A higher-order unification implementation for automated theorem proving

by

Christopher Martin Loura

B.S., Kansas State University, 2023

A REPORT

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2025

Approved by:

Major Professor
Dr. Torben Amtoft

Copyright

Abstract

Higher-Order Unification is the process for algorithmically establishing equality between typed lambda expressions. This methodology serves as a foundation for numerous automated theorem provers such as Isabelle, λProlog and Carnap. However, Higher-Order Unification is inherently undecidable, with the most state-of-the-art solution being Gérard Huet's semi-decidable higher-order unification algorithm from 1975.

This report is an implementation of Huet's Algorithm for higher-order unification. The methodology involves parsing lambda expressions and converting them into De Bruijn index notation to eliminate the need for α-conversion. A β-reduction algorithm then reduces the lambda terms to beta-normal form. A matching tree for the two expressions is created to attempt to unify them through pattern matching and simplification. This implementation, written in the Zig programming language, uses its fast performance and memory optimization to create a performant abstract machine and to leverage the capability of Zig to compile into Webassembly.

Table of Contents

List of Figures

Acknowledgments

Firstly, I would like to thank all of my committee members, Dr. Torben Amtoft for advising this project, Dr. Graham Leach-Krouse for presenting the concepts of this report and for being an inspiring professor during his tenure at Kansas State University, and Dr. John Hatcliff for providing really helpful feedback in the places where this project was lacking.

Also, a big thank you to Jacob Legg for his helpful insights and with wrangling Zig, and to all my friends of the 1118 lounge. You guys are the best and most supportive group of people I've had the pleasure of knowing.

And finally, the biggest thank you to my father, Paulo.

Chapter 1

Introduction

Unification is the process of finding a suitable substitution that makes both sides of an equality equal. It is similar in nature to middle school algebra and solving equations for a value of an unknown variable. For example, take the unification problem $g(x, 6) = g(20, y)$. Trivially, applying the substitutions

$$x \mapsto 20 \qquad y \mapsto 6$$

unifies the expression since after applying these substitutions, we get:

$$g(20, 6) = g(20, 6)$$

which correctly unifies. Mathematically, a unification problem is a finite set of equations to solve $E = \{l_1 =? r_1, \ldots, l_n =? r_n\}$ where l_i, r_i are terms or expressions.[1] Unification becomes a more complicated topic when trying to unify higher-order variables, or variables that can be functions.

This problem of Higher-order Unification is what lies at the heart of automated theorem provers. It is concerned with finding substitutions that make typed lambda expressions equal. At its core, higher-order unification operates within the framework of the pure simply typed lambda calculus, a formal system developed by Alonzo Church. Unlike first-order unification,

whose unification algorithm was developed by Robinson[2][3] in 1965 and is both decidable and widely used in logic programming languages like Prolog, higher-order unification is undecidable. Despite its undecidability, higher-order unification is essential for automated theorem proving, type inference, and logical frameworks. Automated Theorem provers such as Isabelle use higher-order unification to perform resolution. Similarly, Carnap, a web-based formal logic framework, depends on higher-order unification by means of Huet's Algorithm.

In Leach-Krouse's paper on Carnap[4], he defines the way that Carnap performs unification. Consider the Carnap rule for existential generalization:

$$\frac{\phi(c)}{\exists x \phi(x)} \text{ (EG)}$$

The inference from $P(f(k)) \wedge Q(f(k))$ to $\exists x(P(x) \wedge Q(x))$ is an instance of existential generalization if the unification problem

$$P(f(k)) \wedge Q(f(k)) = \phi(c)$$
$$\exists x(P(x) \wedge Q(x)) = \exists x.\phi(x)$$

has a solution, when ϕ, c are "metavariables" (called schematic variables in the paper), and $\exists, P, Q, f, k$ are constants. There is a solution:

$$\phi \mapsto \lambda y.(P(y) \wedge Q(y)) \qquad c \mapsto f(k)$$

This can be seen by applying these substitutions to the previously defined unification problem:

$$\phi(c) = (\lambda y.(P(y) \wedge Q(y)))(f(k)) = P(f(k)) \wedge Q(f(k))$$

and

$$\exists x.\phi(x) = \exists x.(\lambda y.(P(y) \wedge Q(y)))(x) = \exists x.(P(x) \wedge Q(x))$$

which successfully unifies the terms. Any rule justification used in Carnap utilizes a form of unification to determine if that was a proper application of the rule. Specifically, Carnap uses Huet's algorithm to check whether certain simple higher-order unification problems have solutions.[4]

Gérard Huet's 1975 semi-decidable algorithm[5] remains one of the most influential approaches to higher-order unification. Due to the semi-decidable nature of the algorithm, however, Huet's method can sometimes not terminate with certain inputs. In other cases with functions containing many variables, the algorithm can be inefficient. As such, potential alternatives to Huet's method become necessary given these limitations.

This report proposes an approach to higher-order unification that, rather than attempting to solve the general undecidable problem, instead focuses on optimizing performance and maintaining clarity and modularity in the implementation. This alternative involves four major steps. Firstly, a lambda expression parser is required to convert a lambda expression into its corresponding abstract syntax tree (AST) and then into the expression's De Bruijn index form. Secondly, an implementation of Krivine's Abstract Machine that will perform β-reduction on the AST until the expression is in weak head normal form. However, since we want full reduction of lambda terms, weak head normal form will not be good enough. As such, for the third step, a β-reduction algorithm that will fully reduce lambda expressions in De Bruijn notation into its beta-normal form. Section 2.1.1 details a more thorough explanation of lambda normal forms. Finally, once the lambda expressions are able to be fully reduced, the final step is to use Huet's higher-order unification algorithm to find a unifier for the lambda expressions.

1.1 Zig

This implementation leverages the Zig programming language, a modern systems programming language designed for its robustness and optimality. The choice of Zig for this implementation is deliberate and offers several advantages. Zig provides low-level control similar to C but with better memory management and debugging due to lack of hidden control flow. Zig allows user control over allocation patterns which are important for the performance of the algorithms. The Zig syntax is also very user-friendly and easy to learn. Finally, since this implementation could be used as a potential baseline for the Carnap theorem prover, a programming language that could compile into Webassembly is necessary to be able to

interface with Carnap's front end.

Chapter 2

Background and Related Work

2.1 The Lambda Calculus

To perform any higher-order unification, we need a language to work in, and that language is the lambda calculus. The Lambda Calculus is a Turing-complete formal system developed by Alonzo Church that utilizes function abstractions and function applications for its computations. We can define the language of the lambda calculus with the following context-free grammar:

$$
\begin{aligned}
\langle E \rangle ::= \ & \lambda x.\ \langle E \rangle \\
| \ & x \\
| \ & \langle E \rangle\ \langle E \rangle
\end{aligned}
$$

Where x is any user-defined lambda variable.

Lambda expressions can be reduced by a process known as β-reduction by which a lambda expression appears on the left side of a function application. This is called a β-redex, and when encountered, the variable bound by the lambda abstraction is substituted with the argument expression. Formally, the reduction $(\lambda x.M)N \rightarrow M[x := N]$ substitutes all occurrences of x in M with N.

A variable, x, is considered free in any lambda expression E if and only if at least one of the following is true:

1. E is a variable and E is identical to x
2. E is of the form $(E_1\ E_2)$ and x occurs free in either E_1 or E_2 or both
3. E is of the form $\lambda y.E'$ where $y \neq x$ and x is free in E'[6]

A variable x is a bound variable in a lambda expression E if and only if at least one of the following is true:

1. E is of the form $\lambda x.E'$ and any occurrence of x in E' is bound by this abstraction.
2. E is of the form $(E_1 E_2)$ and x is bound in either E_1 or E_2 or both
3. E is of the form $\lambda y.E'$ where $y \neq x$ but x is bound in E'

α-conversion allows for the names of bound variables to be changed. For instance, the lambda expression $\lambda x.x$ is equivalent to the expression $\lambda y.y$ via α-conversion.

The language used for my implementation of the lambda calculus is deliberate and is as follows:

1. Unbound variables are denoted by lowercase letters. (i.e. $\lambda x.y$)
2. Metavariables, or flexible, arbitrary expressions, are denoted by capital letters. (i.e. $\lambda.M$)

2.1.1 Normal Forms

Lambda expressions can be reduced to different normal forms. For the purposes of this paper, two main normal forms are important: β-normal form and Weak Head Normal Form.

A lambda expression is in β-normal form if there are no β-redexes located anywhere within the expression. An expression like $\lambda x.(\lambda y.y)(m)$ is **not** in β-normal form because

it contains a β-redex: $(\lambda y.y)(m)$. An expression in β-normal form is fully reduced, as no further β-reduction is possible.

A lambda expression is in Weak Head Normal Form (WHNF) if the outermost structure, or "head," of the expression is not a β-redex. The head of an expression is either a variable, a lambda abstraction, or the leftmost operator in an application chain. For example, $\lambda x.(\lambda y.y)(m)$ **is** in Weak Head Normal Form since its head is a lambda abstraction ($\lambda x.$), even though the body contains a β-redex. Similarly, a variable applied to any number of arguments (possibly containing redexes) is also in WHNF. However, an expression of the form $(\lambda x.M)(N)$ is not in WHNF because its outermost structure is a redex.

2.1.2 De Bruijn Indices

As mentioned previously, the creation of a lambda expression often involves the explicit definition of a bound variable within the instantiation. As an example, take the lambda identity function: $\lambda x.x$. This function takes in some input and returns it. In this case, the variable x is the bound variable. According to α-conversion, we can replace the bound variable x for whatever user-defined variable is desired, therefore as above, we can say that

$$\lambda x.x \equiv \lambda y.y$$

under α-conversion.

However, keeping user-defined variables inside of these lambda expressions makes it quite difficult for a computer to discern the equality between lambda expressions. How can we tell a computer that $x \equiv y$ in this case? Is there a way that we can standardize the structure of lambda expressions to make it easier to discern equality that does not involve using α-conversion repeatedly until all variables are standardized?

The best way to normalize the lambda expressions would be to use De Bruijn indices. Crégut[7] defines the idea in the following manner:

- There is a unique path from the root of a term to a given variable.

- The declaration is located somewhere along the path and can be identified by the number of declarations between it and the occurrence. That number is then called the De Bruijn index of the occurrence.

As a practical example, consider the following lambda expression:

$$(\lambda y.y(\lambda z.zy))(\lambda x.x)$$

The result of applying 1-based De Bruijn indexing to the expression yields:

$$(\lambda.1(\lambda.1\ 2))(\lambda.1)$$

By using De Bruijn indexing, we remove the need to use α-conversion and greatly avoid the possibility of variable capture. Using the previous example, both $\lambda x.x$ and $\lambda y.y$ have the same form of $\lambda.1$. This thus makes it much easier for a computer to determine lambda equivalence.

2.2 A Description of Huet's Algorithm

Overall, Huet's Algorithm involves the decomposition of a given pair of expressions to be unified into simpler pairs and finding substitutions for a certain type of those pairs. This decomposition occurs in the **Simplify** method, and the substitution finding occurs in the **Match** method. Simplify concerns itself with taking the pair to be unified, and reducing the pair to quivalent unification problems. If any pairs are rigid/rigid (or the head of both expressions are not a metavariable), it will try to turn the pair into one of two forms: flex/rigid (where flex is a word describing an expression with a metavariable as its head), or flex/flex. The match function will then take any flex/rigid pairs and attempt to find possible substitutions that would make both terms equal. The program will then stop when one is found.

2.2.1 Other Implementations of Huet's Algorithm

As Huet's algorithm is the most well-known algorithm for performing higher-order unification, there are a number of preexisting implementations that can be found. One of the most popular results is Daniel Gratzer's Haskell implementation, which is also the basis for my Zig implementation[8]. Gratzer's unification method uses two main functions, **simplify** and **tryFlexRigid**. Simplify continuously reduces the given lambda terms of a context until there are no more rigid/rigid simplifications to be done. tryFlexRigid takes a rigid term and a flexible term and generates an infinite list of substitutions that potentially solve the unification problem. Finally, as Gratzer himself describes, his algorithm:

1. Applies the given substitution to all our constraints.
2. Simplifies the set of constraints to remove any obvious ones.
3. Separates flex-flex equations from flex-rigid ones.
4. Picks a flex-rigid equation at random, if there are none, it terminates.
5. Uses tryFlexRigid to get a list of possible solutions.
6. Tries each solution and attempts to unify the remaining constraints, backtracking if it gets stuck.[8]

Another Haskell implementation of Huet's Algorithm comes from Daniel Louwrink. Louwrink's solution claims to be more faithful to Huet's original concept due to the use of "type information to guide the search."[9] His implementation decomposes rigid-rigid pairs into smaller subproblems, like in the simplify algorithm mentioned above. For the pairs that are considered flex-rigid, the algorithm attempts to generate a substitution using the **match** algorithm. The substitution is then applied to all of the remaining equation pairs, and continues to do this until no flex-rigid pairs remain.

Chapter 3

Implementation

3.1 Language and Parsing

For the first step of the Zig implementation, a grammar for the language needed to be defined. While the grammar for the general pure lambda calculus was previously defined in Chapter 2.1, it has the glaring issue of being left-recursive. Left recursion in a grammar can sometimes lead to non-termination in the form of infinite recursion. With certain types of parsers, like LL(1) parsers, left-recursive grammars cannot be parsed at all.

To address this issue, the grammar was modified to eliminate left recursion and adapted to use symbols that can be typed on a standard American keyboard. The resulting grammar is::

$\langle E \rangle$ ::= lam x. $\langle E \rangle$
| x
| ($\langle E \rangle$)
| lam x. $\langle E \rangle$ $\langle E \rangle$
| x $\langle E \rangle$
| ($\langle E \rangle$) $\langle E \rangle$

Where x is any user-defined lambda variable.

While the modified grammar successfully addresses left recursion, it introduces a new challenge: ambiguity. This ambiguity manifests when expressions could be derived in multiple ways using different production rules. Consider the expression $\lambda x.x\ y$, which can be derived through two distinct paths:

$$E \to \lambda x.E \to \lambda x.x\ E \to \lambda x.x\ y$$

and

$$E \to \lambda x.E\ E \to \lambda x.x\ E \to \lambda x.x\ y$$

3.1.1 Handling Grammar Ambiguity

To resolve this ambiguity, our recursive-descent parser implements a deterministic strategy with the following key principles:

1. **Prioritized Pattern Recognition:** The parser first identifies the primary construct (lambda abstraction, variable, or parenthesized expression) and then looks ahead to determine if it should be treated as part of an application.

2. **Left Associativity for Applications:** When encountering sequential terms that could form applications, the parser constructs application nodes in a left-associative manner, following the standard lambda calculus convention.

3. **Greedy Consumption:** After parsing a complete term, the parser greedily consumes any subsequent terms and combines them into application expressions until reaching a syntactic boundary (end of input or a right parenthesis).

The parsing algorithm therefore follows the following steps:

1. When a lambda token is encountered, the parser:

    - Consumes the lambda token and the binding variable
    - Recursively parses the body expression after the period

- Creates a Lambda abstraction node

2. When an identifier token is encountered, the parser:
   - Creates a Variable node
   - Checks for additional tokens
   - If another term follows, it builds an Application node with the Variable as the function and the following term as the argument
3. For application expressions, the parser:
   - Processes the left-hand term first
   - Continues consuming right-hand terms, building a chain of Application nodes
   - Ensures left associativity by making each new application the function of the next application

For the expression $\lambda x.\ x\ y$:

1. The parser recognizes $\lambda x.$ and prepares to parse a lambda abstraction.
2. It recursively parses the body, starting with x.
3. After creating a Variable node for x, it detects that y follows.
4. Rather than ending the lambda body at x, it continues parsing and recognizes that $x\ y$ forms an application.
5. It constructs an Application node with x as the function and y as the argument.
6. This entire Application node becomes the body of the lambda abstraction.
7. The final AST represents $\lambda x.(x\ y)$, enforcing the expected precedence and associativity.

By applying these rules consistently, the parser produces a unique abstract syntax tree for each input, despite the grammar's ambiguity. This approach effectively encodes the

established conventions of lambda calculus directly into the parsing algorithm rather than relying solely on the grammar specification.

The implementation in Zig leverages the language's strong type system to create a clean parser design. An enumeration type defines tokens for lambdas (represented as "lam"), periods, parentheses, and identifiers. A struct type associates each token with its corresponding string representation. This strong typing helps catch potential parsing errors at compile time. The recursive-descent parser implementation takes advantage of Zig's control flow and error handling abilities to manage the potentially complex parsing logic required for lambda expressions. By enforcing left associativity and proper binding of variables, the parser ensures that expressions are interpreted according to standard lambda calculus conventions, despite the ambiguities present in the grammar itself.

3.2 De Bruijn Indexing Algorithm

Once the parsing is complete, we now have a reference to the AST of the lambda expression. To be able to β-reduce, the expression needs to be turned into its corresponding De Bruijn form. The algorithm will perform a preorder traversal of the AST and will take in a dictionary with a key of the lambda argument and a value of 1+ the number of subsequent lambda occurrences within scope, to keep track of any lambda functions we come across. However, if a lambda function is defined on the left side of an application expression, the right side of the application must be outside the scope of that lambda expression. If a new lambda expression is seen, we increment all dictionary values by 1, remove the lambda's argument value, and add the new lambda variable with a value of 1 to the dictionary. If we come across a bound variable expression, the De Bruijn index of that variable is set to the value in the dictionary.

If an unbound variable or metavariable is seen, then the algorithm simply returns the original expression. Once the entire tree has been traversed, the rewritten AST is returned. For the previous example, running the De Bruijn conversion algorithm on the previous AST results in the AST depicted in Figure 3.2.

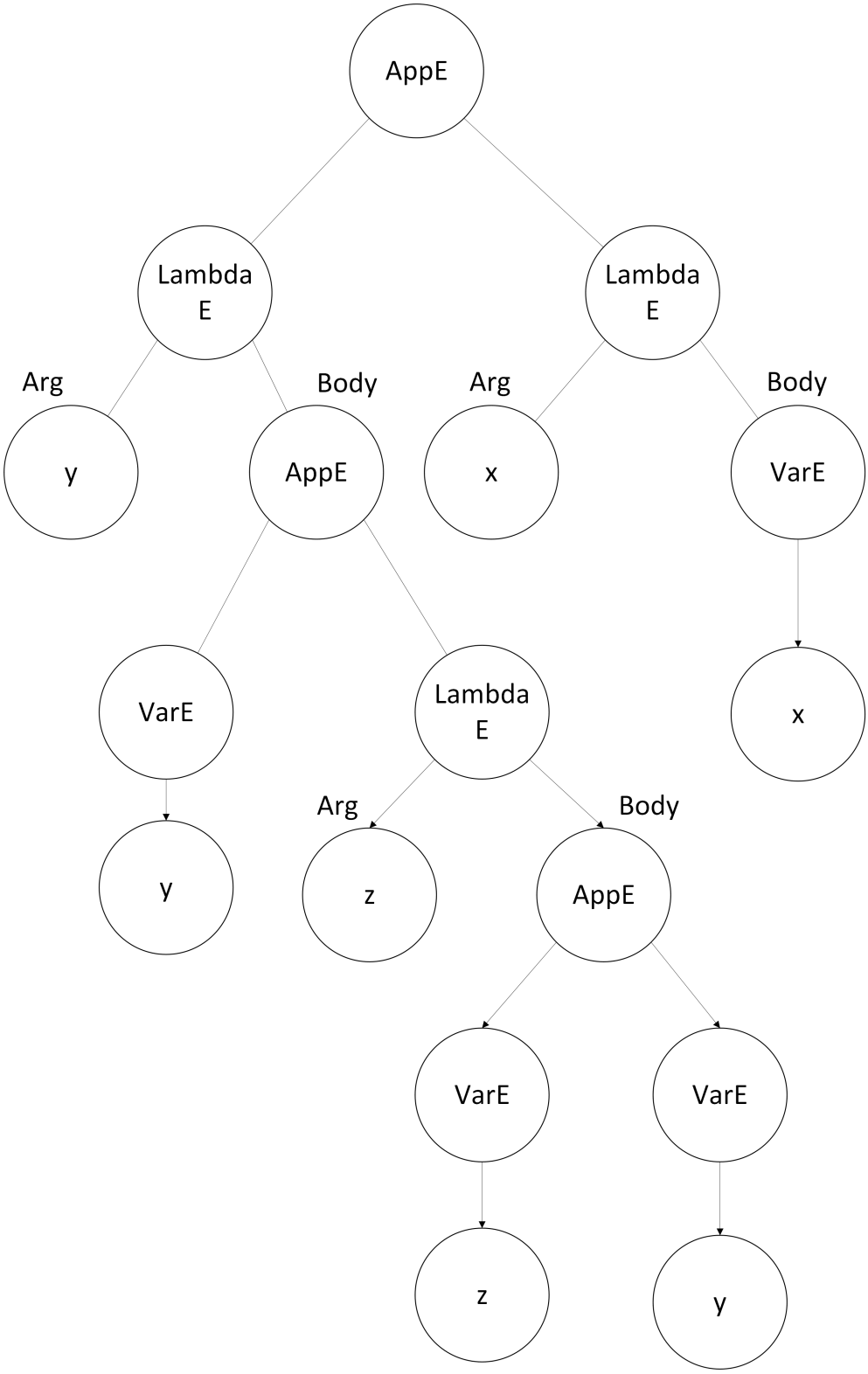

Figure 3.1: *The AST for* $(\lambda y.y(\lambda z.zy))(\lambda x.x)$

3.3 β-Reduction Methods

3.3.1 β-Reduction with Shifting

Once lambda terms are expressed in De Bruijn notation, they can be properly β-reduced. The goal of the algorithm is to identify β-redexes and substitute the argument for the bound variable in the lambda expression. The algorithm performs a preorder traversal of the term to locate β-redexes.

When a β-redex is found (an application of a lambda abstraction to an argument), the algorithm first fully reduces the argument. Then, it performs substitution by searching for all instances of the variable bound by the lambda abstraction, which is represented by the De Bruijn index 1 (though some implementations may use 0 as the starting index).

The substitution process handles three cases for bound variables:

1. If a bound variable's De Bruijn index equals the index being searched for, it is replaced with a copy of the argument expression.

2. If the index is less than the one being searched for, the variable remains unchanged as it refers to a different lambda abstraction.

3. If the index is greater than the one being searched for, its value must be decremented by 1 because the substitution removes one lambda abstraction from the context, thus reducing the "distance" to the binding lambda.

When encountering a lambda expression during substitution, special care must be taken because De Bruijn indices are relative to the nesting depth of lambda abstractions. When substituting a term under additional lambda abstractions, we need to adjust the De Bruijn indices in the substituted term to maintain their correct references. Bound variables, therefore, need to be shifted when passing through lambda abstractions. This shifting ensures that bound variables continue to reference their proper binders after substitution. This method avoids potential variable capture problems that arise in traditional lambda calculus implementations.

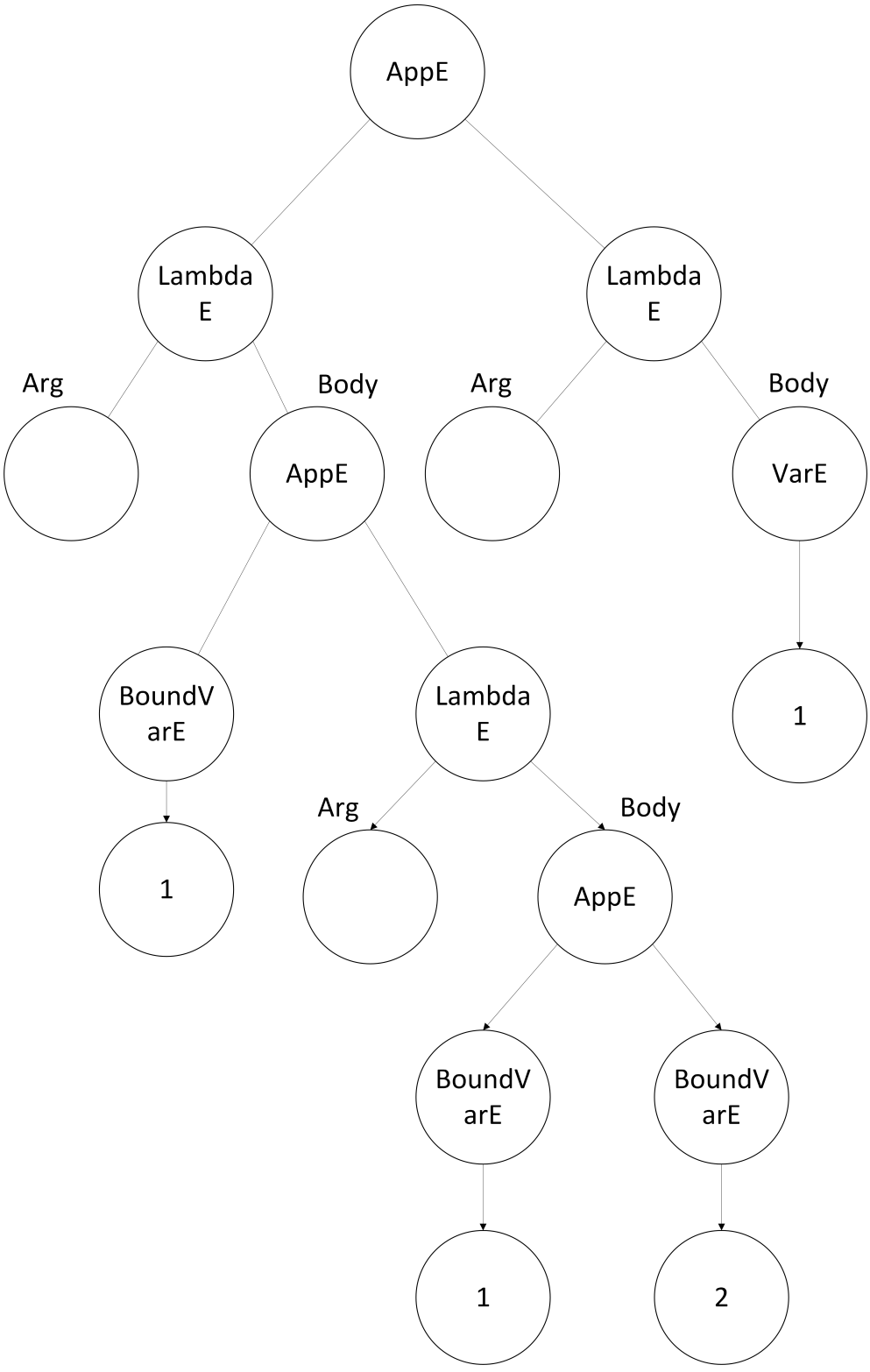

Figure 3.2: *The De Bruijn Converted AST for* $(\lambda y.y(\lambda z.zy))(\lambda x.x)$

Let us examine several examples to illustrate these cases:

$(\lambda.\lambda.\lambda.3\ 2\ 1)((\lambda.1)(\lambda.1))$

First, we evaluate the argument $(\lambda.1)(\lambda.1) = \lambda.1$

We need to substitute $\lambda.1$ into $(\lambda.\lambda.\lambda.3\ 2\ 1)$. Since there are two nested lambdas within the body's expression, the number the algorithm looks for is 3 to substitute into. When the 3 is encountered, we substitute the $\lambda.1$ in for the 3. However, since this insertion is on the left side of a function application, the 2 and 1 are not affected. This then reduces to: $\lambda.\lambda.(\lambda.1)\ 2\ 1$

Finally, we can substitute 2 into $\lambda.1$. This does not change anything since the outer binding lambdas are not being affected and 2 is being substituted into only the identity function:

$\lambda.\lambda.\ 2\ 1$

$\lambda.(\lambda.1(\lambda.2))((\lambda.1)1)$

Step 1: Identify the beta redex at the top level, which is

$(\lambda.1)1$.

Step 2: Evaluate the argument $(\lambda.1)1$

$(\lambda.1)1 = 1$

Step 3: Apply the substitution to the outer beta redex

We need to substitute the result 1 for the bound variable in $\lambda.1(\lambda.2)$

Note that the body of the abstraction $\lambda.1(\lambda.2)$ does not contain any occurrences of the variable bound by the outermost lambda. The indices 1 and 2 are the same variable bound by the first lambda in $\lambda.1(\lambda.2)$. The argument that we are substituting, indicated by a 1, is a variable bound by the outermost lambda. in $\lambda.(\lambda.1(\lambda.2))(1)$

Step 4: Apply the substitution with index adjustment

When we substitute the value 1 into the body, we need to adjust its De Bruijn index as it moves under a new lambda binder in $\lambda.2$. As such, the index must be incremented.

This results in the final expression: $\lambda.1\ (\lambda.2)$

3.3.2 Krivine's Abstract Machine

A Krivine Machine is an abstract machine that uses call-by-name reduction on pure lambda terms to reduce them into head normal form, or in other words, the head of the machine evaluates the state that the lambda term is in and uses both an environment and a stack to continuously push and pop closures, a structure similar to a linked list that references previous environments and stores lambda expressions for potential further evaluation.

The operational semantics of the machine are best defined by Douence and Crégut[10][7]. The state of the machine is a triple: (C, E, S) where C is the code to be evaluated, E is the environment in which the code is being evaluated and S is the stack.

$$1).\ (C_1\ C_2, S, E) \rightarrow (C_1, (C_2, E) :: S, E)$$
$$2).\ (\lambda(C), E, (C_s, E_s) :: S_1) \rightarrow (C, (C_s, E_s) :: E, S_1)$$
$$3).\ (1, (C_e, E_e) :: E_1, S) \rightarrow (C_e, E_e, S)$$
$$4).\ (m+1, (C_e, E_e) :: E_1, S) \rightarrow (m, E_1, S)$$

The starting state of the code is $(C, [], [])$ for some starting input C and the machine halts when there is no rule that can be applied.

This Krivine Machine, also called the Simple Krivine Machine, is a good first step to reducing lambda terms, but by itself does not fully reduce them. Suppose, for instance, that we have a lambda expression in the head position as the starting input. The starting state

would be $(\lambda(C), [], [])$. However, as there is no rule that can be applied to this state since there are no closures on the stack, the machine halts and returns the starting expression. As Krivine states in his paper formulating the machine, "computation stops if there is no redex at the head of the λ-term." [11] As a result of this, we need a way to continue to reduce to ensure that the expression reduces to β-normal form.

For the final implementation, it was decided to not move forward with using a Krivine Machine as the de facto β-reduction method. This is due in part to the finalizing of the shifting β-reduction algorithm, which reduces terms to β-normal form as its final result. In Chapter 5.3, there is mention of future work regarding the implementation of benchmarking tests on the speed of using only the reduction algorithm as opposed to a hybrid reduction algorithm using the simple Krivine Machine and the β-reduction algorithm. Although it did not make it to the final result of the project, its implementation is worth describing.

For the implementation of the Krivine Machine, three basic structures needed to be defined: a stack type, a closure type, and an environment type. The stack type was implemented based on Huy's generic stack implementation in Zig[12]. A closure was defined as a structure that takes in an expression and an environment, and finally an environment type with a head of a closure and a reference to the next environment in the list. The environment also was given the ability to look up closures located within the environment based on some index.

The implementation of the Krivine Machine follows the following procedure:

Switch over the code of the state and perform the following based on the code's expression:

1. If a bound variable is seen and if the De Bruijn index is 1, then reassign the code and environment to the result of popping from the current environment. Otherwise, evaluate the bound variable of the current De Bruijn index - 1 in the environment obtained from the popped closure.

2. If a lambda expression is seen, pop a closure from the stack and create an environment whose head is the popped closure and whose next environment is the current environment. Assign the code to the body of the lambda expression. If there is nothing on

the stack, we are done.

3. If an application expression is seen, push a closure of the argument and the current environment onto the stack. Assign the code to the function.

Let us examine a couple of examples to illustrate the machine:

Suppose we want to reduce the following lambda expression with a simple Krivine Machine:

$$(\lambda.\lambda.\lambda.\ 3\ 2\ 1)((\lambda.1)(\lambda.1))$$

We start by creating a starting state consisting of the expression, an empty environment and an empty stack:

$$((\lambda.\lambda.\lambda.\ 3\ 2\ 1)((\lambda.1)(\lambda.1)), \emptyset, \emptyset)$$

$(\lambda.\lambda.\lambda.\ 3\ 2\ 1, \emptyset, < ((\lambda.1)(\lambda.1), \emptyset) >) \rightarrow$ push arg and environment on stack

$(\lambda.\lambda.\ 3\ 2\ 1, < ((\lambda.1)(\lambda.1), \emptyset) >, \emptyset) \rightarrow$ evaluate lambda body and move from stack to env

Stop processing with weak-head normal form reduction result of $\lambda.\lambda.\lambda.3\ 2\ 1$ in environment $< ((\lambda.1)(\lambda.1), \emptyset) >$

Suppose that we want to reduce the following lambda expression with a simple Krivine Machine:

$$(\lambda.1\ 1)(\lambda.1)$$

We start by creating a starting state consisting of the expression, an empty environment and an empty stack:

$$((\lambda.1\ 1)(\lambda.1), \emptyset, \emptyset)$$

$((\lambda.1\ 1), \emptyset, <(\lambda.1), \emptyset>) \rightarrow$ push arg and environment onto stack

$(1\ 1, <(\lambda.1), \emptyset>, \emptyset) \rightarrow$ evaluate lambda body and move from stack to env

$(1, <(\lambda.1), \emptyset>, <1, <(\lambda.1), \emptyset>>) \rightarrow$ push arg and env onto stack

$(\lambda.1, \emptyset, <1, <(\lambda.1), \emptyset>>) \rightarrow 1$ was seen, update code and env from env

$(1, <1, <(\lambda.1), \emptyset>>, \emptyset) \rightarrow$ evaluate lambda body and move from stack to env

$(1, <(\lambda.1), \emptyset>, \emptyset) \rightarrow$ 1 was seen, update code and env from env

$(\lambda.1, \emptyset, \emptyset) \rightarrow$ 1 was seen, update code and env from env

Stop processing with a correct reduction result of $\lambda.1$

Chapter 4

Higher-Order Unification with Huet's Algorithm

With all of the preliminary information out of the way, we can begin explaining the necessary components to begin Huet's Algorithm to perform higher-order unification.

At the most basic level, Huet's Algorithm searches for metavariables and generates potential solutions to the unification problem by substituting the solution in for the metavariable. If that solution makes the two terms equal, we can stop. Otherwise, another solution is generated and the process continues. Due to the undecidability of the algorithm, however, it may or may not terminate based on the input.

Formally, substitutions are best defined by Fabian Huch[13]:

A substitution, σ, is a mapping from metavariables to terms

$$\sigma = \{M_1 \mapsto t_1, \ldots, M_n \mapsto t_n\} \quad M_i \neq M_j \text{ for } i \neq j$$

A unifier of a unification problem $t =? u$ is a substitution, σ such that for every pair $< t, u >$ of the problem, the terms σt and σu have the same normal form.[14]

The core of Huet's Algorithm occurs in two main functions - **simplify** and **match**.

4.1 Simplify

The simplify procedure takes in a set of two terms, or a constraint, to unify, with the goal of producing more constraints equivalent to the starting one. The function will begin breaking down the constraint into simpler problems to solve, and the constraints will simplify to be flex/rigid or flex/flex. As a motivating example, suppose that the following constraint was passed into the simplify function:

$$< y\ M, y\ N >\ \text{for some unbound variable y}$$

By observation, it would be much easier to solve the flex/flex constraint $< M, N >$. As such, the simplify algorithm performs a sequence of checks on a given constraint and does simplification in the following manner:

1. If two terms are exactly equal and aren't both metavariables, then there is no more simplification to be done. This is a trivial case.

2. Attempt to β-reduce each term. If the term is β-reducible, create a new constraint consisting of the simplified and the other initial term. Return the result of simplifying this new constraint.

3. Check each term's head for rigidity. If they are both unbound variables, check to ensure that their string values are equal. If they are not equal, then there is no valid simplification. Similarly, for each term, take all the terms that are applied after the head and create a list of them. if the lengths of the lists are not equal, there is nothing more to simplify. If they are equal, simplify each of the matching terms. As an example:

   $$\text{Let}\ \ < y\ M(\lambda.N), y\ z\ (\lambda.1) >$$

   Since $y \equiv y$ (they are both unbound variables), create 2 lists: $l1 = [M, \lambda.N]$ and $l2 = [\text{z},\ \lambda.1]$

   $$\text{Because } len(l1) = len(l2), \text{ create new constraints } \ < M, z > \ \text{ and } \ < \lambda.N, \lambda.1 >$$

for potential further simplification.

4. If both terms are lambda expressions, generate a fresh unbound variable and substitute the variable into the body of the lambda expression. Return both bodies as a new constraint. As an example:

$$< \lambda.M, \lambda.1 > \equiv < M, a > \text{ with } a \text{ being a fresh variable}$$

5. If any of the terms still have a metavariable as its head, then return the original constraint for later flex-flex processing.

6. If none of the above are true, then return an empty result.[15][16]

In order to completely simplify a given constraint, we will repeatedly call the simplify function until there are no additional simplification results possible. As a practical example, suppose that we are trying to simplify the following constraint:

$$\textbf{simplify}(< M(\lambda.\lambda.1\ N), z\ (\lambda.\lambda.1(\lambda.h)) >) = [< M(\lambda.\lambda.1\ N), z\ (\lambda.\lambda.1(\lambda.h)) >]$$

This result is due to the head of the left expression being a metavariable and the head of the right expression being an unbound variable. This is as simplified as the function can take this constraint, and will try to find a solution for it in the **match** function.

$$\textbf{simplify}(< y\ M\ (\lambda.N), y\ z\ (\lambda.1) >) = \ [< M, z >, < N, a >]$$

This result is due to the heads of both terms being the same unbound variable. The lengths of applications of each are also equivalent. This means that we can map each element with the same index of each list to each other and simplify the results. $< M, z >$ doesn't reduce further but $< \lambda.N, \lambda.1 >$ does. Since both are lambda abstractions, we can remove the lambda and replace any bound variables with a fresh unbound one. This results in $< N, a >$. Therefore, there are two constraints once the simplify method is complete.

4.2 Match

The goal of the match function is to try to take flex/rigid constraints and generate good potential substitutions for the metavariables and to continue generating valid solutions until one is found. If we are inside the match function, our constraints will be of the form:

$$< M\ t_1\ t_2\ \ldots\ t_n,\ i\ k_1\ k_2\ \ldots\ k_m >$$

where M is a metavariable and i is some rigid term that is likely to be a free variable.[8]

The first thing done by the function is determine which side of the constraint is flexible and which is rigid. The heads of each expression are saved and similar to the simplify method, creates a list of expressions being applied (i.e. if $M\ t_1\ t_2\ \ldots\ t_n$ was the one of the sides of the constraint, we would get a list $[t_1,\ t_2,\ \ldots,\ t_n]$ of applications and a variable assigned to the value of M).

The first case is trivial - if the lengths of both lists are 0, then we create a substitution mapping from the metavariable to the rigid term and return the substitution. If the lengths of the lists are not zero, then more work needs to be done. M must be in one of the following forms:

1. $M = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.\ x_i(M_1\ x_1\ x_2\ \ldots\ x_n)\ldots\ (M_r\ x_1\ x_2\ \ldots\ x_n)$
2. $M = \lambda x_1.\lambda x_2.\ldots.\lambda x_n.\ i(M_1\ x_1\ x_2\ \ldots\ x_n)\ldots\ (M_r\ x_1\ x_2\ \ldots\ x_n)$

Where r is the arity of x_i and M_j is a fresh metavariable for $j \in [r]$. Since these replacements are completely generalized and cover every possible case, then if a constraint is unifiable, then it must be unifiable under one of the substitutions[8 13]. This should generate an infinite number of solutions that the code should lazily evaluate. Programming languages like Haskell with lazy evaluation built into the language are great programming languages to be able to do this, but Zig is somewhat limited. Due to time constraints, I was unable to get the matching algorithm's search space to that infinite level. However, there are lazy evaluation libraries out there that could be integrated into the project at a later date.

The reader may ask what happens when there are flex-flex constraints that need to be simplified? How do we simplify a constraint of the form $< M, N >$? Nicely enough, Huet provides an answer - we do not need to solve flex-flex constraints because they always have solutions. This is called Huet's Lemma[14]. Flex-flex constraints in the implementation are therefore not processed. Some additional work needs to be done with the displaying proper output when trying to solve these equations.

For some motivating examples of the **match** procedure, let us suppose that we wanted to unify the above example of:

$$< y\ M\ (\lambda.N), y\ z\ (\lambda.1) >$$

We know that the results from simplifying result in

$$[< M, z >, < N, a >]$$

As such, there are two flex-rigid constraints that need to be taken care of. The first constraint returns the substitution $?M \mapsto z$ because the length of the arrays after obtaining the heads of each side of the constraint are both 0.

Similarly, the second constraint $< N, a >$ will return the substitution $?N \mapsto z$ for a similar reason.

As a more complicated example, suppose that we wanted to unify:

$$< \lambda.M(j\ (M\ 1)), \lambda.j\ 0 >$$

After simplifying, we have the constraint: $< M(j(M\ a)), j\ a >$ The code would produce the following possible substitutions:

$$M \mapsto \lambda.1,\ M \mapsto \lambda.1(A\ 1),\ M \mapsto \lambda.j(A\ 1),\ M \mapsto \lambda.1(A\ 1)(B\ 1)\ldots$$

Once these substitutions are generated, the code will begin to apply these generated substi-

tutions to the constraint that needs to be solve. As such, for the first substitution $M \mapsto \lambda.1$, we apply the substitution to the constraint, $< (\lambda.1)(j((\lambda.1)a), j\ a >$ and ensure that these are equivalent modulo β-reduction: $< j\ a, j\ a >$. Since both expressions are now equal, we return the substitution $M \mapsto \lambda.1$.

Chapter 5

Testing and Conclusions

5.1 Limitations

5.1.1 Implementation Limitations

There are a few limitations that are worth mentioning. Firstly, this is a simple implementation of Huet's algorithm. With the way that it is set up, it will try to find a unifier for only one metavariable. I was also unable to implement lazy matching tree generation in Zig with the amount of time given for the project (whereas implementations in Haskell can very easily do this). There was also not enough time to thoroughly test the code and develop a trustworthy testing suite.

5.1.2 Testing Limitations

For testing, I wanted to run both Gratzer's and Louwrink's unification implementations to see how it performs against mine. However, due to how deprecated Gratzer and Louwrink's Haskell code is, I was unable to get either of them to work.

The biggest roadblock that was encountered when trying to test the code is the nonexistence of a standard baseline testing framework for higher-order unification. Since making a testing framework myself could lead to reliability issues and developing a set of unifica-

tion tests that covers both breadth and depth of unification problems takes time, there was simply not enough to work with. As such, I chose a handful of test examples that would demonstrate a breadth of unification problems along with how long it took for the program to unify them.

5.2 Testing Results

The tests chosen were deliberate and enough to show at least a level of basic and also deeper workings of the algorithm, while also not being too many examples to clog up an academic paper. In particular, I wanted to exhibit tests that:

1. Demonstrated absolute baseline unification examples
2. Demonstrated that the unification result makes expressions equivalent modulo β-reduction
3. Used the simplify algorithm to break down bodies of lambda abstractions
4. Generated a breadth of matches from the match algorithm.
5. Demonstrated that some unification problems have no solutions.

For testing the speed of the unification process, I used Zig's timer from the standard library. I tested the code on 6 examples:

1. A trivial example: $< M, k >$
2. Another trivial example involving lambda removal: $< \lambda.\lambda.j, \lambda.M >$
3. A more involved example involving β-reduction: $< F, (\lambda.1\ y)(\lambda.1) >$
4. The example from above that involves creating a matching tree and ensures that the expressions are equal modulo β-reduction: $< \lambda.M(j\ (M\ 1)), \lambda.j\ 1 >$
5. An even more complicated example of a deeper matching tree being generated: $< (\lambda.\lambda.s)\ y\ b, (\lambda.\lambda.2\ 1\ b)\ M\ y >$

6. An example of unification failing due to the constraint failing to simplify into a valid flex/rigid problem: $< (\lambda.1)\ y\ M, z\ k >$

The results of the tests are as follows:

- $< M, k > \rightarrow$ 13.931 ms with substitution $?M \mapsto k$
- $< \lambda.\lambda.j, \lambda.M > \rightarrow$ 31.088 ms with substitution $?M \mapsto \lambda.j$
- $< F, (\lambda.1\ y)(\lambda.1) > \rightarrow$ 24.213 ms with substitution $?F \mapsto y$
- $< \lambda.M(j\ (M\ 1)), \lambda.j\ 1 > \rightarrow$ 57.325 ms with substitution $?M \mapsto \lambda.1$
- $< (\lambda.\lambda.s)\ y\ b, (\lambda.\lambda.2\ 1\ b)\ M\ y > \rightarrow$ 219.500ms with substitution $?M \mapsto \lambda.\lambda.s$
- $< (\lambda.1)\ y\ M, z\ k > \rightarrow$ 5.947ms with no unification result possible.

The tests[17] for this code can be run from the terminal with the command **zig test paper_tests.zig**. Make sure that Zig is installed onto your computer and that the terminal is pointing to the directory in which the files are located.

5.3 Future Work

Several enhancements could further develop this masters project. Adding the capability to support unification for more than one metavariable is the number one plan due to how simple it would be to implement. A key area for additional work is to implement lazy evaluation in Zig to construct an infinite matching tree to generate as many possible solutions to the unification problem. I would also be interested in changing β-reduction to use the Krivine Machine implementation and developing the hybrid β-reduction algorithm presented earlier in the paper. Additionally, benchmarking this Zig implementation against established Haskell implementations like Gratzer's or Louwrink's would provide meaningful comparative performance data, and testing both implementations would provide valuable insight into their relative optimizations.

I would also be interested in the development of a reliable testing suite for unification problems. This testing framework would have a significant amount of unification examples and tests for both first-order and higher-order unification implementations. While doing research into unification, a major roadblock was an understanding of what could and couldn't be unified, or what a unification problem practically looked like. This was due to the theoretical nature of this topic, and the examples that did exist were few and far between and were represented in a way that was too mathematical for the common person. As such, I believe that the development of some testing framework that explicitly lays out the results of unifying these terms would be a great tool for researchers and those who want to go into practical applications of unification.

The user interface also presents opportunities for improvement. Enhancing the UI to provide more meaningful feedback during the unification process would significantly improve usability. This could include visualizing the unification steps in a clearer way, displaying intermediate substitutions in a more intuitive format, and implementing more detailed error messages when unification fails. A more robust UI could also offer interactive exploration of the solution space, allowing users to better understand the algorithm's decision-making process.

Finally, since this implementation was created with potentially replacing Carnap's baseline in mind, it would be amazing to see this work integrated into Carnap's backend to perform unification. There are some things that would need to be added to the current implementation. Firstly, the code needs to be extended to be able to handle the typed lambda calculus instead of the pure lambda calculus. It would also need to be extended to be able to generate the infinite matching tree. Finally, since Carnap is also not written in Zig, we would need some code to interface between Carnap's Haskell code and the Zig unification code. Thankfully, there are interfacing techniques for Haskell, such as using a Foreign Function Interfaces (FFI) that can call C functions. Since Zig uses the C's Application Binary Interface (ABI), C can call Zig functions. This makes integrating this Zig implementation into Carnap incredibly feasible, as long as the necessary modifications to the Zig code are made.

Bibliography

[1] Unification (computer science). URL https://en.wikipedia.org/wiki/Unification_(computer_science).

[2] J.A. Robinson. Computational logic: the unification computation. *Machine Intelligence*, 1971.

[3] Conal Elliot. *Extensions and Applications of Higher-order*. PhD thesis, Carnegie Mellon University, 1990.

[4] Graham Leach-Krouse. Carnap: An open framework for formal reasoning in the browser. *EPTCS*, pages 70–88, 2018.

[5] Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, pages 27–57, 1975.

[6] OpenDSA. Programming languages chapter 3 lambda calculus. 2011. URL https://opendsa.cs.vt.edu/ODSA/Books/PL/html/FreeBoundVariables.html#.

[7] Pierre Crégut. An abstract machine for the normalization of -terms. *ACM*, 20, 1990. URL https://dl.acm.org/doi/pdf/10.1145/91556.91681.

[8] Daniel Gratzer. higher-order-unification. Github.com. URL https://github.com/jozefg/higher-order-unification. A small implementation of higher-order unification.

[9] Daniël Louwrink. huet-unify. Github. URL https://github.com/ocecaco/huet-unify/tree/master. Huet's pre-unification algorithm for the simply-typed lambda calculus, implemented in Haskell.

[10] Rémi Douence and Pascal Fradet. The next 700 krivine machines. *Higher Order and Symbolic Logic*, 2007. URL https://www.cs.tufts.edu/~nr/cs257/archive/remi-douence/krivine.pdf.

[11] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher Order Symbolic Computation*, pages 199–207, 2007.

[12] Huy. Zig / case study: Implementing a generic stack. URL https://www.huy.rocks/toylisp/01-08-2022-zig-case-study-implementing-a-generic-stack.

[13] Fabian Huch. Higher order unification. 2020. URL https://www21.in.tum.de/teaching/sar/SS20/5.pdf.

[14] Gilles Dowek. Higher-order unification and matching. *Handbook of Automated Reasoning*, pages 1009–1062, 2001.

[15] Tobias Nipkow. Functional unification of higher-order patterns. *Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, 1993.

[16] Daniel Proksch. Higher-order unification. Technical report, Leopold-Franzens-Universität Innsbruck, 2018.

[17] Christopher Loura. Huets-algorithm-zig. GitHub. URL https://github.com/cmloura/Huets-Algorithm-Zig/tree/main.