Contents

Abstract

...

Chapter 1

Introduction

I Background information

…

II Problem statement

…

A. *Subsection1*

…

B. *Subsection2*

…

Fig. 1. Inno logo

III Overview

Our work focuses ...

In this thesis, Ch. 2 ... Then, Ch. 3 ... Next, Ch. 4 ... Following that, Ch. 5 ... Finally, Ch. ...

Chapter 2

Literature Review

Our initial goal was to build a language server and an interpreter for a typed lambda calculus using the Free Foil [1] representation for its abstract syntax tree (AST). In this representation, the AST has two types of nodes: one for scoped variables and another for all other expressions, potentially under a binder. Such expressions can be `if`-expressions and function applications. The Free Foil representation can handle capture-avoiding substitution while a user only needs to specify how to handle non-variable expressions. We planned to use the Free Foil implementation provided by the `free-foil` Haskell library [2].

Later, we hypothesized on extending our language and discovered several limitations of the Free Foil library.

A. Free Foil limitations

First, the library provides convenience functions for generating the scope-safe AST data type from data types produced by the BNFC parser generator, but these functions have limitations.

One problem is that these functions only support terms, variable identifiers,

scoped terms, patterns Control.Monad.Free.Foil.TH.Convert.

Another problem is that in some cases, due to limitations of the BNFC, it is necessary to introduce dummy nodes. For example, a floating number should be parsed as a string of characters, and then a new node should be created with a floating number instead of the string. Currently, some modifications to convenience functions are necessary to to support specifying which data type constructors to omit during generation.

Both of these problems can be avoided by manually writing the AST data types, but then the whole point of convenience functions is lost.

Second, currently, the library requires that mutually recursive types in the AST be combined into a single data type. This limitation may be inconvenient when developing languages where nodes for expressions and expression types are more convenient to represent via different Haskell data types, e.g., languages without dependent types. The library author states that the library may support mutually recursive types.

Third, we realized that the Free Foil approach would not work very conveniently in presence of modules. In a language with modules, relating variable usage and declaration sites may require performing multi-phased typechecking [3]. So, after parsing, language implementors would need to construct an AST without binders, then type check it, and then construct an AST with resolved binders. However, a Free Foil AST without binders does not provide any considerable benefits over other AST representations.

These limitations led us to considering other AST representations.

B. *Alternative AST representations*

First of all, we did not want to reuse the data type produced by the BNFC parser generator because it contained unnecessary data constructors as mentioned in Sec. 2.0.1.

Since we wanted to implement a language server, we needed to annotate some nodes in the AST with additional information. That information could be corresponding syntax construct positions after parsing and types after type checking.

We decided to look at the AST data types used in the GHC because it was a working compiler at hand.

Studying the compiler source code led us to learning about the Trees that Grow approach that lets one extend particular constructors with additional fields and extend a type with additional constructors.

Unlike Free Foil, this approach did not prohibit mutually recursive types.

Our next step was to choose a type system for our language.

C. *Type system*

Initially, we studied [4] to learn about bidirectional type systems.

The work was not easy to follow and it was not obvious how to extend the described systems to support features like type classes and type families that are available in the GHC.

We checked the [5] to learn about a specific bidirectional type system for higher-rank polymorphism.

That work was not approachable either, but referenced a more understandable earlier work about a type system with similar properties [6] authored by the creator of the GHC.

[6] provided an implementation of the algorithm.

That fact was convenient because it allowed us to better understand an algorithm similar to the one used in GHC.

Our next step was to revive the code and use a new AST representation.

Chapter 3

Design and Implementation

Chapter 4

Evaluation and Discussion

This chapter analyzes the research results.

Sec. 4.1 presents the main findings that are connected with the research purpose. Sec. 4.2 interprets how the research results support these findings. Sec. 4.3 contrasts my findings with the results of the past researches. Sec. 4.4 describes the limitations of my research. Sec. 4.5 suggests the possible applications of my research findings.

I Key findings

...

II Results and findings

...

III Previous research

...

A. Subsection

...

IV Limitations

...

V Potential applications

...

Chapter 5

Conclusion

...

Bibliography cited

[1] N. Kudasov, R. Shakirova, E. Shalagin, and K. Tyulebaeva, *Free foil: Generating efficient and scope-safe abstract syntax*, May 26, 2024. DOI: 10.48550/arXiv.2405.16384. arXiv: 2405.16384. Accessed: Nov. 11, 2024. [Online]. Available: http://arxiv.org/abs/2405.16384.

[2] "Free-foil," Hackage, Accessed: Apr. 25, 2025. [Online]. Available: https://hackage.haskell.org/package/free-foil.

[3] C. B. Poulsen, A. Zwaan, and P. Hübner, "A monadic framework for name resolution in multi-phased type checkers," 2023.

[4] J. Dunfield and N. Krishnaswami, *Bidirectional typing*, Nov. 14, 2020. DOI: 10.48550/arXiv.1908.05839. arXiv: 1908.05839. Accessed: Nov. 12, 2024. [Online]. Available: http://arxiv.org/abs/1908.05839.

[5] J. Dunfield and N. R. Krishnaswami, *Complete and easy bidirectional typechecking for higher-rank polymorphism*, Aug. 22, 2020. DOI: 10.48550/arXiv.1306.6032. arXiv: 1306.6032[cs]. Accessed: Apr. 2, 2025. [Online]. Available: http://arxiv.org/abs/1306.6032.

[6] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *J. Funct. Prog.*, vol. 17, no. 1,

pp. 1–82, Jan. 2007, ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796806006034. Accessed: Apr. 3, 2025. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0956796806006034/type/journal_article.

Appendix A

Extra Stuff

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Appendix B

Even More Extra Stuff

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.