Contents

Chapter 1

Introduction

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

For us, the Glasgow Haskell Compiler (GHC) [1] represented that "sufficiently advanced technology." Despite programming in Haskell for several years prior to beginning this thesis, we had not yet explored the internal workings of its primary compiler.

Fortunately, GHC is open source, its implementation is grounded in scientific publications, and a wealth of resources — including the project wiki, online presentations, and supportive online communities with compiler developers and experts — facilitated our learning.

To deepen our understanding of GHC's internals and create a practical context for further Haskell programming, we decided to implement a small, extensible, typed functional language employing approaches similar to those used in GHC.

Key techniques adopted from GHC included utilizing a sophisticated abstract syntax tree (AST) representation, supporting parametric predicative higher-rank

polymorphism, implementing a bidirectional type inference algorithm, employing constraint generation and solving, translating the source language to a System F-based core language, and interpreting this core language.

Additionally, we investigated available options for the AST representation and type inference engine to inform our design choices. For example, we studied the Free Foil approach [2], which enabled type-safe capture-avoiding substitution and offered potential applications for representing types or the core language.

To facilitate rapid iteration on the language syntax, we chose the BNFC parser generator [3] rather than the combination of Happy and Alex employed by GHC [4].

The project aimed to produce a complete toolchain for our language, including a parser, a type inference engine, an interpreter, a language server, and a corresponding VS Code extension.

Our search on GitHub revealed a lack of well-documented, open-source implementations of simple languages that feature parametric predicative higher-rank polymorphism, include the aforementioned components, and closely mirror GHC's architecture. This work attempts to partially address this gap by documenting the design options considered and detailing our final implementation. The implementation [5] is available on GitHub under the MIT license.

I Overview

This thesis is structured as follows:

Ch. 2 reviews several abstract syntax tree (AST) representations and type inference algorithms, including those ultimately chosen for our implementation. Subsequently, Ch. 3 details the theoretical foundations and implementation specifics of our language. Ch. 4 evaluates the results obtained and discusses the development process and experience. Following this, Ch. 5 outlines potential directions for future work. Finally, Ch. A contains relevant code snippets referenced throughout the thesis.

Chapter 2

Literature Review

This chapter reviews several abstract syntax tree (AST) representations (Sec. 2.2) and type inference algorithms (Sec. 2.3). This review aimed to identify approaches suitable for different aspects of our implementation, considering the technical constraints outlined below.

I Technical Constraints

Our design and implementation choices were guided by the following technical constraints and requirements:

- **Extensibility:** We anticipated future extensions to the language, including support for modules, integer and floating-point numbers, and record types. The chosen representations should accommodate such additions gracefully.
- **Mutual Recursion:** A requirement was the ability to employ mutually recursive data types for different categories of AST nodes. For instance, categories like modules and statements might be mutually dependent: a module could contain statements, and a statement could potentially introduce a

nested module.

- **Annotation for Tooling:** The language server necessitated annotating AST nodes with auxiliary information. This includes the source code location (span) corresponding to each node and the inferred type of the expression represented by a node's subtree.

- **Parser Generator Integration:** We utilized the BNFC parser generator. BNFC produces an AST data type parameterized by an annotation type variable. Post-parsing, the annotation for most AST nodes includes the source code span from which the node was derived.

II AST Representations

Several AST representation techniques were considered, including the default data types generated by BNFC (Sec. 2.2.1), the `hypertypes` library (Sec. 2.2.2) and related approaches, the `compdata` library (Sec. 2.2.3), the Stitch representation (Sec. 2.2.4), the Free Foil technique (Sec. 2.2.5), and the Trees that Grow pattern (Sec. 2.2.6).

A. BNFC AST

The parsers generated by BNFC do not directly support parsing signed integer literals. A common workaround involves defining a custom `token` for the pattern, parsing it into a node containing the raw string representation, and subsequently post-processing the generated AST to convert these string-based nodes into nodes containing the actual numeric values using an `internal` (non-parsable) constructor.

```
token IntegerSigned ('-'? digit+) ;
-- Node holding the raw string
LitIntegerRaw. Object ::= IntegerSigned ;
-- Node holding the parsed Integer
internal LitInteger. Object ::= IntegerSigned ;
```

A drawback of this approach is the persistence of both node variants (`LitIntegerRaw` and `LitInteger`) within the AST data type definition. This requires handling the `LitIntegerRaw` case in pattern matching, even though it becomes redundant after the post-processing step.

B. hypertypes

The `hypertypes` package [6] enables the construction of expressions from individual components, reminiscent of the Data types à la carte approach [7]. These components can represent mutually recursive types, similar to `multirec` [8], and are processed using type classes. The package documentation discusses the limitations of several preceding approaches.

Key features include primitives for node annotation (`Hyper.Ann`), constructing typed lambda calculus expressions (`Hyper.Syntax`), and unification (`Hyper.Unify`). The `Hyper.Diff` module demonstrates tree annotation, while the `TypeLang` example provides a type inference implementation for a language with row-types, utilizing the package's primitives.

C. compdata

Similarly, the `compdata` package [9] supports mutually recursive data types, including GADTs, via its `Data.Comp.Multi` module. Available ex-

amples demonstrate the use of annotated ASTs within this framework (example1, example2, example3 related to issue 35).

D. Stitch

Eisenberg [10] explored an implementation of a simply typed λ-calculus. This work featured a non-typechecked AST employing type-level de Bruijn indices to ensure the construction of only well-scoped terms during parsing, alongside a type-checked AST indexed by type-level contexts and node types. The implementation heavily utilizes advanced Haskell extensions, serving as a practical case study for their application.

E. Free Foil

The Free Foil approach [2] by Kudasov et al., implemented in the [11] package, allows for constructing ASTs where nodes are indexed by a phantom type variable representing the scope. Nodes represent either variables or other language constructs. Constructs can be scoped under a (single) binder that extends the scope associated with the phantom type variable. This structure enables type-safe, capture-avoiding substitution of variables for expressions. Furthermore, it supports the definition of generic recursive AST processing functions applicable to any AST whose node signatures (`sig`) implement required type classes, such as `Bifunctor`.

The example in `Control.Monad.Free.Foil.Example` demonstrates defining an AST for an untyped lambda calculus, using the `LamE` pattern synonym to construct expressions under a binder.

The Free Foil approach presents two potential limitations relative to our technical constraints (Sec. 2.1):

First, the current library version requires mutually recursive types within the AST to be combined into a single sum data type acting as the signature `sig`. The library author notes that supporting truly separate mutually recursive types might be theoretically possible using a different internal representation.

Second, Free Foil might increase AST complexity or size in languages featuring modules or other forms of non-lexical scoping. For instance, in a language with modules, correctly relating variable declarations to their usage sites might necessitate multi-phase analysis (e.g., using scope graphs [12]). If binders are used to track declaration sites, one might need to: (1) parse into an initial AST without resolved binders, (2) perform scope analysis, and (3) construct a second AST with appropriate binders reflecting the resolved scopes. For module imports, this could involve adding nodes within the import's subtree that introduce binders for all imported symbols, while simultaneously needing to retain information about the originating module for each binder.

F. Trees That Grow

GHC utilizes a variant of the Trees That Grow pattern [13] for its internal AST representation (HsSyn, Guidance).

This approach involves parameterizing data types (like AST nodes) with a type variable, often representing a compilation phase or state. Instead of concrete types, constructor fields use open type families applied to this parameter. This allows tailoring the AST structure for different phases by defining distinct instances of these type families. For example, GHC parameterizes its AST by phase (parsed, renamed, typechecked), using type families to enable/disable specific constructors or select appropriate annotation types for fields in each phase.

For extensibility, data types typically include an "extension constructor"

whose payload field is also defined via a type family. This allows "adding" new constructors post-definition by resolving this field's type to a new data type and providing pattern synonyms that wrap the new constructors, making them appear as part of the original type. A similar technique, using extra fields defined by type families within existing constructors, allows "adding" fields to those constructors.

Unlike the current Free Foil implementation, Trees That Grow readily supports mutually recursive data types. Its usage pattern resembles employing standard parameterized algebraic data types, although it necessitates defining a significant number of type family instances for constructor fields.

III Type Inference Algorithms

We reviewed several approaches to type inference, ranging from the established system used in GHC to more recent algorithmic developments.

A. GHC's Type Inference

The GHC type inference engine, comprising thousands of lines of code, has evolved incrementally over many years to support numerous extensions to the Haskell type system. Many of these extensions were accompanied by scientific publications.

One foundational extension is `RankNTypes`, which enables arbitrary-rank predicative polymorphism. Its implementation is based on the work by [14], which describes a bidirectional type inference algorithm for such systems. The accompanying Technical Appendix [15] provides proofs of soundness and completeness for the theoretical system described in the paper, but not for the attached Haskell implementation.

B. Developments Beyond GHC

Since the publication of foundational papers like [14], research has yielded multiple new type inference algorithms distinct from GHC's evolution. Several of these have publicly available Haskell implementations [16], [17], [18]. The following subsections highlight some key developments.

C. Bidirectional Typing

Bidirectional typing systems typically operate in two modes: an inference mode, that determines the type of a program construct and helps reduce the need for explicit type annotations, and a checking mode, which verifies top-down whether a program construct conforms to an expected type [19]. This second mode allows typing constructs (like lambda abstractions without annotated arguments in certain contexts) for which types cannot be uniquely inferred.

[20] presented a relatively simple bidirectional type inference algorithm for systems with higher-rank predicative polymorphism. Their algorithm exhibits properties similar [20, Fig. 15] to those described in [14, Sec. 6], including adherence to the η-law [21, Ch. 4] and soundness and completeness with respect to System F [21, Ch. 8]. The authors argue that their formulation, using ordered contexts, existential variables, and precise scoping rules, offers a better type-theoretic foundation compared to the "bag of constraints," unification variables, and skolemization techniques employed in earlier work like [14].

Building on this, Dunfield and Krishnaswami [22] extended the approach to a significantly richer language featuring existential quantification, sums, products, and pattern matching. Their formal development utilizes a desugared core language [22, Fig. 11] derived from a more user-friendly surface language

[22, Fig. 1].

Furthermore, [19] provide an extensive survey of bidirectional typing techniques and offer practical guidance for designing new, programmer-friendly bidirectional type systems.

D. Modifications and Generalizations of Bidirectional Typing

[23] review the algorithm from [20] (Sec. 2.3) and propose refinements, including adding an application mode alongside inference and checking (Sec. 3). They also present a novel algorithm for kind inference (Sec. 7) and compare their system to GHC's implementation (Sec. 8.6), identifying potential areas for GHC improvement (Appendix Sec. C).

[24] address limitations of traditional bidirectional typing (Sec 2.5) by generalizing it to contextual typing. Instead of propagating only type information, this approach propagates arbitrary contextual information relevant to type checking. It replaces the binary inference/checking modes with counters that track the flow of contextual information. This allows for more fine-grained specification of precisely where programmer annotations are required.

E. Beyond Bidirectional Typing: Impredicativity

Parreaux et al. [25] propose SuperF, a novel, non-bidirectional type inference algorithm designed to support first-class (impredicative) higher-rank polymorphism. Impredicative polymorphism permits the instantiation of type variables with polytypes, whereas predicative polymorphism restricts instantiation to monotypes [14, Sec 3.4]. SuperF infers a type for each subterm and then checks these against user-provided annotations written in System F syntax. The authors argue that the subtype inference employed by SuperF is better suited for implementing

first-class polymorphism than approaches relying on first-order unification. As evidence of its expressive power, the authors demonstrate that SuperF can successfully type a wide variety of terms, often even without explicit type annotations (Sec 5.4, Sec 5.5).

Chapter 3

Design and Implementation

I Architecture

A. Static view

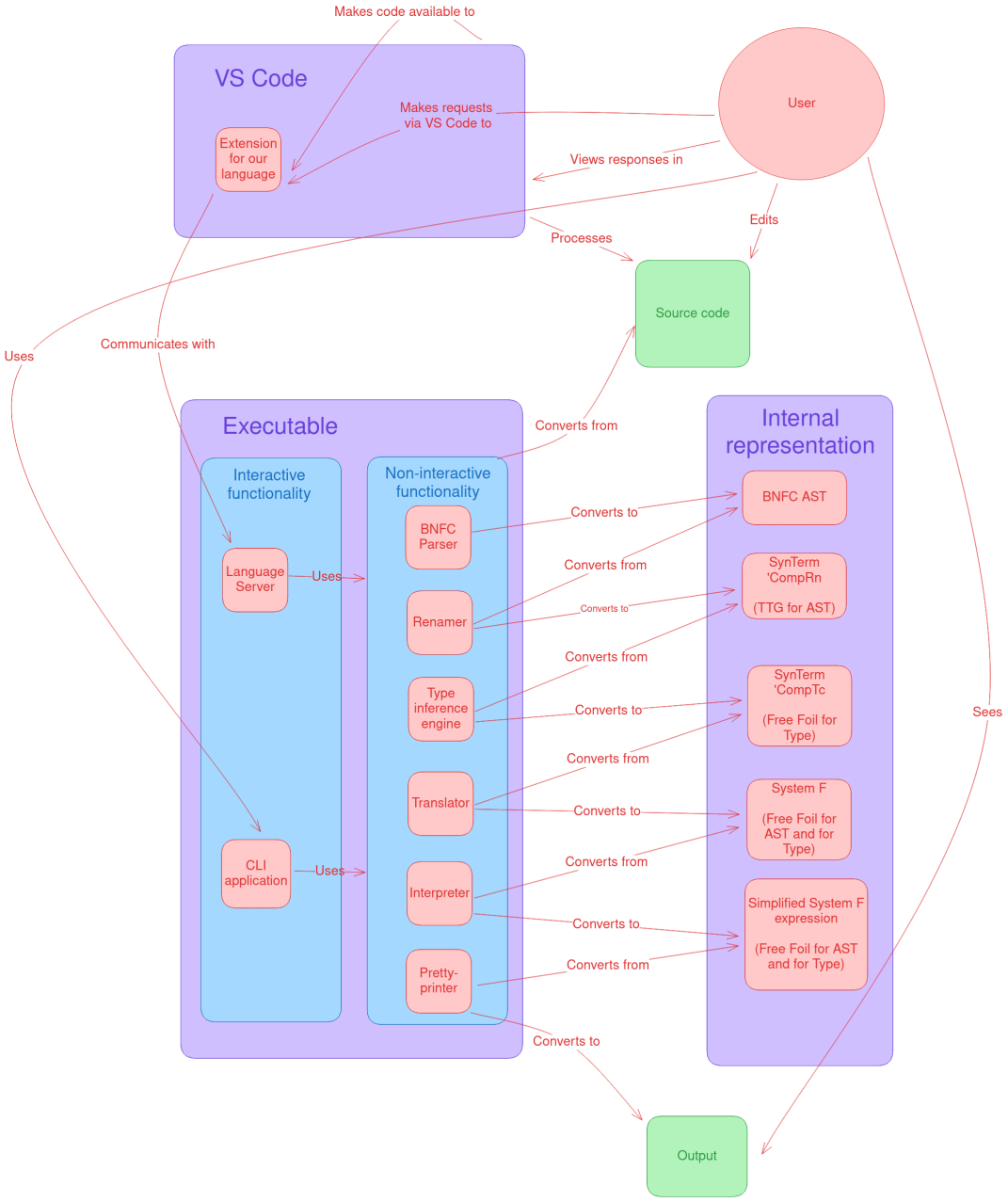

Fig. 1. Static view

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.

D. Process

The language server updates the AST if necessary, performs type checking, finds the node at the given position, and returns its type to the extension.

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.

```haskell
-- 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)
```

```haskell
-- 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.

```haskell
-- 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

...