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

University of Warwick Dissertation

Toby Bailey

January 29, 2021

Contents

Abstract

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

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

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

Part I.

Introduction

1. Motivation

In 2021, video games are more popular than ever. In the US alone, a 2020 ESA report[1] estimated that there were more than 214 million individuals who play games. Considering this, it's surprising how many games are released with major bugs in their software—some of which end up being so notable that news and footage of them appear on mainstream media[2].

As programming languages have evolved, many have begun to address more errors at compile time. Features similar to optional types have been added to languages such as Java[3] and C#[4], and languages like Rust have pioneered ways of safely handling dynamic allocation through ownership types[5]. Many compilers now force the developer to handle classes of errors that previously could only be encountered at runtime, such as null pointer exceptions.

However, catching logical errors in imperative languages is almost always done during execution. Many software systems use runtime features such as exceptions to discover and deal with errors and misuse of APIs. Enforcement of invariants and rules is typically dynamic; if a check fails, an exception is thrown and potentially handled. However, if a programmer forgets to implement such a check, the behaviour is unpredictable. A 2007 study [1] on Java and .NET codebases indicates that exceptions are rarely used to recover from errors, and a 2016 analysis of Java codebases [2] 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 [3], using *type families* [4] [5] 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 [6]. 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

[1] `https://www.theesa.com/wp-content/uploads/2020/07/Final-Edited-2020-ESA_Essential_facts.pdf`

[2] `https://www.bbc.co.uk/news/technology-50156033`

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html`

[4] `https://docs.microsoft.com/en-us/dotnet/csharp/nullable-references`

[5] `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`

in complex applications. 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.

2. History

Programming languages have *type systems* for the main purpose of avoiding errors [7]. 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.

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

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

[1] For instance, dogs can bark, but the number 3 cannot.

3. Objectives

The objective of this project is to develop an EDSL, nicknamed Chesskell, with which to express games of Chess. 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 simulation 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[1]. 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.

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

Part II.

Background

4. Related Work

4.1. Type-level Rule Checking

4.2. Embedded Domain-Specific Languages

5. The Basics of Chess

6. Programming for Chess

6.1. Chess Programs

6.2. Chess Data Structures

6.3. Why is Chesskell different?

Part III.

Design

7. Type-level Programming

7.1. Type Families

7.2. First-Class Families

7.3. Type-Level Data Structures

8. Giving Runtime Information to the Compiler

9. Development Approach

9.1. Methodology

9.2. Testing

Part IV.

Implementation

10. Type-Level Chess

10.1. Chess Types and Kinds

10.1.1. The Pieces

10.1.2. The Board

10.1.3. Miscellaneous Types

10.2. Chess Rules

10.2.1. Movement Rules

10.2.2. Attack/Capture Rules

10.2.3. Checking For Violations

10.2.4. Exceptions

Castling

Pawn Capture and En Passant

11. The EDSL

11.1. Minimum Viable Product

11.2. Flat Builders

11.3. Moving the pieces

11.4. Setting up a board

12. Testing

12.1. Type-level Unit Testing

12.2. Testing Chesskell Games

Part V.

Evaluation

13. Difficulties

13.1. Session-typed Chesskell

13.2. Compile Time and Memory Usage

13.2.1. Optimisation Attempts

Board Decorators

Finger Trees

13.2.2. GHC Bug Report

13.3. Descriptive Error Messages

13.3.1. Move Number

14. Chesskell EDSL vs Other Chess Notations

Part VI.

Conclusion

15. Results and Accomplishments

16. Future Work

Part VII.

Appendix

17. Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

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

[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.