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

Toby Bailey

February 17, 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 full type-level model of, and rule-checker for, Chess—along with a Haskell-Embedded Domain-Specific Language for notating Chess games. 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

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 (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 program behaviour becomes 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 obvious 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 is suitable 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

6

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].

Of course, type-level rule checking in Haskell is very possible. Through clever use of types, Lindley and McBride's merge sort implementation [10] is guaranteed to produce sorted outputs. Unit tests for the sorting implementation become unnecessary,

---

[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-d readed-and-wanted-languages

7

since the GHC type checker is used to ensure that 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 unrelated 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 enforcers, 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 compiling 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 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[9,10]. 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; 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 value-level. Such a feat is both simple and unoriginal.

However, choosing to write Chesskell in Haskell means figuring out how to perform typical value-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 primary method of interfacing 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 purely functional language, which is statically typed and lazily evaluated[1]. As we explain above, languages with a dependent type system have no separation between types and values. In Haskell however (among other languages), programs undergo *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 have. Haskell cannot be used to write functions on types, in the same way that a dependently typed language like Idris can.

This is just one of the differences between Haskell's type system and a dependent type system, but it is an important one; type-level programming in Haskell is made more complex due to type erasure. This chapter summarises key aspects of Haskell's type system, including how programmers can use GHC extensions and advanced features to circumvent type erasure and perform computation at the type level in Haskell.

2.1 Types in Haskell

For a Haskell compiler, such as GHC, to accept a program, it must be well-typed (that is, every expression has a valid type). In some cases, the type of an expression can be inferred, or it can be manually annotated by the programmer (in which case the compiler must unify the annotated type with the inferred type). If the expression has no permissible type, or its' inferred type does not match its programmer-annotated type, then the compiler is responsible for generating a type error.

For instance, as in many other languages, one of the possible types for 3 is `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'
```

However, Haskell has support for *polymorphism*; firstly, 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

---

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

value of any type, provided that all elements of the list are of the same type. For instance, a list of booleans, such as [`True, False`], has type [`Bool`], which is notated as [`True, False`] :: [`Bool`].

In Haskell, type variables are conventionally named as lowercase single letters in alphabetical order; so the first general type in an type annotation is typically 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 (except for keywords). As such, the polymorphic empty list [] has type [a], where the type variable a can be unified with other types, such as `Bool` in the example above.

Secondly, Haskell supports ad-hoc polymorphism, whereby functions can be specialised to operate on specific types, with separate definitions for each type. Haskell achieves this through *type classes*, a feature which is akin to interfaces in object oriented languages. We give an example below, involving the basic equality operator in Haskell (which is a member of the `Eq` typeclass). Example definitions for both `Bool` and `Int` are given to demonstrate how different definitions can be used for the same function:

```haskell
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Bool where
    True  == True  = True
    False == False = True
    True  == False = False
    False == True  = False

instance Eq Int where
    0 == 0 = True
    0 == y = False
    x == 0 = False
    x == y = if x > 0 then (x - 1 == y - 1) else (x + 1 == y + 1)
```

Haskell functions are of the form a -> b, for some type variables a and b which can unify with any other type, including function types. For instance, Boolean logical AND would have the type `Bool` -> `Bool` -> `Bool`, while logical NOT would have the type `Bool` -> `Bool`.

All Haskell functions can be curried [18]; as an example, assume the definition of a function and, which performs logical AND on its two inputs. The type of and would be `Bool` -> `Bool` -> `Bool`, while the type of and `True` would be `Bool` -> `Bool`, and the type of and `True False` would just be `Bool`. The expression and `True` is well-typed, and would compile; contrasting with other languages such as C or Java, in which functions cannot be partially applied and an analogous expression (e.g. and(`True`)) would fail with a type error.

Haskell allows the programmer to define their own data type with the keyword `data`.

These data types are *algebraic*, meaning that they are types 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:

```haskell
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 GHC extension allows the definition of *Generalised Algebraic Data Types* (GADTs) [19] which allows more complex type constructor definitions. 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 Lazy Evaluation

Haskell is a *lazily evaluated* language, which means that values are computed on-demand. This contrasts with *eagerly evaluated languages* like C or Java, which evaluate their arguments before applying functions. A good example of this behaviour is with infinite list types. Trivially, if a program tries to fully evaluate an infinite list, then it will never terminate. However, in a lazily evaluated environment, an infinite list can be used for computation.

Consider the following example, where the (:) operator adds an element to the beginning of a list, and **take** n list takes n elements from the start of list. One could define an infinite list as such:

```haskell
infiniteList = 1 : infiniteList
```

With such a definition, the expression `take` 5 `infiniteList` both terminates and is well-typed, returning the value [1,1,1,1,1].

Another example demonstrates lazy evaluation: in Haskell, the **undefined** value simply throws an exception whenever evaluated. We use it to demonstrate Haskell lazy evaluation: consider an expression list defined as list = [1,2,3,**undefined**] (which compiles successfully). **take** 3 list evaluates to [1,2,3], while **take** 4 list generates an exception. For a non-list example, consider if expressions. **if True then** 3 **else undefined** evaluates to 3, since the else branch is not evaluated.

All values in Haskell are lazily evaluated (unless explicitly set otherwise in code). The expression 5 + 2 * 3 will only be evaluated to 11 when that value is required somewhere in code, and if it is never required, it is never evaluated.

2.3 Type-level Programming

While GADTs and lazy evaluation are certainly useful in day-to-day programming, they are not enough to achieve complex type-level computation. Luckily, there are many more GHC extensions, a large number of which bring the language closer to dependent types.

2.3.1 Kind Promotion

A key concept in type-level programming in Haskell is that of *promotion* [4]. The data types that programmers define (as we explain above) can be promoted to *kinds*. Kinds are, conceptually, the types of types; that is, values 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 * -> *, while the kind of a Boolean list ([**Bool**]) is *.

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, with either regular or GADT syntax:

```
data Book = Fiction | NonFiction

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 can be left out where the compiler can unambiguously state whether an expression should be a type or a value.

2.3.2 Type Families

Another key extension introduces *type families* [5] [6]. 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 value-level definition could be as follows:

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

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

```
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:

```
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. Imagine a type family `Map`, of kind `(a -> b) -> [a] -> [b]`, analogous to the value-level function `map`. While the value-level expression `map (isEitherFiction NonFiction) [NonFiction, Fiction]` evaluates to [**False**, **True**], the type-level equivalent (`Map (IsEitherFiction 'NonFiction)` `'[ 'NonFiction, 'Fiction ]`) causes a type error.

Sadly, much of functional programming relies on partial application, and these facilities simply aren't available when using Haskell's Type Families. Additionally, Type Families are not lazily evaluated, but are eagerly evaluated; we demonstrate with the definition of a custom `If` type family, combined with a type error:

```
type family If (x :: Bool) (t :: k) (e :: k) :: k where
    If True  x _ = x
    If False _ y = y
```

The type-level expression `If` **True** 3 4 has kind `Nat`, and evaluates to 3. However, the type-level expression `If` **True** 3 (`TypeError (Text "Not lazily evaluating!")`) results in a type error; demonstrating that Type Family arguments are evaluated before application. Compared with the behaviour of Haskell's value-level **if**, it is clear that type families are eagerly evaluated rather than lazily evaluated.

2.3.3 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 like so:

```
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:

---

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

16

```
data IsEitherFiction :: Book -> Book -> Exp Bool
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 by calling **Eval**: the expression **Eval** (Map (IsEitherFiction 'NonFiction) '['NonFiction, 'Fiction]) evaluates to '[**'False**, **'True**].

However, since an open type family is used to define First Class Families, they cannot have overlapping definitions. For instance, the below will not compile:

```
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
```

When using a closed type family, or a value-level function, the definitions written are implicitly ordered, so if instances overlap, 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:

```
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
```

Type-level Lazy Evaluation

First Class Families introduce more than just partial application; they can be used to perform type-level lazy evaluation. Consider a First Class Family, TE, which takes in an ErrorMessage and returns a TypeError:

```
data TE :: ErrorMessage -> Exp a
type instance Eval (TE msg) = TypeError msg
```

When combined with **Eval**, TE can be used to return a type error: **Eval** (TE (Text "Custom error!")) generates error: Custom error!.

By guarding the actual type error behind **Eval** calls, type errors can be lazily evaluated, as can other type level constructs. Consider another First Class Family for if statements:

```
data If :: Bool -> Exp a -> Exp a -> Exp a
type instance Eval (If True  x y) = Eval x
type instance Eval (If False x y) = Eval y
```

Alongside a First Class Family version of the identity function:

```
data ID :: a -> Exp a
type instance Eval (ID x) = x
```

The First Class Family version of If supports lazy evaluation; as can be demonstrated by placing an unevaluated type error in the else branch. The type-level expression Eval (If True (ID 3) (TE (Text "Not lazily evaluated!"))) evaluates to 3, rather than generating a type error.

The behaviour of the First Class Family definition is different from that of the Closed Type Family definition we give above; the type error is never evaluated, as the Eval call on the else branch is never made. As such, First Class Families can be used to perform lazy evaluation at the type level.

2.3.4 Type Applications

The -XTypeApplications Haskell syntax provides a way for the programmer to directly specify type variables [20]. Consider an empty list, with type [a]. Using type application syntax, where we prefix a type name with @, one can specify the type of an empty list by stating what type should inhabit type variable a.

For example: the empty list [] @Int has type [Int], and the empty list [] @Bool has type [Bool]. Note that the empty list value has been used in all cases; the thing that has changed is the type of that empty list.

2.3.5 Proxies and Singletons

While promotion and type families allow the programmer to compute at the type level, there must be some way to pass information between the value level and the type level for this to be useful. For Chesskell, this communication only needs to be one-way; the value-level EDSL passes information up to the type system, which either compiles successfully or throws a type error. Promoted types (such as 'Fiction) have no runtime values, and so cannot be used as value-level function argument types. There are two widely used methods of circumventing this limitation in Haskell to mimic values with these promoted types; *proxies* and *singletons*.

Proxy Types

Proxy types provide a wrapper to allow arbitrary types to have kind `*`. As we explain above, all value-level functions take in values, and all values have types with kind `*`. The `Proxy` type constructor takes in a single type variable, and exposes a polymorphic value `Proxy`. The `Proxy` type constructor, when applied to some type, has kind `*`. To follow on from the previous `Book` example, while `'NonFiction` has kind `Book`, `Proxy 'NonFiction` has kind `*`, and so values of type `Proxy 'NonFiction` can be passed around at the type level.

By making use of the type application syntax we explain above, we can demonstrate how `Proxy` values can be used to pass around type variables with arbitrary kinds. While `'NonFiction` has kind `Book`, `Proxy 'NonFiction` has kind `*`. Additionally, since it has kind `*`, it has a runtime value; `Proxy @NonFiction` is the value of type `Proxy 'NonFiction`.

Singletons

As helpful as proxy types can be, they have one limitation; since all the value-level code sees are `Proxy` values, all of the relevant information is only available at the type-level. However, singleton types [8] provide an alternative approach. Each singleton type has a single inhabitant value, and each individual value has a single unambiguous type. In other words, it's a large step towards dependently typed programming.

This is achieved through, for each defined type, running it through Richard Eisenberg's singletons library's Template Haskell definitions[3]. Template Haskell is a compile-time meta-programming system, similar to macros in that it allows programmers to define programs to modify and generate Haskell source code [21]. The singletons library uses Template Haskell to define new data types, given data type definitions. Should singletons be given the definition of the `Book` datatype which we detail above, it will generate a new `SBook` datatype, defined as below:

```
data SBook :: Book -> * where
    SFiction    :: SBook 'Fiction
    SNonFiction :: SBook 'NonFiction
```

Due to datatype promotion, this introduces new values, types, and kinds. The runtime value `SFiction` has the type `SBook 'Fiction`, and the type `'SFiction` has kind `SBook Fiction`. These definitions are designed to enable the programmer to access type information through values, and vice versa.

The type of `SNonFiction` can only be `SBook 'NonFiction`, and so value-level code now has some intuition of types; and conversely, when given a type `SBook a`, type families can use the type variable a which will be either `'Fiction` or `'NonFiction`.

---

[3]https://hackage.haskell.org/package/singletons

19

3 Design

This chapter of the dissertation details the general design of Chesskell. Broadly, Chesskell is split into two main sections; the type-level chess model and ruleset, and the value-level EDSL which acts as an interface for the type-level chess model.

Additionally, we explain some basic Chess knowledge in this chapter, to aid in understanding. However, we tackle the more complex rules when they become relevant; this chapter does not constitute a formal introduction to Chess, but a simple summary to make the design of Chesskell clear.

3.1 The Basics of Chess

Chess is a two-player game, played in alternating moves by players typically named *Black* and *White*, after the colours of their pieces. In each turn, the player will move a single piece, and cannot abstain from making a move (or move a piece from its position to that same position). Each piece is governed by its own movement rules, which depend on the state of the board and, in some cases, the history of that piece or other pieces' movements.

3.1.1 The Board

The board is an 8x8 grid of 64 square tiles, each of which is coloured Black or White such that each square is next to tiles of the opposite colour (see Figure 8.1). The pieces move within this board, and cannot take moves that would wrap around it or take them off of the board.

At the beginning of the game, all Chess pieces lie in a specific arrangement (see Figure 8.2). All Black and all White pieces are opposite one another, such that their positions are mirrored.

3.1.2 The Pieces

While each team has 16 pieces total, there are only 6 types of pieces; Pawns, Rooks, Knights, Bishops, Queens, and Kings (in rough order of value during play). Each

have their own strict movement rules, and in all but a single case, pieces of the opposite team can be *captured* by moving to their square. A capture removes a piece from play; there is no way to regain a piece once captured (although there is a way to transform a Pawn into another piece). We give an example of capturing in Figure 8.3.

3.1.3 The Game

A King is in *check* when they are in the attack path of another piece. The objective of the game is to place the opponent's King into *checkmate*, whereby every move the King could make is to a position where they would be in check (see Figure 8.4 for an example). Additionally, a move by a team that would place that team's King into check is an invalid move, and cannot be made.

There are additional ways in which a Chess game may end, such as when two opponents agree to a draw; however, these additional rules concern the players of Chess rather than the game itself, and so are not a part of the implementation of Chesskell.

3.1.4 Chess Notation

There are two main categories of chess notation; those concerning the state of the game, and those concerning the state of the board. Further relevant details on specific Chess notation will be tackled as and when relevant; this section is only aimed as a minor note of their existence.

Chess notation concerning the state of the game tends to be an account of the whole set of moves, starting from the standard start positions. Algebraic notation is the most common chess notation, and is used by FIDE to record matches between professional chess players. Each piece type other than Pawn is denoted with a capital letter: K for King, Q for Queen, B for Bishop, R for Rook, and N for Knight. As such, the move Na4 means that a Knight has moved to the position "a4" on the board. Algebraic notation typically does not include the square the piece moved from; only its destination square. Figure 8.5 shows the initial state of the board, followed by a set of moves in algebraic notation, and the resulting state of the board.

There are stylised variants of Algebraic Notation, such as Figurine Algebraic Notation, in which symbols for the pieces replace the capital letter. For example, Na4 is written as ♘a4, whereby the N is replaced with the symbol for a Knight. This is the variant used in the dissertation; future examples use Figurine Algebraic Notation, where chess symbols replace capital letters.

The other class of chess notation is that for board creation or description. Forsyth-Edwards Notation (FEN), a popular example, simply states which pieces are where on a board, line-by-line. White pieces are denoted with uppercase letters, and Black pieces are denoted with lowercase letters. The letters used match those for Algebraic

Notation, save for the introduction of P for White Pawns (and p for Black Pawns). Lines are described as series of pieces and empty spaces, such that a row with a White Pawn on every other position would be described as `P1P1P1P1`. Another row with two spaces between Black Pawns would be described as `p2p2p1`, since the total number of row positions must equal eight. See Figure 8.6 for an example of a board created with FEN notation.

3.2 Type-Level Data Structures

As helpful as type families and First Class Families are in enabling computation at the type level, this computation is useless without something to compute on. Chesskell requires some central repository of information for the state of the board, as well as general data structures for passing around information while validating the Chess ruleset. This section describes the type-level data structures in Chesskell.

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 [22] [23], including combinations of multiple representations to yield greater speed [24]. 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 value-level code.

3.2.2 Singly-linked Lists

In Chesskell, Haskell's built-in type-level lists are not used as the primary board type. These lists are singly linked, and so have a variable length which is checkable in $O(n)$ time. Ensuring that the chess board remains an 8x8 grid at all times would incur a repeated cost on the compile time of the program. However, these lists are used for data which can be of variable length; such as the list of available moves for a piece in a specific position.

3.2.3 Finger Trees

An alternative to type-level lists would be to use 2-3 Finger Trees [25]. Unfortunately, singly-linked lists have no quick "append" operation. As such, combining lists of moves takes $O(n)$ time, which could be considerable for pieces like Queens who have many moves available to them at any one time. However, Finger Trees can be combined in $O(log(min(n_1, n_2)))$ time, where n_1 and n_2 are the sizes of the respective FingerTrees. Singly linked lists have an $O(1)$ append, while Finger Trees have an *amortized* $O(1)$ append operation.

Finger Trees are so named because while the main portion of the data is in recursive tree form, each tree maintains two "hands" full of data. Essentially, each of these appendages is a small overflow buffer for the tree itself, since inserting into the tree is more costly ($O(logn)$)) than inserting into the buffer ($O(1)$). A pleasant side effect of this approach is that not only can you access data at the beginning of the sequence in $O(1)$ time, you can also access data at the end of the sequence in $O(1)$ time; something impossible with Haskell's built in singly linked lists.

Disappointingly, there exists an implementation of Chesskell using Finger Trees as opposed to lists for variable length data, but as we discuss in the Evaluation chapter, there was no significant increase in compile time relative to the effort spent implementing Finger Trees at the type level.

3.2.4 Length-indexed Vectors

To implement the board, a desirable data structure would be one that had a fixed type, which could be guaranteed to remain at 8. As such, a variant of singly-linked lists, named length-indexed vectors. A length-indexed vector is a singly linked list which contains its' length in its' type. That is, a length-indexed vector of size 0 has a different type than a length-indexed vector of size 3. We give an example GADT data type definition below:

```haskell
data Vec (n :: Nat) (a :: *) where
    VEnd   :: Vec 0 a
    (:->)  :: a -> Vec n a -> Vec (n + 1) a
```

If the programmer should require the input vector to be of length 5, then all they must do is include its length in the function definition:

```haskell
someFunc :: Vec 5 a -> b
someFunc vec = -- ...
```

This makes it a perfect candidate to act as the central chess board type, containing all pieces. To guarantee that a board is an 8x8 grid, it simply needs to contain 8 length-indexed vectors of length 8. Due to the use of the -XDataKinds extension to

enable promotion, this length-indexed vector definition immediately also defines a type-level length-indexed vector.

Almost all operations available on lists are available on length indexed vectors. However, since length-indexed vectors have an additional type variable (their length), they are difficult to dynamically create without some length type variable. That is, a function `f :: a -> Vec n b` cannot exist, since the type variable `n` will have nothing to unify with when `f` is called.

3.3 Modelling Chess with Functions

Ideally, the Chess board alone would be sufficient to calculate whether a move was valid. A value-level function for determining the validity of moves could take in the current state of the board, and two positions (the position moving from and the position moving two), and either return the new board state or some king of error. Since Chess is conducted move by move, to simulate a game, this function could be chained repeatedly, with each new move and the previous generated board as input. Such an ideal function could have type `ChessBoard -> Position -> Position -> `**`Maybe`**` ChessBoard`.

In a game of Chess, the majority of moves are time-agnostic; that is, they are not tied to previous moves, only the current state of the board. There are, however, two exceptions; Castling and *en passant* capture. Castling is a move by both a King and a Rook, and an *en passant* capture is a special form of capture available only to Pawns. However, the only additional information required to calculate whether these moves are valid is the last piece that moved, and for each piece the number of times that piece has moved. As such, the board representation can be defined to include this information; ensuring that not only pieces and teams are recorded, but also the number of moves made and the last piece to make a move. Therefore, with a new type `DecoratedChessBoard` containing the new information (as well as the board state), a function for calculating move validity could have type `DecoratedChessBoard -> Position -> Position -> `**`Maybe`**` DecoratedChessBoard`.

A pure function implementation is therefore possible, making use of a Chess board data structure which includes this information. Translating this approach to the type-level, a Type Family (or First Class Family) can be defined which performs similarly, with kind `'DecoratedChessBoard -> 'Position -> 'Position -> 'DecoratedChessBoard`. This type-level model of Chess, implemented as a single movement Type Family, must be interacted with via the defined EDSL. The EDSL is responsible for gathering move-wise positional information, and chaining together calls of the movement Type Family, which will either return a valid Chess board or a type error depending on whether the described move is permissible or not.

3.4 Designing an EDSL for Chess

Since the EDSL is for describing games of Chess, it makes sense that it should draw inspiration from Chess game notation, such as Algebraic Notation (which we explain briefly above). In such notation, the board state is implicit and undescribed; that is, the state of the board must be inferred by the reader from the moves made thus far, assuming that the game started in standard configuration (Figure 8.2).

One possible match in Haskell for this style is monadic computation. If the board information were stored in a custom monad, then the *bind* operator (written as >>=)) could be used to chain together these chess moves, in some way akin to below:

```
game = chessStart
    >>= move e2 e4
    >>= move e7 e5
    >>= -- ...
```

However, this approach introduces a few problems. Firstly, the EDSL is more difficult to read for those unfamiliar with Haskell. It immediately would be less an EDSL, and more a set of plain Haskell functions with nice names. Secondly, it is not immediately clear how a monad for the type-level chess board could be defined. It could piggyback off of another defined monad, such as the `Maybe` monad, but this is introducing further complexity for no good reason.

Luckily, there exists an alternative; Continuation Passing Style (CPS). The core idea is value transformation through a series of continuation function applications, until the final continuation function returns a value. Due to the left-to-right composition of functions, CPS results in very readable code, and could be utilised to avoid any Haskell-specific operators and appear as a clear stretch of Chess notation.

Consider the following example of CPS code. We give the definition of two functions, `add` and `to`:

```
add :: Int -> ((Int -> Int) -> m) -> m
add x cont = cont (+ x)

to :: (Int -> Int) -> Int -> Int
to f x = f x
```

With the definition of these two functions, the line `add 5 to 7` is well-typed, and evaluates to `12`. This is because `add` takes in a continuation of type (`Int` -> `Int`) -> m, and returns a value of type m. In other words, the continuation (in this case `to` is responsible for the output. Using such a scheme enables Chesskell to be much closer to conventional chess notation than to Haskell code, and avoids wrapping the types in an unrelated monadic context.

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

5.2.4 Chesskell Shorthand

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

6.2.2 Type-level Bitboards

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] H. P. Barendregt, "Introduction to lambda calculus," 1984.

[19] 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.

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

[21] T. Sheard and S. P. Jones, "Template meta-programming for haskell," in *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pp. 1–16, 2002.

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

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

[24] 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.

[25] 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.
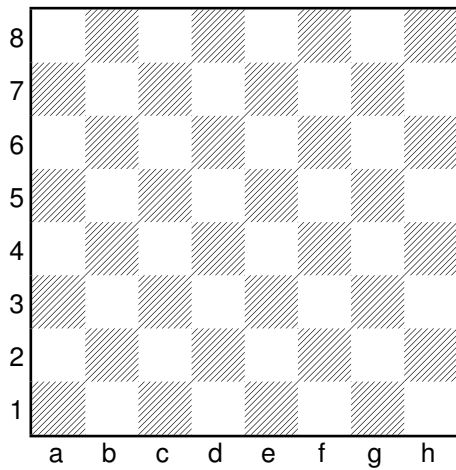
8 Appendix
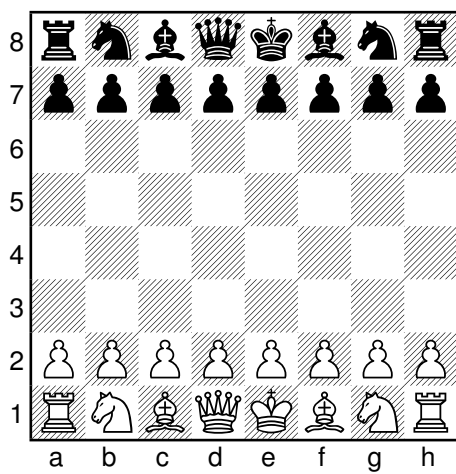
Figure 8.1: An empty chess board.

Figure 8.2: An standard chess board where all pieces are in their starting position.
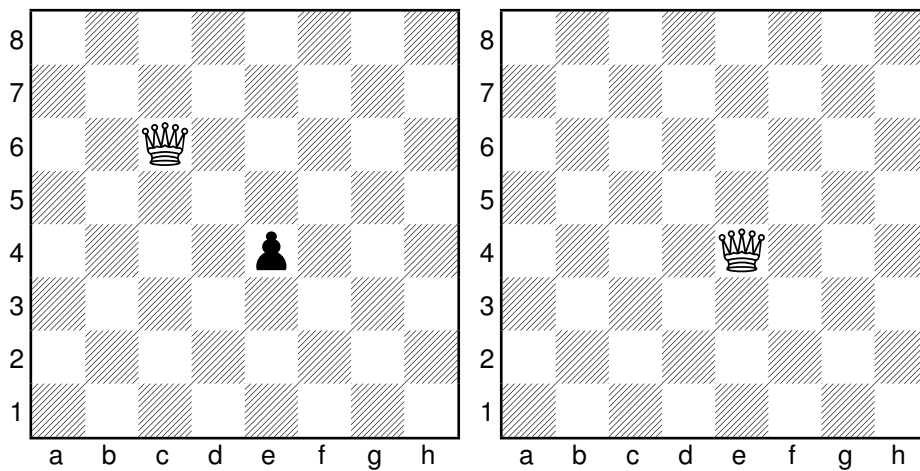

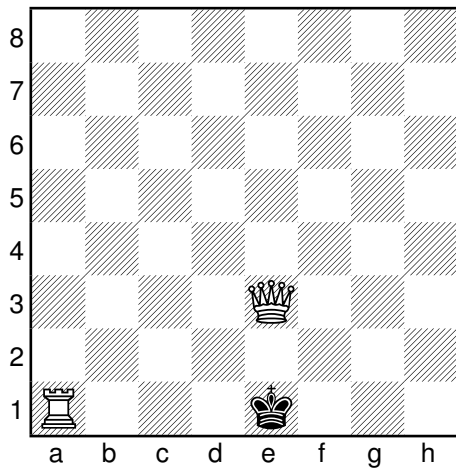Figure 8.3: The White Queen captures a Black Pawn by moving to its position, and removing it from play.

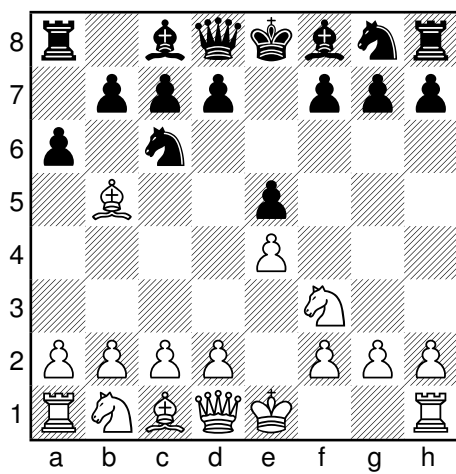Figure 8.4: The White Queen and White Rook place the Black King into checkmate.

Figure 8.5: The position of the board after: 1.e4 e5 2. Nf3 Nc6 3.Bb5 a6
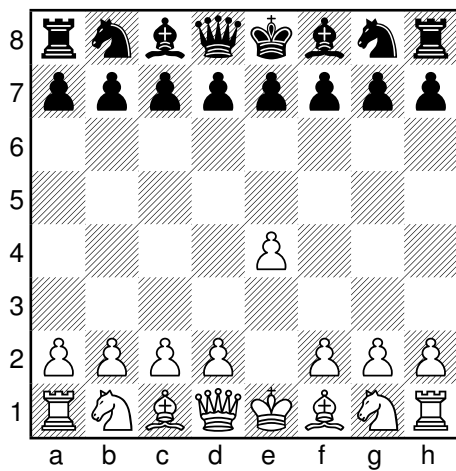

Figure 8.6: The board created with: (rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1)