Contents

Chapter 1

Introduction

> Any sufficiently advanced technology
> is indistinguishable from magic.
>
> *Arthur C. Clarke*

For us, that "sufficiently advanced technology" was the Glasgow Haskell Compiler (GHC) [1]. Although we had been programming in Haskell for several years before we started working on the thesis, we somehow were not curious enough to learn about the internals of the Haskell compiler.

Luckily, the compiler had open source code, its implementation was based on scientific publications, the project wiki and online presentations about the compiler were insightful, and there were online communities where the compiler developers and other knowledgeable people were ready to answer our questions.

To improve our understanding of the compiler internals and make an excuse for writing some more Haskell, we decided to implement a small extensible typed functional language using approaches similar to those employed in GHC.

These approaches included using a fancy abstract syntax tree (AST) representation, support for higher-rank types, a bidirectional type inference algorithm,

collecting and then solving type constraints, translation to a core language based on System F, and interpretation of that language.

Additionally, we decided to study available options for the AST representation and the the type inference engine to educate ourselves and possibly use in our project. For example, we learned about the Free Foil approach [2] that enabled type-safe capture-avoiding substitution and could be used to represent types or the core language.

As we wanted to iterate on the language syntax quickly, we decided to use the BNFC parser generator [3] instead of the combination of Happy and Alex used in the GHC [4].

At the end, we expected to obtain a parser, a type inference engine, an interpreter, a language server, and a VS Code extension for our language.

We could not find on GitHub any well-documented implementation of a very simple language featuring predicative higher-rank polymorphism, having the mentioned components, and resembling the GHC. In our work, we attempted to partially close this gap by documenting the considered options and explaining our final implementation. The implementation [5] is available on GitHub under the MIT license.

I Overview

The Ch. 2 reviews several AST representations and type inference algorithms including the ones that we chose for our implementation. Then, Ch. 3 explains the theoretical basis and details of the implementation of our language. Next, Ch. 4 summarizes the obtained results and describes the development experience. Following that, Ch. 5 mentions possible directions for future work. Finally, Ch. A provides a number of snippets mentioned in our work.

Chapter 2

Literature Review

We studied several abstract syntax tree representations (Sec. 2.2) and type inference algorithms (Sec. 2.5) to understand which ones could be used in different parts of our implementation.

I Technical constraints

- We thought about extending our language to support modules, integer and floating-point numbers, and record types in future.
- We wanted to be able to freely use mutually recursive data types for different categories of AST nodes. For example, such categories could be modules and statements. A module could contain a number of statements and a statement could introduce a module.
- For the language server, we needed to annotate some of the AST nodes with additional information such as locations of the source code that corresponded to these nodes and types of expressions represented by these node subtrees.

- We used the BNFC parser generator that generated an AST data type parameterised by an annotation type variable. After parsing, the annotation of almost each AST node contained the position of a source code span parsed to produce that node.

II AST representations

For the AST representation, we could use the data types produced by the parser generator BNFC (Sec. 2.2.1), `hypertypes` (Sec. 2.2.2) and alternative representations described in that project, `compdata` (Sec. 2.2.3), the Free Foil (Sec. 2.4), or the Trees that grow approach (Sec. 2.4.1).

A. BNFC AST

The BNFC-generated parsers did not support parsing signed integer numbers out of the box. Our workaround was to define a `token`, then use that `token` to parse code and construct a node containing a string in the correct format, then post-process the parsed AST to replace nodes containing such raw values with `internal` (not parsable) nodes containing numbers.

```
token IntegerSigned  ('-'? digit+) ;
LitIntegerRaw.       Object ::= IntegerSigned ;
internal LitInteger. Object ::= IntegerSigned ;
```

It was inconvenient though that both variants of nodes were in the AST and we would have to pattern-match on the `LitIntegerRaw` even if that variant of node was completely unnecessary after the post-processing.

B. `hypertypes`

The `hypertypes` package [6] let users construct expressions from individual components like in the Data types à la carte [7] approach. These components could be mutually recursive types like in `multirec` [8] and could be processed via type classes. The package description explained the limitations of several previous approaches.

The package provided primitives for annotating nodes (`Hyper.Ann`), for constructing typed lambda calculus expressions (`Hyper.Syntax`), and for unification (Hyper.Unify). The `Hyper.Diff` module showed how to annotate trees and the `TypeLang` module provided an implementation of type inference for a language with row-types using the primitives provided by the package.

C. `compdata`

Likewise, the `compdata` package [9] supported mutually recursive data types, including GADTS (`Data.Comp.Multi`). Provided examples showed (example1, example2, example3 related to this issue) how to use annotated ASTs.

III Stitch

Eisenberg explored an implementation of a simply typed λ-calculus with a non-typechecked AST with type-level de Bruijn indices that enforced the construction of only well-scoped terms during parsing and a type-checked AST indexed with type-level contexts and node types. The implementation uses lots of Haskell extensions and can be a good playground for studying them in action.

IV Free Foil

The Free Foil [2] approach by Kudasov et al. implemented in the [11] package allowed for constructing an AST where nodes were indexed with a phantom type variable denoting the scope. Each node represented either a variable or any other language construct, potentially scoped under a (single) binder that extended the scope. The approach enabled type-safe capture-avoiding substitution of variables for expressions. Additionally, it allowed to define generic recursive AST processing functions that could be used for any AST where nodes were constructed from a user-supplied type (`sig`) that had necessary type class instances such as `Bifunctor`.

The example in Control.Monad.Free.Foil.Example showed a definition of an AST for an untyped lambda calculus. The pattern synonym `LamE` was used to construct expressions under a binder.

The Free Foil approach had two limitations that could be inconvenient given our Technical constraints (Sec. 2.1).

First, the last library version required that mutually recursive types in the AST be combined into a single data type. The library author stated that the library could theoretically support mutually recursive types if another representation were used.

Second, it could increase the AST size if our language had modules, or, more generally, supported non-lexical scoping. In a language with modules, relating the variable declaration and usage sites may require performing multi-phased type checking, e.g., using the scope graphs approach [12]. If we wanted to use binders to track the declaration sites, after parsing, we would need to construct an AST with nodes for modules and without binders, then type check that AST, and then construct an AST with resolved binders. In the subtree of each node that denoted

a module import, we would need to create nodes that would introduce binders for visible variables from that module. At the same time, we would still need to track for each binder the module that the variable came from.

A. Trees That Grow

The GHC used a variant of the Trees That Grow [13] approach (HsSyn, Trees that grow guidance).

This approach suggests to parameterise a data type with a type variable and use open type families applied to that type variable instead of concrete types for constructor fields. This way, it is possible to specify a different set type family instances and corresponding field types for each instantiation of the parameter. For example, in GHC, the AST is parameterised by the compilation phase (parsed, renamed, typechecked), and the type families are used to disable certain constructors and choose which nodes should have annotations of particular types.

The data type should also have an additional constructor with a field specified in the similar manner using a type family. Then, it will be possible to "add constructors" to the data type by resolving the type of that field to another data type and providing pattern synonyms that would make constructors of another type wrapped in the constructor of the additional field look as if they belong to the initial data type. A similar approach could be used with constructors of the initial data type to "add" fields to these constructors if they provided an additional field represented as an application of a type family.

Unlike Free Foil, this approach supported mutually recursive types and was similar to using an ordinary data type parameterised by a type variable except one had to define quite a lot of type family instances for constructor fields.

V Type inference algorithm

A. GHC

The current GHC type inference engine has tens of thousands lines of code. It has been incrementally developed over years to add new extensions to the Haskell type system. Some of these extensions had accompanying scientific publications. One of such extensions was `RankNTypes` that enabled arbitrary-rank predicative polymorphism. The extension was based on [14] that described a bidirectional type inference algorithm for a type system with arbitrary-rank types. The Technical Appendix [15] provided the proofs of theorems mentioned in the paper, including the proofs of completeness and soundness, but not of the attached Haskell implementation.

B. New algorithms

[14] was published around 20 years ago, and since then, multiple new type inference algorithms outside of the work on the GHC had been published. Moreover, some of them were implemented in Haskell [16] [17] [18].

The following sections

C. Bidirectional typing

There are two modes in bidirectional typing - the inference mode that helps reduce the number of type annotations required from the programmer and propagates the available type information, and the checking mode that can type expressions for which a type could not be inferred otherwise [19].

In [20], Dunfield and Krishnaswami provided an arguably simple bidirec-

tional type inference algorithm for a system with higher-rank predicative polymorphism. Their algorithm had similar properties [20, Fig. 15] as the one presented in [14, Sec. 6], namely, it abided the η-law [21, Ch. 4] and was sound and complete wrt. the System F [21, Ch. 8]. The authors argued that using ordered contexts, existential variables, and precise scoping rules in their work were a better fit from the type-theoretic point of view than using the “bag of constraints" approach, unification variables, and skolemization that were described, e.g., in [14].

In a later work [22], Dunfield and Krishnaswami built on [20] and described the type inference algorithm for a significantly extended language that featured existential quantification, sums, products, pattern matching, etc. For their algorithm and proofs, they used a desugared version [22, Fig. 11] of the surface language [22, Fig. 1].

Additionally, Dunfield and Krishnaswami extensively surveyed [19] the works on bidirectional typing and supplied a guide to creating new programmer-friendly bidirectional type inference algorithms.

D. Modifications of bidirectional typing

Xie [23] reviews [20] (Sec. 2.3), suggests using the application mode in addition to inference and checking modes (Sec. 3), provides a novel algorithm for kind inference (Sec. 7), and relates it to the current GHC implementation (Sec. 8.6), noting possible points for improvement (Appendix Sec. C).

Xue and Oliveira [24] explain the limitations of bidirectional typing (Sec 2.5) and generalize bidirectional typing to contextual typing by propagating any necessary contextual information instead of just type information and replacing the two modes (inference and checking) with counters that track the amount of the contextual information to be propagated. Their approach allows to specify exact

places in the code where a programmer must provide annotations.

E. Beyond bidirectional typing

Parreaux et al. suggest a novel non-bidirectional type inference algorithm SuperF that supports first-class (i.e., impredicative) higher-rank polymorphism. Impredicative polymorphism allows instantiation of type variables with polytypes while predicative allows only monotypes [14, Sec 3.4]. The algorithm infers a type for each subterm and then checks against user annotations written in System F syntax. The authors claim that subtype inference used in their approach suits much better for implementing first-class polymorphism than first-order unification-based approaches. As a demonstration of the capabilities of the algorithm, the authors show that it types almost any term even without type annotations (Sec 5.4, Sec 5.5).

Chapter 3

Design and Implementation

I Architecture

A. Static view

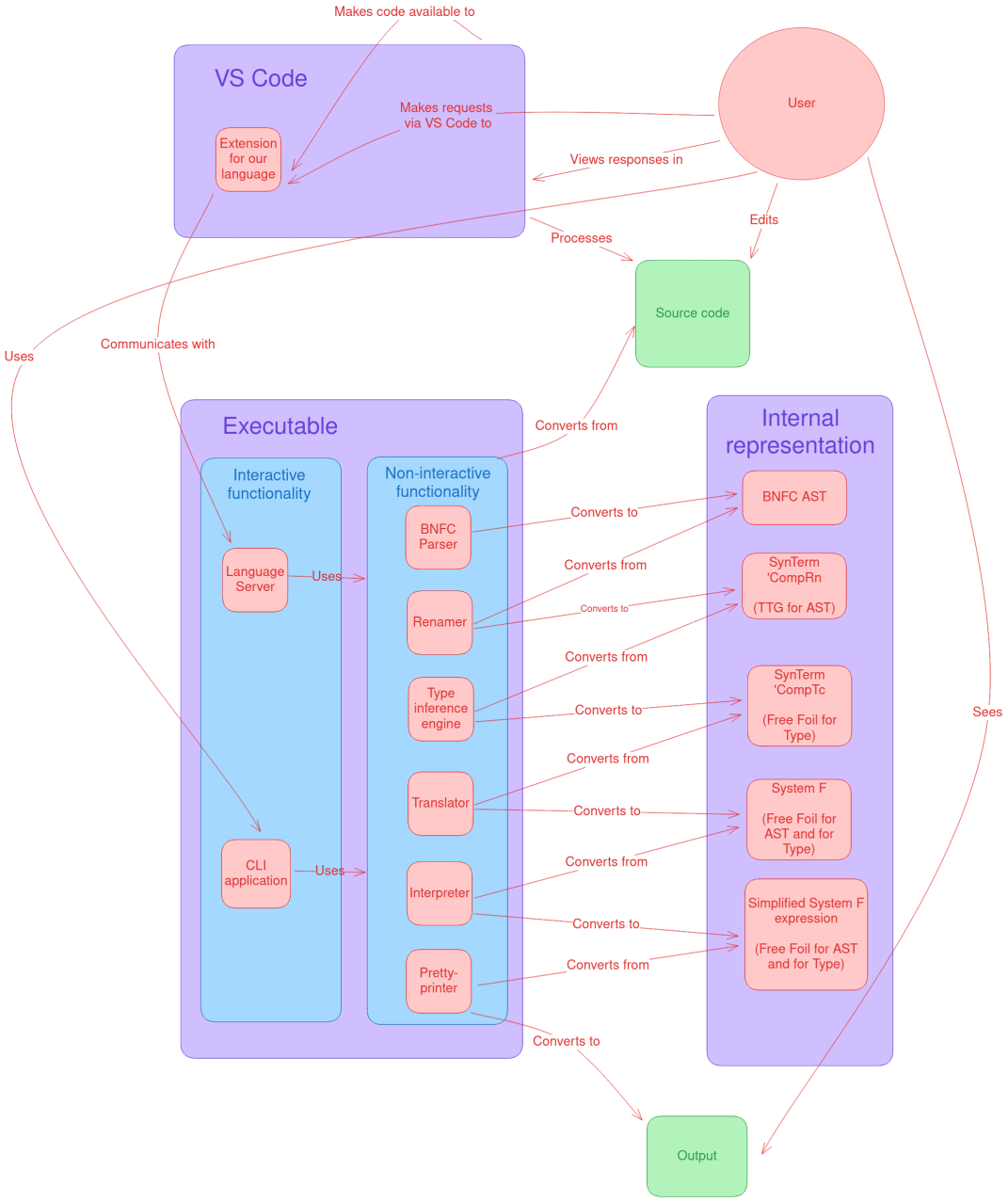

Fig. 1. Static view

B. Use Case: Interpreter is called.

1) Parsing: A BNFC-generated parser reads the source code written in our language and produces a raw BNFC AST with positions of tokens.

2) Renaming: The raw BNFC AST is then converted to the Trees That Grow representation `SynTerm 'CompRn`. During conversion, positions are copied to annotations and each variable in the program gets a unique identifier under condition. Same variables appearing in multiple places in the program get the same identifiers.

3) Typing: The typing algorithm runs. It uses the Free Foil representation of Core Types. The typing algorithm annotates the tree with fully instantiated (not having metavariables) types producing `SynTerm 'CompTc` or throwing an error.

4) Translation to System F: The `SynTerm 'CompTc` is converted to the System F syntax using the Free Foil representation.

5) Interpretation: The interpreter evaluates the System F code.

6) Output: The program pretty-prints the resulting expression.

C. Use case: user queries a type of a variable.

1) Start language server: VS Code extension starts the language server if not started. VS Code extension makes the language server read the code in the current file.

2) Query: VS Code extension passes a position to the language server and asks for the type of the variable at that position.

II AST

We used the Trees That Grow approach [13] for the AST representation.

In GHC, some fields are just types constructed using the index parameter. In contrary, we used type family applications to the index parameter in all fields of the AST to make the AST more flexible. Additionally, we named the type families and constructors consistently to improve code navigation.

```
-- GHC

-- Language/Haskell/Syntax/Expr.hs

data HsExpr p
  = HsVar      (XVar p)
               (LIdP p)
    ...

-- Language/Haskell/Syntax/Extension.hs

type LIdP p = XRec p (IdP p)

-- Ours

-- Language/STLC/Typing/Jones2007/BasicTypes.hs
```

```
data SynTerm x
    = -- | variables
      SynTerm'Var (XSynTerm'Var' x) (XSynTerm'Var x)
    ...

-- The Name already contains an SrcSpan, so it doesn't need
type instance XSynTerm'Var x = Name
```

Like in GHC, we had separate data types that represented syntactic types and types used during typing.

```
-- Ours

-- Language/STLC/Typing/Jones2007/BasicTypes.hs

data SynType x
  = -- | Type variable
    SynType'Var (XSynType'Var' x) (XSynType'Var x)
    ...

data Type
  = -- | Vanilla type variable.
    Type'Var Var
  ...
```

TODO use the Free Foil representation for Type TODO Free foil - try to marry with indices assigned by the Tc monad TODO write about the type system.

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] "The glasgow haskell compiler," Accessed: Apr. 27, 2025. [Online]. Available: https://www.haskell.org/ghc/.

[2] 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.

[3] "Welcome to BNFCs documentation! BNFC 2.9.6 documentation," Accessed: Apr. 26, 2025. [Online]. Available: https://bnfc.digitalgrammars.com/.

[4] "Glasgow haskell compiler / GHC ů GitLab," GitLab, Accessed: Apr. 27, 2025. [Online]. Available: https://gitlab.haskell.org/ghc/ghc.

[5] D. Danko, *Deemp/higher-rank-free-foil*, original-date: 2024-11-11T16:24:27Z, Apr. 27, 2025. Accessed: Apr. 27, 2025. [Online]. Available: https://github.com/deemp/higher-rank-free-foil.

[6] "Hypertypes," Hackage, Accessed: Apr. 27, 2025. [Online]. Available: https://hackage.haskell.org/package/hypertypes.

[7] W. Swierstra, "Data types à la carte," *Journal of Functional Programming*, vol. 18, no. 4, pp. 423–436, Jul. 2008, ISSN: 1469-7653, 0956-7968. DOI: 10.1017/S0956796808006758. Accessed: Apr. 27, 2025. [Online]. Available: https://www.cambridge.org/core/journals/journal-of-functional-programming/article/data-types-a-la-carte/14416CB20C4637164EA9F77097909409.

[8] "Multirec," Hackage, Accessed: Apr. 27, 2025. [Online]. Available: https://hackage.haskell.org/package/multirec.

[9] "Compdata," Hackage, Accessed: Apr. 27, 2025. [Online]. Available: https://hackage.haskell.org/package/compdata.

[10] R. A. Eisenberg, "Stitch: The sound type-indexed type checker (functional pearl)," in *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*, Virtual Event USA: ACM, Aug. 27, 2020, pp. 39–53, ISBN: 978-1-4503-8050-8. DOI: 10.1145/3406088.3409015. Accessed: Apr. 27, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3406088.3409015.

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

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

[13] S. Najd and S. P. Jones, *Trees that grow*, Oct. 15, 2016. DOI: 10.48550/arXiv.1610.04799. arXiv: 1610.04799[cs]. Accessed: Apr. 26, 2025. [Online]. Available: http://arxiv.org/abs/1610.04799.

[14] 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.

[15] D. Vytiniotis, S. Weirich, and S. Peyton-Jones, "Practical type inference for arbitrary-rank types - technical appendix," [Online]. Available: https://repository.upenn.edu/server/api/core/bitstreams/7c1dc678-93c6-4516-98fd-f82b384eb75d/content.

[16] A. Goldenberg, *Artem-goldenberg/BidirectionalSystem*, original-date: 2024-11-23T14:06:35Z, Jan. 23, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/Artem-Goldenberg/BidirectionalSystem.

[17] K. Choi, *Kwanghoon/bidi*, original-date: 2020-08-16T11:54:57Z, Jan. 3, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/kwanghoon/bidi.

[18] C. Chen, *Cu1ch3n/type-inference-zoo*, original-date: 2024-12-27T09:05:39Z, Apr. 26, 2025. Accessed: Apr. 28, 2025. [Online]. Available: https://github.com/cu1ch3n/type-inference-zoo.

[19] 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.

[20] 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.

[21] P. Selinger, *Lecture notes on the lambda calculus*, Dec. 26, 2013. DOI: 10.48550/arXiv.0804.3434. arXiv: 0804.3434[cs]. Accessed: Apr. 28, 2025. [Online]. Available: http://arxiv.org/abs/0804.3434.

[22] J. Dunfield and N. R. Krishnaswami, "Sound and complete bidirectional typechecking for higher-rank polymorphism with existentials and indexed types," *Proc. ACM Program. Lang.*, vol. 3, 9:1–9:28, POPL Jan. 2, 2019. DOI: 10.1145/3290322. Accessed: Apr. 3, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3290322.

[23] N. Xie, "Higher-rank polymorphism: Type inference and extensions," [Online]. Available: https://i.cs.hku.hk/~bruno/thesis/NingningXie.pdf.

[24] X. Xue and B. C. D. S. Oliveira, "Contextual typing," *Proc. ACM Program. Lang.*, vol. 8, pp. 880–908, ICFP Aug. 15, 2024, ISSN: 2475-1421. DOI: 10.1145/3674655. Accessed: Apr. 4, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3674655.

[25] L. Parreaux, A. Boruch-Gruszecki, A. Fan, and C. Y. Chau, "When subtyping constraints liberate: A novel type inference approach for first-class polymorphism," *Proc. ACM Program. Lang.*, vol. 8, pp. 1418–1450, POPL Jan. 5, 2024, ISSN: 2475-1421. DOI: 10.1145/3632890. Accessed:

Apr. 24, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3632890.

Appendix A

Extra code snippets

...