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

Toby Bailey

February 24, 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

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-dreaded-and-wanted-languages`

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:

```
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:

```
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:

```
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:

```
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 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.

2.2.1 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.

2.2.2 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.

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 like so:

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

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

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:

```
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
```

2.2.4 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.

2.2.5 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

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

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 run-time 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 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

If the intention is to use it for representing a Chess board (or any other structure with a definite length), singly-linked lists have issues; how can we ensure that the chess board is the appropriate size (an 8x8 grid) without a length check each move? This would take at least 56 additions, since list length is computed recursively; as well as 7 more addition operations to put together the list lengths.

A more desirable data structure would be one that had a fixed type, which could be guaranteed to remain at length 8. As such, Chesskell makes use of 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. As with most things in Haskell, we use recursive definitions; an empty vector has length 0, and you express a vector of length (n + 1) by pushing an element to the front of a vector of length n. We give an example GADT data type definition below:

```
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:

```
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.

3.2.5 Type-Level Bitboards

One popular Chess board representation is the Bitboard [22]; using a set of 64-bit binary strings to represent the positions of pieces. Since a chess board is always 8x8, a 64-bit string (when seen as a string of 8 bytes) can hold some binary state of a particular Chess board position. Each piece type and colour needs its own bitboard, since a 1 or a 0 is not enough to differentiate between piece types. For instance, a bitboard describing White Pawns will have a 1 at every index in the 64-bit string that has a Pawn present, and will have 0s in all other positions, where the bottom left of the board is the least significant bit, and the top right of the board is the most significant bit.

The main draw of bitboards is the speed at which potential moves can be generated and the board can be modified. For instance, to move all pieces left by one square, all that is required is a left shift by 1 of the bitboard representation.

Although type-level Haskell has no bitwise operators, they could potentially be emulated through the use of pattern matching. Consider "bitwise" logical AND; each possible pair of inputs could be pattern-matched against, and the outputs enumerated. However, this code would be both laborious to write and harder to read; and a bitboard representation's main benefit is speed. Type-level operations like this would definitely not map directly to hardware bitwise operations, and so the main benefit of Bitboards would be lost. A Bitboard representation of Chesskell may indeed be faster than the vector board representation we explain above; however, it would incur a considerable complexity cost that is unlikely to be worth it, especially since type-level computation will be slow anyway. While it would make an interesting extension to Chesskell someday in the future, it is not part of the final feature set described in this dissertation.

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.

3.3.1 Checking Chess Rules

When determining if a given move of Chess is valid or not, the destination squares for all pieces is not sufficient. In other words, a function to generate the valid positions a piece can move to is not enough to enforce all rules of Chess.

Part of the relevant global state for a Chess game is the team that is currently moving; remember, White and Black teams move in an alternating fashion. It breaks the rules of Chess for a White piece to move after a White piece has just moved. There are also a few implicit Chess rules that would be helpful to have more personalised error

messages for; such as the fact that no piece can actually take the opposite King. While this information will be encoded in the fact that the opposite King's position will not be in the valid move list for that piece (even though the piece can indeed attack that position), it would be helpful to have a more specific error message for this case. Instead of `error: The Piece cannot move to that position`, it should say something like `error: Pieces cannot take their King`.

In Chesskell, an early idea was to simply check for these invariants with either type-level if statements or pattern matching. However, as the number of invariants with specific error messages grows, so too would the number of nested if statements. While such an approach would work, it is harder to follow and rather ugly.

Being in a functional environment, it is natural to express these rule checks as functions that either successfully compute something, or return a type error. Each function could essentially act as an assertion; either the input fulfils some query, or there is an error. For instance, a function `CannotTakeKing :: DecoratedChessBoard -> Position -> DecoratedChessBoard` that takes in the board state and the position to move to, and generates a type error if the position to move to is the position of either of the Kings. The reason it returns a `DecoratedChessBoard` in the successful case is so that it can be naturally composed together with the core movement function, using a First Class Family version of the Haskell function composition operator, `(.)`. Instead of code of the form `(if firstCondition then (if secondCondition then Move a1 a2 else throw "Second error") else throw "First error")`, with more and more nested if conditions, the rule-checking code has the form `(SecondCheck . FirstCheck . Move a1 a2)`, which is much easier to modify and understand.

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 piggy-back 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

The final product of Chesskell allows us to describe games of chess, move-by-move. For example, we express a simple 3-move checkmate by White as follows in Chesskell:

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

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

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

Each chess move is described with a type family, which takes as input the current state of the board, and outputs the board after the move has been processed. The core movement function, aptly named `Move`, takes in the position to move from, the position to move to, and the current state of the board, using this information to return a new board state in which the move has been made. Additionally, it updates relevant piece information for the pieces that have moved, which we further detail below:

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

The EDSL, as we explain in more detail below, uses this `Move` function to perform type-level rule checking of the described chess game. While the continuation-passing style (CPS) structure complicates the relevant types, the intuition of the EDSL is to take in the current board state, as well as the positions to move from and to, and output the new board state generated by that move. A simplified non-CPS example is below, to aid understanding:

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

4.1 Type-Level Chess

The software has a full representation of a game of chess at the type-level, as we explain below. The model is checked move-by-move, with the current board state, as well as some further information, carried between moves via a `BoardDecorator`. This `BoardDecorator` contains all information necessary to encapsulate the current state of a game of chess; in other words, Chesskell does not rely on any global state, and the state of the game is modifiable easily.

4.1.1 Chess Types and Kinds

The `Board` type is the most important type in Chesskell, representing the chess board at the type-level. It is a *composite type* containing all pieces in play. However, `Board` types are usually accessed through a `BoardDecorator`, which we detail below.

We describe Chesskell's types from the bottom up in the following sections, since the types here are composite and require understanding of other types.

Team and PieceName

Both `Team` and `PieceName` are simple algebraic data types, with all constructors defined in code. The `Team` type enumerates all teams a piece can belong to; `Black` and `White`. The `PieceName` type enumerates all possible names of pieces; Pawn, Rook, and so on. Thanks to promotion, as we explain above, these types are immediately available for use with Type Families.

```
data Team = Black | White
data PieceName = Pawn
               | Bishop
               | Knight
               | Rook
               | King
               | Queen
```

Position

The `Position` type holds the positions of pieces on the chess board. It makes use of two more types, one for columns and the other for rows. In chess, columns are labelled with letters and rows are labelled with numbers; "a1" is the top-left of the board, and "h8" is the bottom-right. The `Column` type is another simple algebraic data type enumerating all columns that a piece can reside within. The row type is a type-level implementation of Peano natural numbers, named `Nat`. Early versions of Chesskell had a custom implementation, but the current version simply uses definitions provided in `Data.Type.Nat`.

```
data Column = A | B | C | D | E | F | G | H
data Nat where
    Z :: Nat
    S :: Nat -> Nat
```

Note that the `Position` kind has a potentially infinite number of valid types, but only 64 of these types are valid chess positions. As such, there is a type family, `IsValidPosition`, which outputs **`True`** if the given position is a valid chess position, and **`False`** otherwise.

```
data Position where
    At :: Column -> Nat -> Position
```

The Pieces

Each piece, represented by the `Piece` type, contains information relevant for rule checking: that piece's team, name, and an information type. The information type, named `PieceInfo`, contains a `Nat` and a `Position`, to represent the number of moves that piece has taken, and its current position on the board (respectively). Recording the number of moves the piece has taken is important for several rules in chess, including castling and *en passant*, and so is included in the `PieceInfo` type.

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

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

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

There are several utility Type Families defined for the `PieceInfo` type to simplify code; such as `GetPosition`, a First Class Family which gets the position information from a given `PieceInfo` type.

The Board

We explain above that length-indexed vectors are an ideal choice for representing the Chess board. The board can be expressed using these vectors—a vector of 8 vectors of 8 **`Maybe`** `Piece`-s. We use **`Maybe`** `Piece` instead of just `Piece` because a board square does not necessarily contain a piece (in fact, the majority of board squares are empty):

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

Although this is the main board type, it is augmented with a `BoardDecorator`, so named because the intention is similar to the decorator design pattern [26], with the exception that subclassing and superclassing are not features of Haskell. `BoardDecorator` should be used instead of `Board`, since it provides additional information:

- The last team to move;
- The last position moved to;
- The White and Black King positions, stored as a tuple;
- The number of moves in the game thus far.

Previous versions of the program, to find the King positions (for the frequent operation of determining if a King is in check), would pass repeatedly over the `Board`. Having their positions available in the decorator saves Chesskell from making these passes. While there is the overhead that the `BoardDecorator` must be updated at each move, the code is much conceptually clearer with the use of the decorator.

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

Miscellaneous Types

4.1.2 Chess Rules

In Chesskell, the rules of Chess are expressed as type families that either return a `BoardDecorator` or a type error (as we explain in section 3.3.1). Each such check type family has the suffix `-Check`, such as `NotTakingKingCheck` and `NotTakingOwnTeamCheck`. These checks are broadly split into pre-move checks, and post-move checks—and each check has a custom error message to make clear to the user where exactly the EDSL mistake is.

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

```
data NotSamePosCheck :: Position -> Position -> BoardDecorator -> Exp
   BoardDecorator
type instance Eval (NotSamePosCheck fromPos toPos boardDec)
    = If' (Eval (fromPos :==: toPos))
        (TE' (TL.Text ("Moves from a position to that same position are
   not allowed.")))
        (ID boardDec)
```

Movement Rules

Each piece, depending on its team and name, can move a certain number of spaces. For instance, a King can move a single space in any direction. The `PieceMoveList` type family formalises this, returning a list of spaces that a piece can move to, given that piece as a `Piece` type, and a `BoardDecorator` representing the current state of the board.

```
data PieceMoveList :: Piece -> BoardDecorator -> Exp [Position]
```

Consider a `PieceMoveList` instance for Bishops:

```
type instance Eval (PieceMoveList (MkPiece team Bishop info) boardDec) =
   Eval (AllReachableDiag team boardDec (Eval (GetPosition info)))
```

Bishops can move diagonally in a straight line by any number of spaces. The type family `AllReachableDiag` is used to get a list of all diagonally "reachable" positions. It takes in the `Position` of the relevant piece, that piece's `Team`, and the current state of the board as a `BoardDecorator`. It outputs all diagonal positions that piece can move to.

Reachability for a given direction is defined in Chesskell as all the empty spaces in that direction, stopping at either the first occupied space or the edge of the board. That space is included or excluded depending on whether that space is occupied by a piece of the opposite team, since an attacking piece could move to that space and take the piece there.

When moving pieces, it is important to check that those pieces can indeed make that move. As such, there is a rule-check type family, named `CanMoveCheck` of kind `Position -> Position -> BoardDecorator -> Exp BoardDecorator`, which checks if there is a piece at the first position that can move to the second position. Additionally, for more specific error messages, there exist a few additional checks, such as `TeamCheck`, which ensures that the same team does not move twice in a row.

Attack/Capture Rules

Although they are similar, the list of spaces that a piece can attack and the list of spaces that a piece can move to are not the same. While it is true that every space a piece can attack is one that it can move to, the opposite does not hold; for example, pieces can attack the King of the opposite team, but cannot directly move to that King's position and capture it. There are other differences as well, most notably for Pawns and Kings; so `PieceMoveList` cannot be used to determine which squares a piece can attack. Another type family, `PieceAttackList`, exists which gives the list of all squares that a piece can attack.

Checking for Check

One of the most important rules in Chess, that of placing the opposite King in check, cannot be expressed solely through move and attack lists. Any movement can place either King in check, and it is not always the case that a movement by a piece places the opposite King in check; a move may be ruled as invalid because it places that piece's King into check. For instance, if a Black Rook stands between a White Queen and a Black King, the Rook is not allowed to move out of the Queen's attack path, since such a move would put the King into check.

However, the only time that check is relevant is after each move. A move by a piece is invalid when it places that piece's King in check, or if it leaves that piece's King in check. This means that it fits nicely as a post-move check, implemented as a type family `CheckNoCheck`. Early versions of Chesskell naively computed and combined all attack lists for all pieces, and simply checked if the King's position is in that combined list.

A more efficient approach, found in Chesskell today, is to emulate other pieces' movement code from the King's position. It is worth noting that for all pieces except Kings and Pawns, if they can move from a to b, then they can also move from b to a. As an illustration, if a Queen at the King's position (of the same team as the King) would be able to reach a Queen of the opposite team, then the King would be in check.

As such, several "rays" are sent out from the King's position and check for Queens of the opposite team, as well as Bishops (for diagonal rays) and Rooks (for rank and file rays). Attacking Pawns are also checked here, for the immediate diagonal positions either above the King (if the King is White) or below the King.

Additionally, Knight movement rules are applied to check if there are any Knights reachable from the King's position; and if so, then the King is in check.

There is one last piece type not handled by the above method—Kings. This is deliberate; it would be illegal for a King to move within attacking distance of the opposite King, since then the moving King would be in check.

A code snippet for determining if the King is in check, which checks if any of the above conditions are true, is below for understanding. The first class family `Any` returns true if any elements of a list of Booleans are true, and false otherwise. Each of the `-Ray` functions returns true if a piece could place a King in check from that direction, and the `IsKnightAttacking` function returns true if any Knights of the opposite team are reachable from the given position:

```
data IsKingInCheck :: Position -> Team -> BoardDecorator -> Exp Bool
type instance Eval (IsKingInCheck kingPos team boardDec) = Eval (Any '[
    SendLeftRay kingPos team boardDec,
    SendRightRay kingPos team boardDec,
    SendAboveRay kingPos team boardDec,
    SendBelowRay kingPos team boardDec,
    SendNWRay kingPos team boardDec,
```

```
    SendNERay kingPos team boardDec,
    SendSWRay kingPos team boardDec,
    SendSERay kingPos team boardDec,
    IsKnightAttacking kingPos team boardDec ])
```

Exceptional Rules

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

Castling

Most Chess rules move a single piece, and can capture another piece to remove it from play. However, the *Castling* move involves the movement of two pieces; the King, and one of their Rooks. Not only this, but Castling can only occur if neither the King nor the Rooks have moved, and as long as none of the positions the King would move through are under check and there are no other pieces between the King and the Rook. It is one of the most complex Chess rules, and requires many tests before it can proceed.

There are two varieties of Castle; Queen-side Castle and King-side Castle, depending on the direction that the King moves in. Both varieties are shown for the Black team in **????**. Essentially, the King moves either 2 or 3 spaces towards the Rook, and the Rook wraps around to the other side of the King.

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

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

type family CanCastleToEitherRook (t :: Team) (b :: BoardDecorator) ::
   (Bool, Bool) where
    CanCastleToEitherRook team boardDec = (Eval (PairAnd
   (HaveRooksNotMoved team boardDec)
        (Eval (PairAnd
            (Eval (PairPredicate (Eval (CastleSpacesToTest team
   boardDec)) (Not . AnySpaceInCheck team boardDec)))
            (Eval (PairPredicate (BetweenKingAndRook team) (AllSpacesFree
   boardDec)))))))
```

The above code first checks if the King has moved; if they have not, then it checks if both Rooks have not moved. If they have not moved either, then it determines if any of the spaces the King would move through are in check, and then if there are any pieces between the King and the Rook. This logical AND chaining is performed using a type family `PairAnd`, which performs element-wise logical AND on each pair.

This extended Castling check illustrates why it is useful to have each piece's move count in the `PieceInfo` type; it enables quickly determining if a King or either of the Rooks have moved. Simply checking if the King or Rooks are in their starting positions is not enough, since they could have just moved back to those positions.

Luckily, there are no additional checks required for Kings to castle other than including it in the King's move list and ensuring that castling is valid when it is described; there are no circumstances under which they are obligated to castle. However, a King cannot castle to capture another piece; so these castle positions are not a part of the King's attack list.

Pawn Movement and En Passant

Pawns have the most complex movement rules out of any piece, primarily because their attack patterns are different from their movement patterns. Pawns can move one vertical space forwards, but on their first move can move two spaces instead of one. However, they cannot capture a piece this way; they can only capture one diagonal space in front of themselves.

This means that a Pawn can indeed make a diagonal move, but only if there is a capturable piece there. (For instance, a Black Pawn could move diagonally a single space to capture a White Bishop, but could not move to that square if it were empty.) Additionally, Pawns have one more special capture rule; that of *en passant*.

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

This capture rule is dependent on some unusual factors; the last move made, as well as the relative positions of the pieces. Furthermore, the capture move does not result in the attacking piece landing on the square of the piece being captured. As such, implementing *en passant* capture was one of the more complex parts of development.

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

```
data GetEnPassantPosition :: Position -> BoardDecorator -> Exp [Position]
```

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

data EnPassantPosition :: Team -> Position -> Exp [Position]
type instance Eval (EnPassantPosition team pos) = EnPassantPositionNonFCF
   team pos

type family EnPassantPositionNonFCF (t :: Team) (p :: Position) ::
   [Position] where
    EnPassantPositionNonFCF White (At col row)     = '[ At col (S row) ]
    EnPassantPositionNonFCF Black (At col (S row)) = '[ At col row ]
```

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

While the above is certainly more complicated than other move checks, First Class Families is used to simplify what would otherwise be a very long chain of pattern matched logic. The first class family `GetPieceAtWhichDec` returns a **`Maybe`** `Piece` depending on whether there is a piece at a given position which fulfils a given predicate, returning **`Nothing`** if the predicate does not evaluate to true. Additionally, a type-level first class family version of `FromMaybe` is used to either propagate the **`Nothing`** type, or to transform the wrapped value with a predicate.

Implementing *en passant* captures was the driving factor that prompted the creation of the `BoardDecorator` type, since the last position moved to was required as part of the process. Such a change necessitated much code refactoring. Ultimately, despite the complex nature of the rule, it is successfully implemented in Chesskell, and *en passant* captures are very possible, as we demonstrate with the successful compilation of the below game:

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

Pawn Promotion

Pawns have one last complex movement rule; that of promotion. When a Pawn makes it to the opposite side of the Board, they must be promoted to another piece type; either a Queen, a Bishop, a Rook, or a Knight. Pawns must be promoted; they cannot opt out of promotion.

Promotion itself proved to be another complex inclusion, since the core `Move` family that takes in two positions and a `BoardDecorator` does not hold enough information to determine what a Pawn should be promoted to. While one potential solution is to hold this information as a **`Maybe`** `PieceName` in the `BoardDecorator`, promotion is infrequent and never occurs in many games. Therefore, instead of using the base `Move` first class family, a new first class family called `PromotePawnMove` is used instead:

```
data PromotePawnMove :: Position -> Position -> PieceName ->
   BoardDecorator -> Exp BoardDecorator
type instance Eval (PromotePawnMove fromPos toPos promoteTo boardDec)
    = If' (Eval (IsPieceAtWhichDec boardDec fromPos (IsPiece Pawn)))
        ((PromotePieceTo promoteTo toPos . Move fromPos toPos) boardDec)
        (If (Eval (IsPieceAt boardDec fromPos))
            (TE' (TL.Text ("The piece at: " ++ TypeShow fromPos ++ " is
   not a " ++ TypeShow Pawn ++ ". Non-Pawn pieces cannot be promoted.")))
            (TE' (TL.Text ("There is no piece at: " ++ TypeShow fromPos
   ++ "."))))
```

`PromotePawnMove` is very similar to `Move`, and indeed calls it internally; but in addition to promoting the piece after it has moved, it ensures that the position to be moved from contains a Pawn (since it is the only piece type that can be promoted).

The type family `PromotePieceTo` of kind `PieceName -> Position -> BoardDecorator -> Exp BoardDecorator` is responsible for changing the `PieceName` of the Pawn once it has reached the opposite end of the board. It applies a first class family, `PromoteTo`, to the piece at the given position to change its `PieceName`. Additionally, it generates type errors if the user attempts to promote the Pawn to a King or another Pawn.

However, `PromotePieceTo` and `PromotePawnMove` alone are not enough; because the code must also enforce the rule that a Pawn must always promote when it reaches the opposite end of the board. As such, there is a post-move rule check type family named `ShouldHavePromotedCheck`, which is responsible for determining whether a promotion should have occurred at the last move or not:

```
data ShouldHavePromotedCheck :: Position -> BoardDecorator -> Exp
   BoardDecorator
type instance Eval (ShouldHavePromotedCheck toPos boardDec)
    = ShouldHavePromotedCheck' toPos boardDec

type family ShouldHavePromotedCheck' (t :: Position) (b ::
   BoardDecorator) :: BoardDecorator where
    ShouldHavePromotedCheck' (At col Nat8) boardDec
        = If' (Eval (IsPieceAtWhichDec boardDec (At col Nat8) (IsPawn .&.
   HasTeam White)))
```

```
            (TE' (TL.Text ("Promotion should have occurred at: " ++
    TypeShow (At col Nat8) ++ ". Pawns must be promoted when they reach
    the opposite end of the board.")))
            (ID boardDec)
     ShouldHavePromotedCheck' (At col Nat1) boardDec
         = If' (Eval (IsPieceAtWhichDec boardDec (At col Nat1) (IsPawn .&.
    HasTeam Black)))
            (TE' (TL.Text ("Promotion should have occurred at: " ++
    TypeShow (At col Nat1) ++ ". Pawns must be promoted when they reach
    the opposite end of the board.")))
            (ID boardDec)
     ShouldHavePromotedCheck' _ boardDec = boardDec
```

It takes in the last position moved to, and the `BoardDecorator` (post-move). If the last move was to the 8th row or the first row, it checks if there is a Pawn at that position and if so, generates a type error (since such a Pawn should have promoted). Otherwise, it returns the `BoardDecorator` that was input.

Since this check occurs after every move, it enforces the rule that pawns must promote. If a pawn should fail to promote during a game of Chess, then a descriptive type error is output:

```
-- Below results in the following type error:
-- error: Promotion should have occurred at: A8. Pawns must be promoted
   when they reach the opposite end of the board.
failedToPromote = create
                      put _Wh _P at a7
                  startMoves
                      p a8
                  end
```

4.2 The EDSL

4.2.1 Minimum Viable Product

4.2.2 Flat Builders

A Continuation Passing Style [27] scheme forms the foundation for the EDSL, with inspiration taken from Dmitrij Szamozvancev's Flat Builders pattern [28]. The core idea is value transformation through a series of continuation function applications, until the final continuation function returns a value.

The type `Spec t` is the type of functions which take in a continuation to operate on a value of type `t`. For instance, a function with type `Int -> Spec Int` would take in an integer, and then a continuation to operate on that integer.

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

A function with type `Int -> Spec Int` can be represented with `Conv Int Int`—the `Conv s t` type represents functions which convert a value of type `s` to a value of type `Spec t`.

```
type Conv s t = s -> Spec t
```

Finally, the `Term t r` type ends the continuation stream by taking no continuations, and simply taking in a value of type `t` and returning a value of type `r`. If `t` and `r` are equal, then an example definition would be `id`.

```
type Term t r = t -> r
```

The above continuation types are combined with type-level rule checking, to create a Chess EDSL that operates through passing continuations. Using a combination of singletons, proxies, kind signatures, and type applications, the value-level Haskell code for the EDSL can have specific type variables and involve type family application. Essentially, the term-level EDSL can involve type-level rule checking.

The chess game starts with a `Proxy` value, its type parameterised with a `BoardDecorator` type. Continuations are applied, transforming that value, until the chess game ends or a rule is broken. Chess games begin with the board in a set configuration; and so a type `StartDec` of kind `BoardDecorator` was defined to contain all of this information.

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

The core continuations are named after the pieces, such as `pawn` and `king`. Each of them takes in an `SPosition`, a singleton version of `Position`. We define a new datatype, `MoveArgs`, in order to simplify the process of passing information between the continuations; `MoveArgs` can be partially applied, whereas a regular type family cannot. The `pawn` continuation is used below as an example; however, all of the piece continuations are similar, and only differ in the `PieceName` type passed to the continuation via `MoveArgs`.

```
data MoveArgs where
    MA :: BoardDecorator -> Position -> PieceName -> Position -> MoveArgs

pawn :: Proxy (b :: BoardDecorator) -> SPosition fromPos -> Spec (Proxy
   (MA b fromPos 'Pawn))
pawn (dec :: Proxy b) (from :: SPosition fromPos) cont = cont (Proxy @(MA
   b fromPos Pawn))
```

The next continuation, `to`, takes in another `SPosition` as well as the `MoveArgs`, performs the move computation, puts the resulting board decorator into a `Proxy` type, and passes that `Proxy` into the continuation given.

```
to :: Proxy (MA (b :: BoardDecorator) (fromPos :: Position) (n ::
   PieceName)) -> SPosition toPos -> Spec (Proxy (Eval
   (MoveWithStateCheck n fromPos toPos b)))
```

```
to (args :: Proxy (MA (b :: BoardDecorator) (fromPos :: Position) (n ::
    PieceName))) (to' :: SPosition toPos) cont = cont (Proxy @(Eval
    (MoveWithStateCheck n fromPos toPos b)))
```

The final relevant definition is of `end`, which ends the chess game, as well as the continuation stream.

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

Using the above continuations, we can lay out a chess game, move by move. However, the above continuations are Chesskell's long-form syntax; we discuss the shorter syntax in the next section. Consider the game expressed in the EDSL which we describe in section 4. It compiles successfully; but should Black attempt to move after checkmate, an error will be logged, since no moves by Black will be valid:

```
-- Below results in the following type error:
    -- * The Black King is in check after a Black move. This is not
    allowed.
    -- * When checking the inferred type
    --     game :: Data.Proxy.Proxy (TypeError ...)
game = chess
           pawn e2 to e4
           pawn f7 to f5
           queen d1 to f3
           pawn g7 to g5
           queen f3 to h5
           pawn g5 to g4
       end
```

Or, should White attempt an impossible move in the middle of the game, such as moving a Queen through another piece, a different type error will occur:

```
-- Below results in the following type error:
    -- * There is no valid move from D1 to D3.
    -- The Queen at D1 can move to: E2, F3, G4, H5
    -- * When checking the inferred type
    -- game :: Data.Proxy.Proxy (...)
game = chess
           pawn e2 to e4
           pawn f7 to f5
           queen d1 to d3
           pawn g7 to g5
           queen f3 to h5
       end
```

Shorthand syntax

While the above continuations are feature-complete, and allow the user to fully describe a game of Chess, the resulting notation is considerably more lengthy than Algebraic Notation or other comparable chess notations. As such, a shorthand syntax was introduced towards the end of development, to allow more concise description of Chess games.

Consider the original continuation for moving a Pawn, named `pawn`. To move using `pawn`, both the origin and destination squares are required, as well as the use of the continuation `to`. The new shorthand continuation is a single letter, `p`, which takes in the destination position and a continuation and performs the move immediately; calculating the origin square is left to the type-level model, via a type family `MoveTo`:

```
p :: Proxy (b :: BoardDecorator) -> SPosition toPos -> Spec (Proxy
   (MoveTo Pawn toPos b))
p (dec :: Proxy b) (to :: SPosition toPos) cont = cont (Proxy @(MoveTo
   Pawn toPos b))
```

`MoveTo` knows the destination square, and knows the `PieceName` of the piece that moves there. As such, it can calculate the origin square for that move using the piece type's movement rules in reverse. (Remember, for all pieces except Pawns and Kings, if the piece can move from a to b, then it can also move from b to a.)

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

FIGURE HERE PLEASE

Then, this list of squares is filtered based on whether there is a valid piece in the original `BoardDecorator` of the correct team in any of those squares. If the resulting filtered list has length 1 (i.e. it contains a single piece), then the position of that single piece extracted; otherwise, there are either no valid origin squares, or multiple valid origin squares, in which case the longer Chesskell syntax should be used.

As an example of the above case, consider the board state in figure 8.7. There are two bishops who could potentially move to square c5, and as such Chesskell will not be able to tell which bishop should move to that location, and will fail to compile with a type error:

```
-- Below results in the following type error:
    -- * There is not exactly one Bishop which can move to: C5.
    -- Consider using the long-form Chesskell syntax instead.
    -- * When checking the inferred type
    -- twoBishops :: Data.Proxy.Proxy (...)
twoBishops = create
                put _Wh _B at d6
                put _Wh _B at b4
             startMoves
```

```
                b c5
            end
```

Despite the late introduction of this shorthand syntax, it fits naturally into Chesskell as a form of Chess notation as well as a demonstration of type-level modelling. In fact, as we discuss in section 5.2.4, it resulted in unexpected performance improvements.

4.2.3 Setting up a board

While Chesskell was originally intended only to describe complete games of Chess, testing of the project was complicated through the fact that Chess games always start in a fixed configuration. To test something like Castling or Check required many moves before these game states became possible. Chesskell was extended to include arbitrary board creation syntax, to simplify this process.

Individual pieces can be placed down on the board using the `put` and `at` continuations, which are used in combination with singleton piece names, teams, and positions to modify a `BoardDecorator` type wrapped within the `Proxy` type constructor. Placing a Black Pawn on square a4 is done like so: `Put _Bl _P at a4`.

These commands are paired with a new beginning continuation, `create`, which passes a `Proxy JustKingsDec` value to a continuation, where `JustKingsDec` is a `BoardDecorator` type where the board only contains the Black and White King; nothing else. Along with a few other utility continuations to do things like set the last moved position, these continuations can be used to build custom boards. The below Chesskell snippet builds the board seen in figure 8.7:

```
create put _Wh _B at d6 put _Wh _B at b4 end
```

FEN board creation

However, Chesskell board creation syntax becomes overly lengthy when dealing with placing down a large number of pieces. It fails to be even half as concise as FEN board creation syntax (which we detail in section 3.1.4). As such, a new board creation syntax, based on FEN notation, was also implemented in Chesskell to speed up board creation.

Just like real FEN notation, this Chesskell variant of FEN notation (henceforth abbreviated to CFEN) specifies the chess board line by line. A new data type, named `Fen`, is used to encapsulate the idea of a row of 8 items:

```
data Fen (n :: Nat) where
    FF  :: Fen Nat0
    F1  :: Fen n -> Fen (S n)
    F2  :: Fen n -> Fen (S (S n))
    F3  :: Fen n -> Fen (S (S (S n)))
```

```
    F4  :: Fen n -> Fen (S (S (S (S n))))
    F5  :: Fen n -> Fen (S (S (S (S (S n)))))
    F6  :: Fen n -> Fen (S (S (S (S (S (S n))))))
    F7  :: Fen n -> Fen (S (S (S (S (S (S (S n)))))))
    F8  :: Fen Nat8
    Pw  :: Fen n -> Fen (S n)
    Nw  :: Fen n -> Fen (S n)
    Qw  :: Fen n -> Fen (S n)
    Kw  :: Fen n -> Fen (S n)
    Bw  :: Fen n -> Fen (S n)
    Rw  :: Fen n -> Fen (S n)
    Pb  :: Fen n -> Fen (S n)
    Nb  :: Fen n -> Fen (S n)
    Qb  :: Fen n -> Fen (S n)
    Kb  :: Fen n -> Fen (S n)
    Bb  :: Fen n -> Fen (S n)
    Rb  :: Fen n -> Fen (S n)
```

Essentially, the `Fen` type is used to create a type-level stack of elements; for example, `F5 (Nw (Qw (F1 FF)))`. Each of these `Fen` type constructors can be pattern-matched on to create a list (or vector) of items.

In fact, the type family `FenToRow` does exactly that; given a type of kind `Fen n`, it outputs a `Vec n (Maybe Piece)`, essentially transforming it into the row of a board. It also takes another `Nat`, so that each piece's position can be set correctly:

```
type family FenToRow (f :: Fen Eight) (r :: Nat) :: Row where
    FenToRow x r = FenHelper (FenReverse' x) r A

type family FenHelper (f :: Fen n) (r :: Nat) (c :: Column) :: Vec n
   (Maybe Piece) where
    FenHelper FF       row col = VEnd
    FenHelper F8       row col = EmptyRow
    FenHelper (F1 fen) row col = Nothing :-> FenHelper fen row (R col)
    FenHelper (F2 fen) row col = Nothing :-> Nothing :-> FenHelper fen
   row (R (R col))
    FenHelper (F3 fen) row col = Nothing :-> Nothing :-> Nothing :->
   FenHelper fen row (R (R (R col)))
    FenHelper (F4 fen) row col = Nothing :-> Nothing :-> Nothing :->
   Nothing :-> FenHelper fen row (R (R (R (R col))))
    FenHelper (F5 fen) row col = Nothing :-> Nothing :-> Nothing :->
   Nothing :-> Nothing :-> FenHelper fen row (R (R (R (R (R col)))))
    FenHelper (F6 fen) row col = Nothing :-> Nothing :-> Nothing :->
   Nothing :-> Nothing :-> Nothing :-> FenHelper fen row (R (R (R (R
   (R col))))))
    FenHelper (F7 fen) row col = Nothing :-> Nothing :-> Nothing :->
   Nothing :-> Nothing :-> Nothing :-> Nothing :-> FenHelper fen row (R
   (R (R (R (R (R col)))))))
    FenHelper (Pw fen) row col = Just (MkPiece White Pawn (Info Z (At col
   row) False)) :-> FenHelper fen row (R col)
    FenHelper (Nw fen) row col = Just (MkPiece White Knight (Info Z (At
```

```
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Qw fen) row col = Just (MkPiece White Queen (Info Z (At
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Kw fen) row col = Just (MkPiece White King (Info Z (At col
    row) False)) :-> FenHelper fen row (R col)
     FenHelper (Bw fen) row col = Just (MkPiece White Bishop (Info Z (At
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Rw fen) row col = Just (MkPiece White Rook (Info Z (At col
    row) False)) :-> FenHelper fen row (R col)
     FenHelper (Pb fen) row col = Just (MkPiece Black Pawn (Info Z (At col
    row) False)) :-> FenHelper fen row (R col)
     FenHelper (Nb fen) row col = Just (MkPiece Black Knight (Info Z (At
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Qb fen) row col = Just (MkPiece Black Queen (Info Z (At
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Kb fen) row col = Just (MkPiece Black King (Info Z (At col
    row) False)) :-> FenHelper fen row (R col)
     FenHelper (Bb fen) row col = Just (MkPiece Black Bishop (Info Z (At
    col row) False)) :-> FenHelper fen row (R col)
     FenHelper (Rb fen) row col = Just (MkPiece Black Rook (Info Z (At col
    row) False)) :-> FenHelper fen row (R col)
```

Predictably, `FenReverse'` reverses a `Fen` stack, to ensure that the pieces are in the correct order when they are placed down.

The end user requires a way to create these stacks of `Fen` types. A continuation is defined for each type constructor of `Fen`, with a few example definitions below:

```
fn7 :: (Proxy (b :: Fen n)) -> Spec (Proxy (F7 b))
fn7 (Proxy :: Proxy (b :: Fen n)) cont = cont (Proxy @(F7 b))

fn8 :: Term (Proxy (b :: Fen Nat8)) (Proxy (b :: Fen Nat8))
fn8 = id

wP :: (Proxy (b :: Fen n)) -> Spec (Proxy (Pw b))
wP (Proxy :: Proxy (b :: Fen n)) cont = cont (Proxy @(Pw b))
```

`fn0` and `ff` are delimiters; their definition is different since they are intended to book-end each use of these FEN continuations:

```
ff :: Spec (Proxy FF)
ff cont = cont (Proxy @FF)

fn0 :: Term (Proxy (b :: Fen n)) (Proxy (b :: Fen n))
fn0 = id
```

Stacks of `Fen` types can now be created like so: `ff fn1 wQ wN fn5 fn0`. Now that these stacks can be created, there must be a way to use them. A series of continuations for setting the row of the board with CFEN notation, named `fen1`, `fen2`, ..., `fen8`, are defined and make use of `FenToRow`. The definition of `fen5` is given below as an

example, but the only differences between the continuations are the rows they operate on:

```
fen5 :: (Proxy (b :: BoardDecorator)) -> (Proxy (f :: Fen Eight)) -> Spec
   (Proxy (SetRowDec' b Nat5 (FenToRow f Nat5)))
fen5 (Proxy :: Proxy (b :: BoardDecorator)) (Proxy :: Proxy (f :: Fen n))
   cont = cont (Proxy @(SetRowDec' b Nat5 (FenToRow f Nat5)))
```

Note that these stacks of `Fen` types must be of length 8 (i.e. they must have kind `Fen Eight`) to be accepted by the continuations, ensuring that the user cannot create rows of the wrong size. To put it all together, these commands can be used together to create a board row-by-row. Below is the Chesskell FEN notation for creation of the board seen in figure 8.6, which would be excessively lengthy in the base Chesskell board creation notation:

```
fenBoard = create
               fen1 (ff bR bN bB bQ bK bB bN bR fn0)
               fen2 (ff bP bP bP bP bP bP bP bP fn0)
               fen3 (fn8)
               fen4 (fn8)
               fen5 (ff fn4 wP fn3 fn0)
               fen6 (fn8)
               fen7 (ff wP wP wP wP fn1 wP wP wP fn0)
               fen8 (ff wR wN wB wQ wK wB wN wR fn0)
           end
```

5 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

5.3.1 Captures

5.3.2 Castling

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.

[26] H. Mu and S. Jiang, "Design patterns in software development," in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, pp. 322–325, 2011.

[27] G. J. Sussman and G. L. Steele, "Scheme: A interpreter for extended lambda calculus," *Higher-Order and Symbolic Computation*, vol. 11, no. 4, pp. 405–439, 1998.

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

8 Appendix

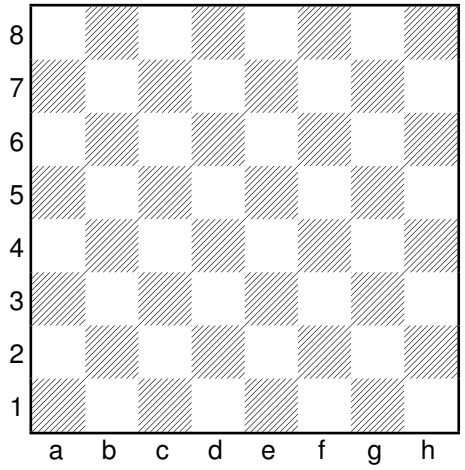

Figure 8.1: An empty chess board.

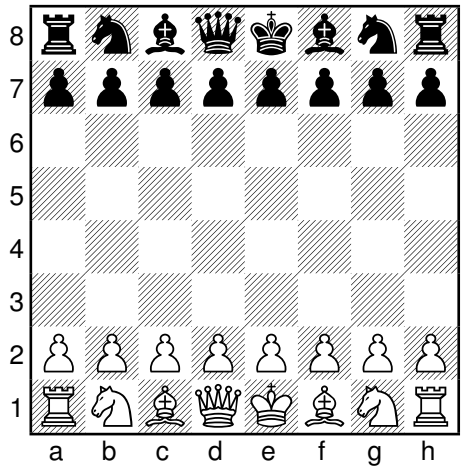

Figure 8.2: An standard chess board where all pieces are in their starting position.

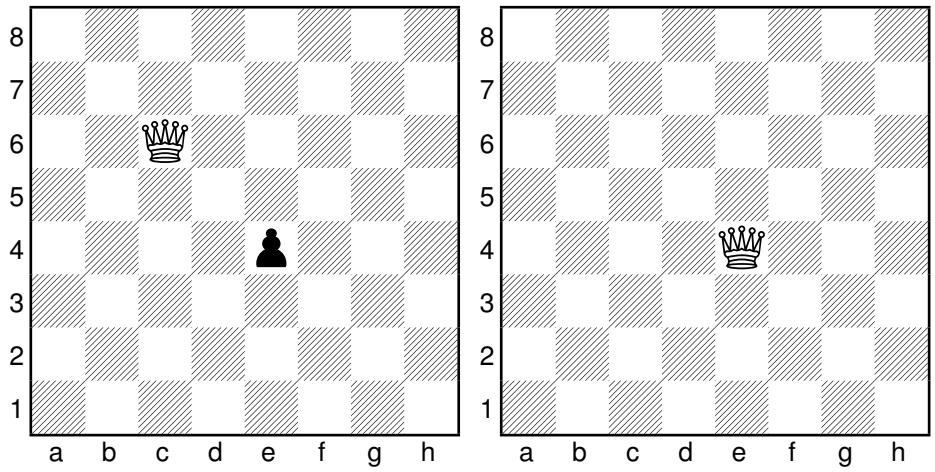

Figure 8.3: The White Queen captures a Black Pawn by moving to its position, and removing it from play.

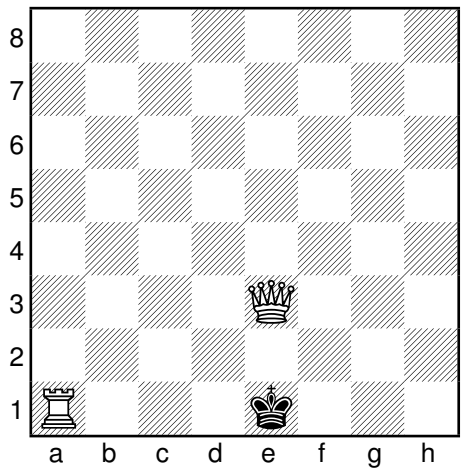

Figure 8.4: The White Queen and White Rook place the Black King into checkmate.

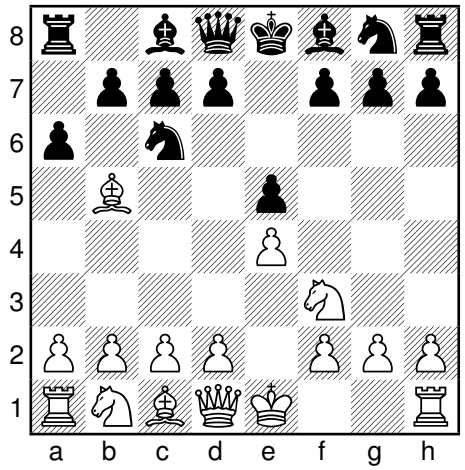

Figure 8.5: The position of the board after: 1.e4 e5 2. Nf3 Nc6 3.Bb5 a6

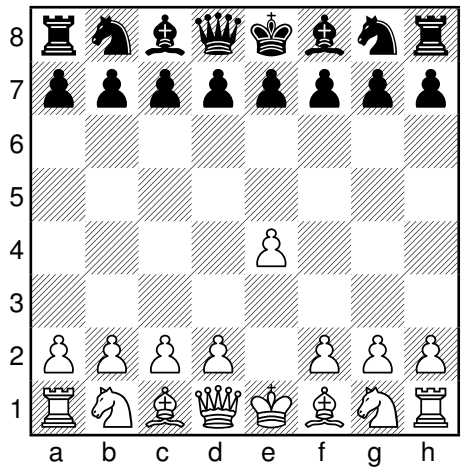

Figure 8.6: The board created with: (rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1)

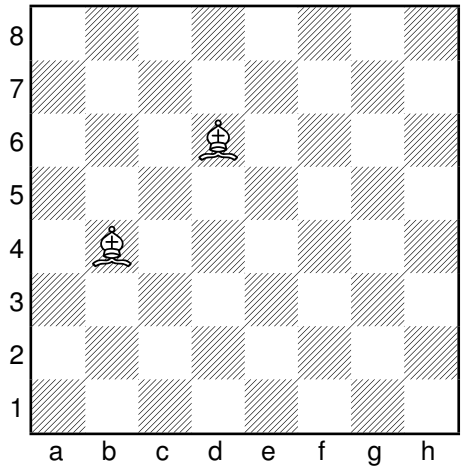

Figure 8.7: A board where two White Bishops can move to c5.