level programming to avoid some dynamic checks. This project uses Chess as a case study for complex rule systems, to determine if such a thing can be modelled at the type level.

1.1 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[5]. The N-queens problem makes use of some chess rules, including the Queen's attack positions[6]; 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.

1.1.1 Type-level Rule Checking

The idea of using types to enforce rules on behaviour is hardly specific to Haskell; C and C-like languages ensure that you only apply the correct operations on types, after all. The programming language Rust[7] has been voted the most loved language (by StackOverflow developers) 5 years running[8]. Rust is touted as a systems language that guarantees memory safety and thread safety; and it achieves this through its type system. By enforcing strict ownership rules, Rust can guarantee that your programs avoid data races and that all memory is freed once and not used after being freed. This is a clear example of types enforcing runtime behaviour; but instead of Chess rules, a series of memory rules are being enforced. In fact, Haskell type-level constructs can be used to enforce basic ownership rules through a method colloquially known as the "ST Trick" [7].

An example of type-level rule checking in Haskell is a specific implementation of merge sort [10]. Through clever use of types, Lindley and McBride's merge sort implementation is guaranteed to produce sorted outputs. Unit tests for the sorting implementation become unnecessary, since the GHC type checker is used to ensure that

[5]`https://github.com/ExNexu/nqueens-idris`
[6]A Queen can attack in a straight line in any direction.
[7]`https://www.rust-lang.org/`
[8]`https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wa`

the sort itself behaves correctly. The type system is used to enforce the rule that sorted data should be in sort order.

The above examples may seem simple when compared to the ruleset of Chess, but they demonstrate the fact that type-level behaviour enforcement is neither new nor specific to Haskell. Though type systems can be complex, since many languages are designed to be general-purpose their type systems are also designed to be so. Chesskell represents an attempt at capturing domain-specific knowledge at the type level, and using that knowledge to maintain safe behaviour. Chesskell, and other type-level behaviour checkers, are not common simply because logic errors are usually dealt with through dynamic checks (it is certainly easier to write dynamic unit tests than it is to model your application domain with types).

1.1.2 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 valid programs written in that language will not exhibit invalid behaviour, then the EDSL becomes an attractive way to interact with that domain.

Mezzo [11] 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.

1.1.3 Chess in Computer Science

Chess has a rich history as a study area of Computer Science. Getting computers to play Chess was a concern back in 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[9,10]. Many of them 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; it simply takes in the moves performed, and state whether they are valid chess moves or not. We are not aware of any such type-level chess implementations in Haskell, or any other language.

Game development, as a more general field in Computer Science, has many Chess-based or Chess-related games available. However, the intention in these cases is usually to facilitate real-time play between multiple players (or indeed a single player with a competitive AI), rather than to teach or program a machine to consistently beat players. There is overlap with Chesskell; Chess as a computer game must necessarily perform move validation (to disallow cheating) and ensure that players take turns. However, Chesskell is intended to check over a complete game, rather than to enable people to conduct a game in real-time with Chesskell as a mediator.

1.1.4 Why Haskell?

Given the previous discussion on dependent type systems, and how Haskell is inching towards one, it begs the question; why not use a dependently typed language, like Idris, to write Chesskell in? The simple answer is because it would be trivial. Writing type-level code in Idris (or any other dependently typed language) would be near indistinguishable from writing a chess validity checker, bundled within an EDSL, at the term-level. Such a feat is both simple and unoriginal.

However, choosing to write Chesskell in Haskell means figuring out how to perform term-level computation at the type-level. Indeed, the majority of the code for Chesskell is reusable, since much of it is not specifically about expressing the rules of Chess, but building components to enable complex computation with types. The project is both more difficult, and more interesting, for having been completed in a language without a full dependent type system.

[9] `https://github.com/mlang/chessIO`

[10] `https://github.com/nionita/Barbarossa`

1.2 Objectives

The objective of this project is to develop a model of Chess at the type level, which will compile a given program if and only if it is a valid game of chess. The user-facing method of working with this type level model will be via a custom EDSL, through which Chess games are expressed. During compilation, the game of Chess will be simulated, such that any invalid move (or the lack of a move where one should have occurred) will result in a type error. The main goals are thus:

- Develop a type-level model of a Chess board;
- Develop a type-level move-wise model of a Chess game;
- Develop an EDSL to express these type-level Chess games in;
- Ensure (through testing) that valid Chess games compile, and invalid Chess games do not.

During the course of the project, a "valid Chess game" is any game that adheres to the FIDE 2018 Laws of Chess[11]. The FIDE laws also contain rules for the players themselves to adhere to; but these are outside the scope of the project, since they are not directly concerning the game of Chess itself.

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

2 Background

Haskell is a pure functional language, based upon lambda calculus with the addition of a type system[1]. All Haskell expressions have a type, which is inferred at compile time. Programs go through *type erasure*—the executable output of the compiler has no notion of types. In other words, Haskell has a clear separation between dynamic values, and the static types those values are members of.

As such, passing information between the type system and the runtime values (as is common in dependently typed languages) is made more difficult. This section summarises notable aspects of Haskell's type system, explaining how programmers can use Haskell extensions and advanced features to perform computation at the type level.

A short note: Haskell has an interactive Read-Eval-Print-Loop interpreter, named GHCi. The below section contains several GHCi snippets, to aid in understanding. `$>` represents a terminal input line, and anything following it can be assumed to have been typed into GHCi. A programmer can query the type of a value with `:t`. So, `$> :t True` denotes an attempt to discover the type of the value `True`, which is of course `Bool`. Additionally, `::` means "has type", so `3 :: Int` reads as "the value `3` has type `Int`".

2.1 Types in Haskell

Programs written in Haskell make use of *types* and *terms* (colloquially known as values). That is, a type can be viewed as a (potentially infinite) set of valid terms, and each term has a type. The primary Haskell compiler, GHC, is responsible for inferring types, and for generating type errors when a given term does not match its recorded or inferred type.

For instance, as in many other languages, `3` has type `Int`. It would be incorrect to declare an expression of a different type, and to give it a value of `3`, as below:

```
x :: Bool
x = 3  -- error: Couldn't match expected type 'Bool' with actual type
   'Int'
```

[1] `https://www.haskell.org/`

However, Haskell has support for *polymorphism*; firstly, ad-hoc polymorphism, whereby functions can be specialised to operate on specific types, with separate definitions for each type.

Secondly, parametric polymorphism, where a value's type is dependent on one or more *type variables*. Consider the list type; it would be nonsensical to define an entire new list data type for each potential inhabitant. As such, the list type in Haskell is more general, in that it can hold any value with a type that can be unified with a type variable:

```
$> :t [True, False]
[True, False] :: [Bool]
$> :t []
[] :: [a]
```

In Haskell, type variables are typically named alphabetically; so the first general type in an type annotation is `a`, followed by `b`, `c`, and so on. In fact, all Haskell types start with a capital letter, so any lowercase string is a valid type variable name.

Haskell functions all have type `a -> b`, where the type variables `a` and `b` extend to more types, including other function types. The rightmost type in a function definition is the return type. For instance, Boolean logical AND would have the type `Bool -> Bool -> Bool`, while logical NOT would have the type `Bool -> Bool`. Haskell functions can be partially applied; the below code snippet assumes the definition of a function `and`, which performs logical AND on its two inputs:

```
$> :t and
and :: Bool -> Bool -> Bool
$> :t and True
and True :: Bool -> Bool
$> and True False
and True False :: Bool
```

Haskell allows the programmer to define their own data type with the keyword `data`. These data types are *algebraic*, meaning that it is a type comprised of other types. For instance, to define a "Hand" type, where someone can hold something on each finger, the definition would be something like as follows:

```
data Hand a = One a
            | Two a a
            | Three a a a
            | Four a a a a
            | Five a a a a a
```

However, this definition syntax has limitations; all of the return values of the type constructors above must be `Hand a`. A Haskell extension allows the definition of *Generalised Algebraic Data Types* (GADTs) which allows more complex type constructor definitions [18]. The above `Hand` datatype could be expressed thus:

```haskell
data Hand a where
    One   :: a -> Hand a
    Two   :: a -> a -> Hand a
    Three :: a -> a -> a -> Hand a
    Four  :: a -> a -> a -> a -> Hand a
    Five  :: a -> a -> a -> a -> a -> Hand a
```

Furthermore, if you wished to modify `Hand` to ensure that it always stored `Int` values on odd fingers, and `Bool` values on even fingers, you can achieve that with GADTs like so:

```haskell
data Hand a where
    One   :: Int  -> Hand Int
    Two   :: Bool -> Int  -> Hand Bool
    Three :: Int  -> Bool -> Int  -> Hand Int
    Four  :: Bool -> Int  -> Bool -> Int  -> Hand Bool
    Five  :: Int  -> Bool -> Int  -> Bool -> Int  -> Hand Int
```

2.2 Type-level Programming

While the above is certainly useful in day-to-day programming, it is not enough to achieve dependent types. Luckily, there are many more Haskell extensions, a large number of which bring the language closer to dependent types.

2.2.1 Kind Promotion

A key concept in type-level programming in Haskell is that of *promotion*. The data types that programmers define (as we explain above) can be promoted to *kinds*. Kinds are, conceptually, the types of types; that is, terms have types, and types have kinds. A type of kind `*` takes no type variables, and a type of kind `* -> * -> *` takes in two type variables and returns a type. Consider an empty list, which takes a type variable; it has kind `* -> *`, which can be verified using GHCi with the `:k` directive, or the `:kind!` directive which additionally evaluates the type to a normal form:

```
$> :k []
[] :: * -> *
$> :k [Int]
[Int] :: *
```

The kind `*` is commonly aliased as `Type`, since it is the kind of types which have runtime values. That distinction becomes important when promotion is involved; programmers can define their own kinds with the `-XDataKinds` extension enabled. Consider a custom `Book` data type, which is either `Fiction` or `NonFiction`. A type definition may look as follows:

```haskell
data Book = Fiction | NonFiction
-- With GADT syntax
data Book where
    Fiction :: Book
    NonFiction :: Book
```

With the `-XDataKinds` extension enabled, the above code not only produces the two values `Fiction` and `NonFiction` with type `Book`, but also the *types* `'Fiction` and `'NonFiction`, of kind `Book`. The key point of understanding is that there are no values of type `'Fiction` or `'NonFiction`—they exist solely at the type level.

The syntax for "has type" and "has kind" is in both cases `::`, which is unfortunate; however, in the rest of the document, where the distinction is unclear, it shall be made so. Additionally, the prefix `'` for promoted types is optional, and is left out where the compiler can unambiguously state whether an expression should be a type or a value.

2.2.2 Type Families

Another key extension introduces *type families*. Type families allow the programmer to compute over types just as functions compute over values; they are the type-level analogue to functions, and come with their own syntax. Following on from the `Book` example above, consider a type family `IsFiction`, which states whether a given `Book` is fiction or not. A term-level definition could be as follows:

```haskell
isFiction :: Book -> Bool
isFiction Fiction    = True
isFiction NonFiction = False
```

And the type family analogue is thus, where `::` below means "has kind":

```haskell
type family IsFiction (x :: Book) :: Bool where
    IsFiction 'Fiction    = True
    IsFiction 'NonFiction = False
```

Both function and family use pattern-matching, and although the type family syntax is a little more verbose, it is still clear. However, the above is a *closed* type family; programmers can define *open* type families which can be extended beyond their initial definition. This mimics ad-hoc polymorphism, in that different implementations of the same type family can be offered with different types as input.

There are more notable differences between (closed) type families and functions beyond syntax. The most important is that type families cannot be partially applied in the same way that functions can. Consider a function (and closed type family) `IsEitherFiction`, which takes in two books and states whether either of them are fiction or not. A function definition, and a closed type family definition, are below:

```haskell
isEitherFiction :: Book -> Book -> Book
isEitherFiction Fiction _ = True
isEitherFiction NonFiction Fiction = True
isEitherFiction NonFiction NonFiction = False

type family IsEitherFiction (x :: Book) (y :: Book) :: Bool where
    IsEitherFiction 'Fiction _ = True
    IsEitherFiction 'NonFiction 'Fiction = True
    IsEitherFiction 'NonFiction 'NonFiction = False
```

While the function `isEitherFiction` can be partially applied, the type family `IsEitherFiction` cannot. One could feasibly map `isEitherFiction` over a list of books, but mapping with the type family `IsEitherFiction` is impossible:

```
$> map (isEitherFiction NonFiction) [NonFiction, Fiction]
= [False, True]
$> kind! Map (IsEitherFiction 'NonFiction) '[ 'NonFiction, 'Fiction ]
error: The type family 'IsEitherFiction' should have 2 arguments, but has
   been given 1
```

Sadly, lots of functional programming relies on partial application, and these facilities simply aren't available when using Haskell's Type Families.

2.2.3 First-Class Families

The above problem is still an open one in type-level programming, but one solution comes from Li-yao Xia, who put together a Haskell library named First Class Families[2]. First Class Families allow the programmer to map over structures, and specialise type families (a la ad-hoc polymorphism), similar to value-level functions. Sadly, First Class Families is not supported by any formal literature on the topic at the time of writing; so we briefly introduce and explain the concept below.

It relies on a type, `Exp`, and an open type family, **`Eval`**. They are defined as so:

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

Using these two definitions, a type-level interpreter becomes available for use. While type families cannot be partially applied, type and kind constructors have no such restriction; and so passing around the types as `Exp` types allow the programmer to partially apply, and to evaluate whenever they choose by calling **`Eval`**. For instance, consider the `IsEitherFiction` type family, but defined in "First Class Family" style instead:

```haskell
data IsEitherFiction :: Book -> Book -> Exp Bool
```

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

```haskell
type instance Eval (IsEitherFiction Fiction Fiction) = True
type instance Eval (IsEitherFiction Fiction NonFiction) = True
type instance Eval (IsEitherFiction NonFiction Fiction) = True
type instance Eval (IsEitherFiction NonFiction NonFiction) = False
```

When combined with a definition of `Map`, mapping (and general Functor behaviour) at the type level becomes possible:

```haskell

```

However, due to their implementation via an open type family, First Class Families have the restriction that they cannot have overlapping definitions. For instance, the below will not compile:

```haskell
data IsEitherFiction :: Book -> Book -> Exp Bool
type instance Eval (IsEitherFiction Fiction _) = True
type instance Eval (IsEitherFiction x Fiction) = True
type instance Eval (IsEitherFiction x NonFiction) = False
```

It will not compile because when determining which definition to use for `IsEitherFiction Fiction Fiction` (or `IsEitherFiction Fiction NonFiction`), the compiler is unable to tell whether to bind the first argument (`Fiction`) to the `x` or to use the first definition. When using a closed type family, or a term-level function, the definitions written are implicitly ordered, so if it is unclear, the behaviour is to default to the first definition written. Luckily, this can be leveraged; by using a combination of First Class Families and closed type families, both partial application and ordered definitions can be used:

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

type family IsEitherFiction' (x :: Book) (y :: Book) :: Bool where
    IsEitherFiction' 'Fiction _ = True
    IsEitherFiction' x 'Fiction = True
    IsEitherFiction' x 'NonFiction = False
```

2.2.4 Proxies and Singletons

3 Design

3.1 The Basics of Chess

3.1.1 The Board

The Pieces

3.1.2 The Game

3.1.3 Chess Notation

3.2 Type-Level Data Structures

3.2.1 Chess Data Structures

An important part of any good chess program is its board representation, since all other parts of the program come from this; move generation, move evaluation, and the entire search space are all defined or influenced by the board representation. A great deal of work has gone into defining memory- or time-efficient chess boards [19] [20], including combinations of multiple representations to yield greater speed [21]. While there is value to be gleaned from examining these representations, Chesskell serves a different purpose; it does not need to search through the valid set of moves to determine which are the best, and speed is not its focus. Chesskell's board representation must be relatively efficient, but it would be naive to expect similar levels of performance from type-level constraint solving computation as from optimised term-level code.

4 Implementation

4.1 Type-Level Chess

4.1.1 Chess Types and Kinds

The Pieces

The Board

Miscellaneous Types

4.1.2 Chess Rules

Movement Rules

Attack/Capture Rules

Checking For Violations

Exceptions

Castling

Pawn Capture and En Passant

4.2 The EDSL

4.2.1 Minimum Viable Product

4.2.2 Flat Builders

4.2.3 Moving the pieces

4.2.4 Setting up a board

5 Evaluation

5.1 Testing

5.1.1 Type-level Unit Testing

5.1.2 Testing Chesskell Games

5.2 Compile Time and Memory Usage

5.2.1 Optimisation Attempts

Board Decorators

Finger Trees

5.2.2 GHC Bug Report

5.2.3 Descriptive Error Messages

Move Number

5.3 Chesskell EDSL vs Other Chess Notations

6 Conclusions

Chesskell is a successful project. ...

6.1 Results and Accomplishments

6.2 Future Work

6.2.1 Session-typed Chesskell

7 Bibliography

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

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

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

[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] 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.

[6] 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.

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

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

[9] H. Xi and F. Pfenning, "Eliminating array bound checking through dependent types," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pp. 249–257, 1998.

[10] S. Lindley and C. McBride, "Hasochism: the pleasure and pain of dependently typed haskell programming," *ACM SIGPLAN Notices*, vol. 48, no. 12, pp. 81–92, 2013.

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

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

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

[14] 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] 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.

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

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

[21] S. K. Bimonugroho and N. U. Maulidevi, "A hybrid approach to representing chessboard using bitboard and compact chessboard representation," in *IOP Conference Series: Materials Science and Engineering*, vol. 803, p. 012018, IOP Publishing, 2020.

8 Appendix