Component-Based Synthesis using Combinatory Logic, Intersection Types and Predicates

Dissertation

zur Erlangung des Grades eines

D o k t o r s d e r N a t u r w i s s e n s c h a f t e n

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

CHRISTOPH STAHL

Dortmund

2025

Tag der mündlichen Prüfung: 05. September 2025

Dekan: Prof. Dr. Jens Teubner

Gutachter:

Prof. Dr. Jakob Rehof (Technische Universität Dortmund, Germany)

Prof. Dr. Boris Düdder (University of Copenhagen, Denmark)

Danksagungen

Mein Dank gilt einer Vielzahl an Menschen, ohne die diese Arbeit nicht möglich gewesen wäre. Allen voran meinen Eltern, die mich bereits früh unterstützt haben und mir so die Möglichkeit gegeben haben, meine Interessen zu verfolgen. Mein Dank gilt insbesondere auch meiner Partnerin, die mich immer – insbesondere emotional – unterstützt hat. Mein Dank gilt auch meinem Doktorvater, Prof. Dr. Jakob Rehof, von dem ich sehr viel gelernt habe und der mir die Möglichkeit gegeben hat an spannenden Themen zu arbeiten und mich ermutigt hat nicht aufzugeben. Die Vorlesung *Logische Methoden des Software Engineerings*, die ich bei ihm in meinem Studium gehört habe, hat mein Interesse an Logik vertieft und das Interesse an Typentheorie geweckt und gehört zweifelsohne zu den Veranstaltungen meines Studiums, die mir den meisten Spaß gemacht haben. Ich möchte mich auch bei meinen Kollegen, allen voran Dr. Andrej Dudenhefner und Dr. Jan Bessai bedanken, die mich sehr in meiner Arbeit unterstützt haben. Mein Dank gilt aber auch allen anderen Kollegen in meiner Arbeitsgruppe und an meinem Lehrstuhl, die eine Arbeitsatmosphäre geschaffen haben, in der ich gerne gearbeitet habe. Ein besonderer Dank gilt der Fachschaft Informatik der TU Dortmund, die mich zu Beginn meines Studiums mit offenen Armen empfangen, und Willkommen geheißen hat. Ich möchte mich auch bei allen meinen Freunden bedanken, die mich unterwegs begleitet haben. Ihr brachtet mir regelmäßig Freude und Ablenkung. Ohne euch alle hätte ich es auch nicht bis hierher geschafft, danke.

Abstract

In this thesis, a new iteration of the Combinatory Logic Synthesizer ((CL)S), called Combinatory Logic Synthesizer with Predicates (CLSP) is introduced. This new framework uses inhabitation in the recently developed Finite Combinatory Logic with Predicates (FCLP) as opposed to Finite Combinatory Logic (FCL), that is used in previous iterations. FCLP provides new specification capabilities through the addition of parameterized types, which allow depending on a finite set of literal values and to be constraint by predicates. The focus of CLSP differs from previous iterations of (CL)S in that it built with an emphasis on execution speed and usability, as opposed provable correctness. Both the theoretical foundations of the framework and the implementation in Python are discussed. An evaluation of the framework in the form of performance benchmarks is presented, comparing various modeling techniques that the new framework offers to each other, as well as the synthesis speed of CLSP with previous iterations of (CL)S, showing a speedup of more than 100 to the fastest implementation of (CL)S, in specific benchmarks. Furthermore, a prototypical parallel implementation of the inhabitation algorithm is presented, showing an almost linear speedup in the number of CPU cores used. Finally, two current applications of the framework are discussed, one for finding strategies for a fragment of LTL and one in the context of simulation models in material flow systems, showing how the framework can be used in both more theoretical and practical applications.

Zusammenfassung

In dieser Arbeit wird eine neue Version des Combinatory Logic Synthesizer ((CL)S) vorgestellt, der Combinatory Logic Synthesizer with Predicates (CLSP) genannt wird. Dieses neue Framework verwendet Inhabitation in der kürzlich entwickelten Finite Combinatory Logic with Predicates (FCLP) anstelle der Finite Combinatory Logic (FCL), die in früheren Iterationen verwendet wird. FCLP bietet neue Spezifikationsmöglichkeiten durch die Hinzufügung von parametrisierten Typen, die es erlauben, von einer endlichen Menge von Literalen abzuhängen und durch Prädikate eingeschränkt zu werden. Der Fokus von CLSP unterscheidet sich von früheren Iterationen von (CL)S darin, dass er mit Schwerpunkt auf Ausführungsgeschwindigkeit und Benutzerfreundlichkeit entwickelt wurde, anstatt auf beweisbare Korrektheit. Sowohl die theoretischen Grundlagen des Frameworks als auch die Implementierung in Python werden diskutiert. Eine Evaluation des Frameworks in Form von Benchmarks wird präsentiert, die verschiedene Modellierungstechniken des neuen Frameworks miteinander vergleicht, sowie die Synthesegeschwindigkeit von CLSP mit früheren Iterationen von (CL)S. Letzteres zeigt eine Verbesserung der Laufzeit von bis zu 100-fach, verglichen mit der schnellsten (CL)S Version, in bestimmten Benchmarks. Weiterhin wird eine prototypische parallele Implementierung des Inhabitationsalgorithmus vorgestellt, der eine beinahe lineare Beschleunigung in der Anzahl der verwendeten CPU-Kerne zeigt. Abschließend werden zwei Anwendungen des Frameworks diskutiert, eins, zum Finden von Strategien in einem Fragment von LTL, und eins, im Kontext von Simulationsmodellen in Materialflusssystemen. Dies zeigt, wie das Framework sowohl in theoretischen und praktischen Anwendungen genutzt werden kann.

Contents

1. Introduction **1**
1.1. Disclosure of Contributions . . . 4

2. Combinatory Terms **7**
2.1. Signature and Algebra . . . 8
2.2. Running Example: Maze . . . 14

3. Finite Combinatory Logic **17**
3.1. Intersection Types . . . 17
3.2. Type System . . . 20
3.3. Synthesis in FCL . . . 25
3.3.1. Inhabitation . . . 31
3.3.2. Enumeration . . . 37

4. Finite Combination Logic with Predicates **41**
4.1. Terms and Parameterized Types . . . 42
4.2. Type System . . . 47
4.2.1. Examples . . . 48
4.3. Synthesis in FCLP . . . 52
4.3.1. Inhabitation . . . 56
4.3.2. Enumeration . . . 61
4.3.3. Optimizations . . . 65

5. Implementation **69**
5.1. Architecture . . . 71
5.2. Data Types . . . 72
5.2.1. Combinators . . . 72
5.2.2. Intersection Types . . . 74
5.2.3. Parameterized Terms and Predicates . . . 76
5.2.4. Repositories . . . 81
5.2.5. Terms . . . 87

5.2.6. Parameterized Tree Grammar . . . 88
5.3. Inhabitation . . . 89
5.4. Enumeration . . . 90
5.5. Interpretation . . . 91
5.5.1. Signatures and Interpretations . . . 91

6. Benchmarks **95**
6.1. Comparing FCLP to FCL . . . 96
6.2. Comparing Term and Literal Predicates to Generate-and-Test . . . 98
6.3. Inference of Literal Variables . . . 101
6.4. Source as Parameter . . . 103
6.5. Comparing Python Implementations . . . 104
6.6. Comparing CLSP to Previous (CL)S Implementations . . . 105

7. Parallelized Inhabitation **107**
7.1. Implementation . . . 107
7.2. Evaluation . . . 108
7.3. Thread-based parallelism . . . 110

8. Applications **111**
8.1. Reactive System Synthesis . . . 111
8.2. Synthesis of Simulation Models . . . 114

9. Conclusion **117**

Bibliography **121**

A. Benchmark Results **127**

Lists of Figures, Definitions, and Examples

List of Figures

Combinatory Terms **7**

1. 5×5 Maze With a Solution Marked in Red 16

Finite Combinatory Logic with Predicates **41**

2. 5×5 Maze With Exactly two Loop-Free Solutions 64

Implementation **69**

3. Architecture of the Synthesis Process in CLSP 71
4. A Graph for Example 40 85

Benchmarks **95**

5. Comparison of FCL and FCLP for Different Maze Sizes 96
6. Comparison of FCL and FCLP for Different Maze Sizes (Logarithmic Scale) 97
7. Runtime for Loop-Free Solutions Using Literal, Term and Cached Term Predicates 100
8. Runtime for Loop-Free Solutions Using Literal, Term and Cached Term Predicates (Logarithmic Scale) 100
9. Comparing Different Strategies of Filtering Substitutions 103
10. Source as Parameter and Source in Intersection Type 104
11. Comparison Between PyPy and CPython 105

Parallelized Inhabitation **107**

12. Runtimes for Different Maze Sizes With no, 2, 4, 8 and 16 Workers . . . 109
13. Comparison Between Different Maze Sizes With 1, 2, 4, 8, 16 and 17 Workers 110

14. Repository to Compute Strategies for Reachability 113

List of Definitions

Combinatory Terms **7**

1. Combinatory Terms . . . 8
2. Sorts . . . 9
3. Many Sorted Set . . . 9
4. Function symbol . . . 9
5. Many-Sorted Signature . . . 10
6. Σ-Term . . . 10
7. Σ-Algebra . . . 12
8. Σ-Homomorphism . . . 13
9. Term Algebra . . . 13
10. Term Folding . . . 14
11. Maze . . . 14
12. Path in a Maze . . . 15
13. Solution for a Maze . . . 15

Finite Combinatory Logic **17**

14. Intersection Type . . . 18
15. BCD-Subtyping . . . 19
16. Combinator Repository . . . 20
17. Finite Combinatory Logic . . . 20
18. Inhabitation . . . 26
19. Synthesis . . . 26
20. Path . . . 27
21. Organized Type . . . 27
22. Multi-Arrow . . . 29
23. Multi-Arrow Partition . . . 29
24. Minimal Set Covering . . . 29
25. Minimal Covering . . . 30
26. Regular Tree Grammar . . . 30

Finite Combinatory Logic with Predicates **41**

27. Combinatory Terms With Literals . . . 42
28. Intersection Types With Literals . . . 42
29. Literal Repository . . . 43
30. Parameterized Types . . . 43

31. Convenience Functions for Parameterized Types 44
32. Closed Parameterized Types 45
33. Literal Substitution 45
34. Multi Literal Substitution 45
35. Term Substitution 46
36. Multi Term Substitution 46
37. Parameterized Combinator Repository 46
38. Finite Combination Logic with Predicates (FCLP) 47
39. Inhabitation in FCLP 52
40. Synthesis in FCLP 53
41. Parameterized Tree Grammar 53
42. Strategy interpretation [44, Definition 10] 114

List of Examples

Combinatory Terms **7**

1. Signature for Natural Numbers 10
2. Signature for Lists Over a Set X 10
3. Nat-Terms and List(X)-Terms 11
4. Nat-Algebra for $\mathbb{N}$ 12
5. List(X)-Algebra for Tuples 12
6. List(X)-Algebra for Lengths 13
7. Folding for Nat-Terms 14
8. Solution for a Maze 16

Finite Combinatory Logic **17**

9. Combinator Repository and Typing Judgement 23
10. Combinator Repository for Mazes in FCL 23
11. Synthesis in FCL 26
12. Organized Types 28
13. Multi-Arrows 29
14. Infinitely Many Inhabitants 30
15. Derivation of a Regular Tree Grammar 31
16. Run of Inhabit for FCL 32
17. Regular Tree Grammar for Maze Solving 36

18. Run of Enumerate for FCL 37

Finite Combinatory Logic with Predicates **41**

19. Parameterized Types 44
20. Repository with Literal Quantifiers and Predicates 48
21. Repository With Term Quantifiers and Predicates 49
22. Repository With Literal and Term Quantifiers and Predicates 49
23. Repository With Equality Constraints 50
24. Repository for Maze Solving in FCLP 51
25. Derivation and Parameterized Tree Grammar With Trivial Predicates . . 54
26. Derivation and Parameterized Tree Grammar With Nontrivial Predicates 55
27. Derivation and Parameterized Tree Grammar With Equality Constraints 55
28. Run of Inhabit for FCLP 57
29. Run of Enumerate for FCLP 61
30. Loop-Free Solutions in a Maze 64
31. Comparison of Different Predicates 66

Implementation **69**

32. Intersection Types in CLSP 75
33. Parameterized Types as Param Objects and eDSL 79
34. CLSP Repository With Literal Quantifier and Predicate 82
35. CLSP Repository With Term Quantifier and Predicate 82
36. CLSP Repository With Literal and Term Quantifier and Predicate 82
37. CLSP Repository With Equality Constraints 83
38. CLSP Repository for Maze Solving 83
39. CLSP Repository Using a Virtual Group 84
40. CLSP Repository Modeling a Power Set 85
41. CLSP Terms 87
42. Signatures and Interpretations for Solutions for Mazes in Python 92

1. Introduction

Synthesis is the process of automatically generating an output according to a specification. Often times, this output is a program [30], but it can also be a circuit [36], a mathematical proof [26] or a mechanical design [9]. Component-based synthesis is a technique for automatically generating these outputs from a set of reusable components. This approach stands in contrast to the traditional synthesis approach, where the result is constructed from scratch [10]. The component-based approach, however, aligns itself with how software is developed in practice, where developers often reuse existing libraries and frameworks to build new applications. In fact, the notion of "Don't repeat yourself" (DRY) is a core principle in software development, which encourages developers to reuse existing code instead of duplicating it [31]. The idea of using combinatory logic with intersection types for component-based synthesis is introduced by Rehof in [39] based on prior work in [40] and [18]. Based on this work, a family of synthesis frameworks, called *Combinatory Logic Synthesizer ((CL)S)* [7] was developed. The earliest versions of (CL)S are implemented in Prolog [16], while newer versions are implemented in Scala [3], F# [19] and Python [6].

Generally, these frameworks work by constructing terms by applying the given components to each other, to satisfy a given specification. Each component is equipped with a specification of its own, and together with a set of rules that determine how components can be combined and what specification the combined terms adhere to, these terms can be built.

All versions of (CL)S use intersection types as both the specification language for the components and the terms. They differ however in their implemented type theory, the aforementioned set of rules. Earlier versions of (CL)S use Bounded Combinatory Logic (BCL) [17], a variant of combinatory logic, that restricts substituting type variables to a given bound, later versions use Finite Combinatory Logic (FCL) [40], that disallows type variables all together.

One drawback of FCL, compared to BCL (and the unconstrained combinatory logic), is that it is not possible to model general polymorphism. This drawback is slightly mitigated by the fact, that intersection types allow for an "ad-hoc" form of polymorphism, where a

single term can have multiple types, but it is not possible to abstract over types in general, as can be done in other type theories, like System F [28, 41]. While this is a limitation of FCL, it is also a strength, as it allows for a more efficient synthesis process, since inhabitation – the underlying decision problem for synthesis – in Finite Combinatory Logic with intersection types $FCL(\cap, \leq)$ is ExpTime-complete [40], inhabitation in BCL is $(k+2)$-ExpTime-complete (where k is the bound) [17] and (relativized) inhabitation in combinatory logic is undecidable [15, 40].

This drawback is further mitigated in the current version of (CL)S, by encoding BCL with a bound of 0 in FCL [3], which allows for substitutions by types without arrows, by requiring the user to explicitly specify the available substitutions. While in theory, this allowes to substitute atomic types by intersections, in practice substitutions are only done by atomic types themselves [48] or not at all [37].

Other extensions to FCL and the (CL)S framework include the introduction of a boolean query language to intersection types [23]. This extension allowes using the boolean combinators $\wedge, \vee$ and $\neg$ to be part of the specification for the synthesis result. While this can be seen as the spiritual predecessor to CLSP, as it introduced separating the type language used to specify the synthesis result from the type language used to specify the combinators, it was ultimately abandoned, since its lack of performance was limiting its applicability.

A limiting factor of (CL)S and using inhabitation in FCL in general is the lack of expressiveness in the type language. While it is possible to manually encode semantic information in the types, it is not possible to specify additional constraints on the synthesis process, like the size of the terms, or the number of times a combinator can be used, without significant overhead. A direct consequence of this is that users of the framework need to have a deep understanding of the underlying type theory to use the framework effectively. A second consequence of this is that, since these constraints need to be encoded, the synthesis process is not aware of these encodings, leading to a slow and unoptimized process overall.

In this thesis, we present a new version of (CL)S, that is based on Finite Combinatory Logic with Predicates (FCLP) [24], an extension to FCL, that tries to solve the aforementioned problems. The usage of FCLP allows for constraints to be directly formulated as predicates. Furthermore, we allow for literal variables to be part of the type language, that range over some finite sets. This allows for a limited kind of polymorphism without the necessity of using BCL. This extension is built into the type language creating the notion of *parameterized types*, which is used as the specification language for components. Since this extends the specifications of the components fundamentally, the name of the

framework is changed to *Combinatory Logic Synthesizer with Predicates (CLSP)*, to reflect this change.

The main focus of CLSP and this thesis is to improve the user experience and usability of component-based synthesis, especially compared to earlier versions of (CL)S, that required a fundamental background in type theory. For this, we formulate the following goals:

User-friendliness. The framework should be easy to use, and should be usable with as little knowledge of combinatory logic as possible, but offer advanced techniques for users more familiar.

Execution speed. The framework should be fast in synthesizing terms, especially compared to previous versions of (CL)S.

Portability. The framework should be easy to integrate into other projects, in particular into projects, that make use of previous versions of (CL)S.

Concerning the goal of user-friendliness, we acknowledge the results made in [46] in which an IDE for (CL)S is presented, that helps to understand and debug the synthesis process. In this thesis, we want to take a different approach, by extending the type language itself to allow for a more understandable synthesis process.

While the main focus of this thesis is the practical implementation of CLSP, we also present the theoretical foundation of the framework, but only as far as it is necessary to understand the implementation, a more thorough examination of the theoretical properties of FCLP can be found in [24]. It has been shown, that inhabitation in FCLP is undecidable [24]. However, as we see in this thesis, in most practical applications, the synthesis process terminates and performance exceeds the performance of previous versions of (CL)S. We acknowledge, that there exists a fragment of FCLP, that is decidable [24], that limits predicates to equality and disequalities, but this restriction is deemed to limiting for many practical scenarios and is therefore not used in the implementation of CLSP. We will give ample examples for each introduced concept, to make understanding the backgrounds accessible, especially for readers not deeply involved with type theory. The thesis is structured as follows:

Chapter 2. In this chapter, we present combinatory terms, signatures, and algebras. While signatures and algebras as presented in that chapter can only handle a subset of combinatory terms, this subset is large enough for many real-world applications. Usage of signatures and algebras to define interpretations of combinatory terms follow best practices, as laid out in later chapters. We also introduce the running

example of finding solutions for mazes. While solving mazes using CLSP is not the fastest way of computing solutions, their synthesis shows much of the features, that CLSP offers.

Chapter 3. We present intersection types and Finite Combinatory Logic (FCL). We also give an algorithm for inhabitation in FCL, as well as an enumeration algorithm.

Chapter 4. We introduce Finite Combinatory Logic with Predicates (FCLP) by giving a formal definition of FCLP, and show how it compares to and extends FCL. We introduce the notion of *parameterized tree grammar*, that acts as an intermediate representation of synthesis results and give an algorithm for inhabitation and enumeration in FCLP, as well as an overview of optimizations that are implemented in CLSP. This chapter is the theoretical foundation of CLSP.

Chapter 5. This chapter introduces the implementation of CLSP. We describe the overall architecture and give a short usage guide. We will describe how the theoretical concepts introduced in Chapter 4 and Chapter 3 are implemented in Python, introduce an eDSL, that is used to specify types in CLSP and show best practices for using it by examples of its usage.

Chapter 6. Here, we present benchmarks for CLSP. We compare the performance of synthesis using FCLP to synthesis using FCL. Furthermore, we compare different approaches for modelling repositories on the example of the maze problem, and evaluate the applicability of each approach.

Chapter 7. This chapter introduces parallel synthesis in CLSP. While work on the parallel synthesis is in a prototypical state and currently halted, new development in the standard Python interpreter can revive this work.

Chapter 8. In this chapter, we present two applications, that were done using CLSP. We show how CLSP is used to synthesize reactive systems and simulation models in the context of material flow systems.

Chapter 9. This chapter concludes the thesis, evaluates the goals and gives an outlook on future work.

1.1. Disclosure of Contributions

The work presented in this thesis includes results from the following works, that were co-authored by the author of this thesis:

- Andrej Dudenhefner, Christoph Stahl, Constantin Chaumet, Felix Laarmann, and Jakob Rehof. "Finite Combinatory Logic with Predicates". In: *29th International Conference on Types for Proofs and Programs (TYPES 2023).* Ed. by Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg. Vol. 303. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Aug. 2024, 2:1–2:22. ISBN: 978-3-95977-332-4. DOI: `10.4230/LIPIcs.TYPES.2023.2`[24]

- Andrej Dudenhefner, Felix Laarmann, Jakob Rehof, and Christoph Stahl. "Finite Combinatory Logic extended by a Boolean Query Language for Composition Synthesis". In: *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts.* 2023, p. 105. URL: `https://types2023.webs.upv.es/TYPES2023.pdf#page=111`[23]

- Christoph Stahl, Felix Laarmann, and Andrej Dudenhefner. "Synthesis of Reactive Systems with Reachability and Safety Certificates". In: *Leveraging Applications of Formal Methods, Verification and Validation. 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings.* Vol. 15219. Lecture Notes in Computer Science. (Submitted). Springer, 2025[44]

- Jan Winkels, Jannik Löhn, Felix Özkul, Robin Sutherland, Christoph Stahl, Jakob Rehof, and Sigrid Wenzel. "Synthesizing Structural Variants in Discrete-Event Simulation Models using Finite Combinatory Logic with Predicates". In: *Software Engineering and Formal Methods: 23nd International Conference, SEFM 2025, Toledo, Spain, November 10-14, 2025, Proceedings.* (In preparation). Springer-Verlag, 2025[49]

In [24], the theoretical foundation of FCLP is presented. The author of this thesis contributed to the idea of adding literals and predicates, the type language and the type system, and is the primary author of the implementation detailed in Section 4 of that paper. This includes the development of the presented embedded domain specific language and the changes to the previous CLS-Framework and inhabitation algorithm to support the additions of FCLP. The author also contributed to the writing of the paper, in particular Section 4. Results of this paper are used in Chapters 4 and 5 of this thesis.

In [23], a boolean query language for inhabitation in $FCL(\cap, \leq)$ is presented. While results of this work are not directly used in this thesis, the extension of the Python implementation of (CL)S is used as a basis for the implementation of CLSP. The author of this thesis implemented the extension of the python implementation of (CL)S to support the boolean query language, and contributed to the writing of the work.

In [44], a case study of using CLSP to synthesize reactive systems is presented. The author of this thesis developed and implemented the repositories shown in Section 3 of that paper. The author also contributed to the writing of the paper, in particular Section 3. Results of this paper are used in Section 8.1 of this thesis. This paper is submitted, but not yet accepted for publication.

In [49], a case study of using CLSP to synthesize simulation models for material flow systems is presented. The author of this thesis helped to develop the repositories used in that paper, and formalized the notion of *network terms* and *normal forms*. The author also contributed to the writing of the paper, in particular Section 4.3. At the time of writing, this is still ongoing work in cooperation with the logistics departement of the University of Kassel and prepared to be submitted to *International Conference on Software Engineering and Formal Methods* (SEFM) 2025. Results of this paper, that are relevant to this thesis are presented in Section 8.2.

2. Combinatory Terms

The concept of combinatory logic is introduced by Moses Schönfinkel in 1924 [42] and further developed by Haskell Curry in 1930 [14]. Combinators themselves are symbols, that can be applied to each other to form combinatory terms. Originally, a fixed set of combinators was introduced, each equipped with a computational meaning. The most widely known combinators are the S, K and I combinators with the following meaning:

$$\mathbf{S}\ M\ N\ O = M\,O\,(N\,O)$$
$$\mathbf{K}\ M\ N = M$$
$$\mathbf{I}\ M = M$$

Combinatory logic with a fixed combinatory base of $\{S, K, I\}$ is Turing complete [43], meaning combinatory terms, that exclusively use the combinators S, K and I, can represent every computable function.

The base of $\{S, K, I\}$, however, is not the only possible base for combinatory logic, as other bases have been introduced, that are also Turing complete [1]. Furthermore, a base does not necessary need to lead Turing completeness, to be useful. In this thesis, we consider a more general case, where the set of combinators is not fixed, but arbitrary. Combinators and combinatory terms are the backbone of our approach to component-based synthesis. The approach laid out in this thesis – and is also used in previous versions of the Combinatory Logic Synthesizer (CL)S – is based on the idea of building combinatory terms from a set of combinators, that are then interpreted as domain-specific structures, like executable programs, CAD assemblies or factory plans.

In this chapter, we define combinatory terms and give a notion of signatures and algebras, that allow us to interpret combinatory terms. We also do not consider an inherent computational meaning of combinatory terms, but only their structure. However, we interpret the combinators as term constructors for using them in an algebraic approach. The approach we introduce in this chapter is a simplified version of the algebraic approach taken in [3], that aligns itself well with the implementation we discuss in Chapter 5.

Combinatory terms are built by applying combinators onto each other, and are defined as follows.

Definition 1: Combinatory Terms

Given a finite set of symbols $\mathcal{C}$, a *combinatory term over* $\mathcal{C}$ is defined as follows:

$$T_{\mathcal{C}} \ni M, N ::= C \mid (M\, N)$$

with $C \in \mathcal{C}$.
We call the elements of $\mathcal{C}$ *combinators*.

We consider the application of combinatory terms as left-associative, so that $M\,N\,O$ is equivalent to $(M\,N)\,O$, and omit the outermost parentheses. Furthermore, we use upper case letters as meta variables for combinatory terms. Unless specifically needed, we consider the set $\mathcal{C}$ to be given implicitly.

2.1. Signature and Algebra

Since combinatory terms – as we consider them – only carry a structure, but no computational meaning, we add those through external means. The most straightforward way is to give each combinator a meaning in terms of a function, and then replace each combinator by their respective function and evaluate the resulting function calls. The advantage of separating the structure from the meaning is that we can define multiple interpretations for a given set of combinators.

We utilize an approach based on universal algebra [29], that allows terms to be described by a signature, and then interpreted in an algebra. This approach allows multiple algebras to be associated with a single signature and thereby giving multiple interpretations to a single term. Traditionally, a signature defines the structure of terms in the context of logics and data types, but this approach aligns itself well to terms and interpretation in combinatory synthesis.

We give a notion of S-sorted signatures, that is usually used to model many-sorted-logics [47]. Furthermore, we restrict function symbols to range over product types and add a notion of base sets. These kinds of signatures are compatible with a restriction of combinatory terms, namely that every combinator C has a fixed arity, meaning that it is always applied to the same number of arguments. While it is possible to define a notion of signatures and algebras, that allows combinators to have a variable arity [3], the fixed arity restriction allows for a direct implementation in Python for users and captures many use-cases of synthesis in combinatory logic. For use-cases, that require combinators with variable arity, a non-algebraic approach is still possible.

We consider some fundamentals from the field of universal algebra. The basic Idea of the algebraic approach is to understand each combinator as a function symbol in terms of a signature, and then give an interpretation of each function symbol as a function. For a term M, we replace each combinator by their respective function and evaluate the functions. We can group the interpretations of each combinator together as an algebra, which allows us to create multiple algebras and therefore multiple interpretations for a given set of combinators.

First, we define the notion of sorts and signatures, that describe the structure of an algebra.

Definition 2: Sorts

A *sort* is a symbol, that can be distinguished from other sorts.

Definition 3: Many Sorted Set

Given a set $\mathcal{S}$ of sorts, an $\mathcal{S}$-*sorted set* is a family of sets $A = (A_s)_{s\in\mathcal{S}}$, with associated projections $\pi_s(A) = A_s$ for each $s \in \mathcal{S}$. We call $\mathsf{dom}(A) = \mathcal{S}$ the domain of A.

We usually refer to the sets A_s directly, and omit the projection.

Given a set of sorts $\mathcal{S} = \{s_1, \ldots, s_n\}$, we often give an $\mathcal{S}$-sorted set explicitly as a mapping

$$A = (s_1 : \ldots, \\ \vdots \\ s_n : \ldots)$$

to denote the association of the sets with the sorts.

We also introduce the notion of *base sorts*. Syntactically base sorts are sorts, but semantically, they act have one global interpretation per signature, while we sorts can have different interpretations for different algebras.

Definition 4: Function symbol

Given a set of sorts $\mathcal{S}$ a *function symbol* f over $\mathcal{S}$ is a symbol, that is equipped with a domain $\mathsf{dom}(f) = s_1 \times \ldots \times s_n$ and a range $\mathsf{ran}(f) = t_1 \times \ldots \times t_m$ with each s_i and $t_i \in \mathcal{S}$. We denote the symbol and its types as $f : \mathsf{dom}(f) \to \mathsf{ran}(f)$.

For $n = 0$, we write $f : 1 \to \mathsf{ran}(f)$, for $m = 0$, we write $f : \mathsf{dom}(f) \to 1$, with 1 being the empty product.

Definition 5: Many-Sorted Signature

Given a set of sorts $\mathcal{S}$, a set of base sorts $\mathcal{BS}$, a $\mathcal{BS}$-sorted set $\mathbb{B}$, a set of function symbols $\mathcal{F}$ over $\mathcal{S} \cup \mathcal{BS}$, we call a triple $\Sigma = (\mathcal{S}, \mathbb{B}, \mathcal{F})$ an $\mathcal{S}$-*sorted Signature*,

Since we only use many-sorted signatures, we implicitly mean many-sorted signatures, when we refer to a signature.

Signatures are used to describe data types, where the function symbols either construct or destruct the data types. The sorts of a signature usually depict parts of the data types, that is described by the signature. Base sorts contain values, that are not constructed by the function symbols, but are used as-is each corresponding algebra. We use many-sorted signatures rather than single-sorted signatures, as they allow us to distinguish between different types of data in a single signature, which gives us more flexibility in the interpretation of the terms. Additionally, this aligns well with the implementation of the algebraic approach in Python. In this thesis, we focus on constructive signatures, meaning signatures, where the range of the function symbols is a sort.

Example 1: Signature for Natural Numbers

The following signature describes the natural numbers:

$$Nat = (\{nat\}, \emptyset, \{zero : 1 \to nat, succ : nat \to nat\})$$

Example 2: Signature for Lists Over a Set X

The following signature describes lists over a set X:

$$List(X) = (\{list\}, (content : X), \{nil : 1 \to list, cons : content \times list \to list\})$$

Structures that are built by applying function symbols from a signature Σ are called Σ-Terms.

Definition 6: Σ-Term

Given a signature $\Sigma = (\mathcal{S}, \mathbb{B}, \mathcal{F})$, and set of base sorts $\mathcal{BS} = \mathsf{dom}(\mathbb{B})$. We call a $(\mathcal{S} \cup \mathcal{BS})^*$-sorted set the set of Σ-*term* T_Σ, if each $T_{\Sigma,s}$ with $s \in (\mathcal{S} \cup \mathcal{BS})^*$ is the smallest set such that:

$$() \in T_{\Sigma,1},$$
$$f(t) \in T_{\Sigma,\mathsf{ran}(f)} \text{ for each } f : \mathsf{dom}(f) \to \mathsf{ran}(f) \in \mathcal{F} \text{ and } t \in T_{\Sigma,\mathsf{dom}(f)}$$

Example 3: Nat-Terms and List(X)-Terms

Given the signature *Nat* (Example 1), the following are elements of T_{Nat}

- $zero()$,
- $succ(zero())$,
- $succ(succ(zero()))$

Given the signature $List(\mathbb{N})$ (Example 2), the following are elements of $T_{List(\mathbb{N})}$

- $nil()$,
- $cons(0, nil())$,
- $cons(1, cons(0, nil()))$,

Syntactically, the structure of Σ-terms differs from the structure of combinatory terms, as Σ-terms are built by applying function symbols to terms, while combinatory terms are built by applying combinatory terms to each other. We can, however, define an isomorphism between Σ-terms with empty base sorts and combinatory terms with a fixed arity per combinator as follows:

Lemma 1

Given a signature $\Sigma = (\mathcal{S}, \emptyset, \mathcal{F})$. The set of Σ-terms T_Σ is isomorphic to the set of combinatory terms with fixed arity over $\mathsf{dom}(\mathcal{F})$.

Proof. We consider the following functions $[\![_]\!]_\mathcal{C} : T_\Sigma \to T_\mathcal{C}$ and $[\![_]\!]_\Sigma : T_\mathcal{C} \to T_\Sigma$:

$$[\![f(t_1, \dots, t_n)]\!]_\mathcal{C} = f\ ([\![t_1]\!]_\mathcal{C})\ \dots\ ([\![t_n]\!]_\mathcal{C})$$
$$[\![c\, M_0\ \dots\ M_n]\!]_\Sigma = c([\![M_0]\!]_\Sigma, \dots, [\![M_n]\!]_\Sigma)$$

We can show that $[\![_]\!]_\mathcal{C}$ and $[\![_]\!]_\Sigma$ are inverses of each other by induction over the structure of the terms. □

We do not use the isomorphism explicitly, but consider combinatory terms with fixed arity and Σ-Terms interchangeable.

Definition 7: Σ-Algebra

Given a set of sorts $\mathcal{S} = \{r_1, \ldots, r_n\}$, a set of base sorts $\mathcal{BS} = \{b_1, \ldots, b_m\}$, a $\mathcal{BS}$-sorted set $\mathbb{B}$, a set of function symbols $\mathcal{F}$ and a signature $\Sigma = (\mathcal{S}, \mathbb{B}, \mathcal{F})$, a Σ-*algebra* is a tuple $\mathcal{A} = (A, Op)$, where

- A is an $\mathcal{S} \cup \mathcal{BS}$-sorted set $(A_{r_1}, \ldots A_{r_n}, A_{b_1}, \ldots, A_{b_m})$ with $A_{b_i} = \mathbb{B}_{b_i}$. We call A the *carrier set*,
- Op is a set containing a function $f^{\mathcal{A}}$ for each function symbol $f \in \mathcal{F}$, such that for $\mathsf{dom}(f) = e_1 \times \ldots \times e_i$ and $\mathsf{ran}(f) = e_1 \times \ldots \times e_j$, $\mathsf{dom}(f^{\mathcal{A}}) = A_{e_1} \times \ldots \times A_{e_i}$ and $\mathsf{ran}(f^{\mathcal{A}}) = A_{e_1} \times \ldots \times A_{e_j}$.

Example 4: Nat-Algebra for $\mathbb{N}$

Given the signature *Nat* (Example 1), the following is a *Nat*-algebra $\mathcal{N} = (N, \{zero^{\mathcal{N}}, succ^{\mathcal{N}}\})$, interpreting *Nat*-terms as natural numbers:

$$N_{nat} = \mathbb{N}$$
$$zero^{\mathcal{N}}() = 0$$
$$succ^{\mathcal{N}}(n) = n + 1$$

Example 5: List(X)-Algebra for Tuples

Given the signature $List(X)$ (Example 2), the following is a $List(X)$-algebra $\mathcal{L} = (L, \{nil^{\mathcal{L}}, cons^{\mathcal{L}}\}$ over a set X, interpreting $List(X)$-terms as tuple over X:

$$L_{list} = X^*$$
$$L_{content} = X$$
$$nil^{\mathcal{L}}() = ()$$
$$cons^{\mathcal{L}}(x, (l_0, \ldots, l_n)) = (x, l_0, \ldots, l_n)$$

Example 6: List(X)-Algebra for Lengths

Given the signature *List* (Example 2), the following is a *List*-algebra $|\mathcal{L}| = (|L|, \{nil^{|\mathcal{L}|}, cons^{|\mathcal{L}|})$ over a set X, interpreting *List*-terms as natural numbers, that correspond to the length of the list:

$$\begin{aligned} |L|_{list} &= \mathbb{N} \\ |L|_{content} &= X \\ nil^{|\mathcal{L}|}() &= 0 \\ cons^{|\mathcal{L}|}(x, l) &= l + 1 \end{aligned}$$

Given two algebras of the same signature, we can define functions between them, that preserve the structure defined by the underlying signature. We call such functions homomorphisms. Note that homomorphisms are not necessarily unique, as we can always add additional structure to the carrier set, as long as the structure is preserved by the homomorphism.

Definition 8: Σ-Homomorphism

Given a many-sorted signature $\Sigma = \{\mathcal{S}, \mathbb{B}, \mathcal{F}\}$ and two Σ-algebras $\mathcal{A}$ and $\mathcal{B}$, a Σ-*homomorphism* is a family of functions $h = (h_s)_{s \in (\mathcal{S} \cup \mathsf{dom}(\mathbb{B}))^*}$, such that

- for each $s \in \mathsf{dom}(\mathbb{B})$, h_s is the identity function,
- for each $f : s \to t \in \mathcal{F}$:
$$h_t \circ f^{\mathcal{A}} = f^{\mathcal{B}} \circ h_s$$

We use the notation $h : \mathcal{A} \to \mathcal{B}$ to denote that h is a Σ-homomorphism from $\mathcal{A}$ to $\mathcal{B}$.

The collection of all Σ-terms can be described as an algebra, where the carrier set is the set of Σ-terms, and the interpretations of the function symbols simply build the term. This is a useful view, as it allows for a unique homomorphism from the term algebra to any other algebra, that interprets the same signature.

Definition 9: Term Algebra

Given a many-sorted signature $\Sigma = (\mathcal{S}, \mathbb{B}, \mathcal{F})$, we define the *term algebra* $\mathcal{T}_\Sigma$ as the Σ-algebra, where the carrier set is the set of Σ-Terms T_Σ, and the interpretation of each function symbol constructs the respective term, applying its function symbol to the terms of its arguments.

Definition 10: Term Folding

Given a signature Σ and a Σ-algebra $\mathcal{A}$, we define the *term folding* as the unique Σ-homomorphism $\mathit{fold}^{\mathcal{A}} : \mathcal{T}_{\Sigma} \to \mathcal{A}$. The term folding is defined as follows:

$$\mathit{fold}^{\mathcal{A}}(C) = C^{\mathcal{A}}$$
$$\mathit{fold}^{\mathcal{A}}(f(t_1, \ldots, t_n)) = f^{\mathcal{A}}(\mathit{fold}^{\mathcal{A}}(t_1), \ldots, \mathit{fold}^{\mathcal{A}}(t_n))$$

We omit the subscript of the term folding when applying to a term, since the function from the family is uniquely determined by the range of the outermost function symbol. We also write $[\![M]\!]_{\mathcal{A}}$ to denote the folding of a term M in an algebra $\mathcal{A}$.

Intuitively, term folding is a function, that replaces each function symbol in a term by the respective function in the algebra, and then evaluates the term.

Example 7: Folding for Nat-Terms

Consider the *Nat* terms in Example 3 and the *Nat*-algebra $\mathcal{N}$:

- $[\![zero()]\!]_{\mathcal{N}} = \mathit{fold}^{\mathcal{N}}(zero()) = 0$,
- $[\![succ(zero())]\!]_{\mathcal{N}} = \mathit{fold}^{\mathcal{N}}(succ(zero())) = 1$,
- $[\![succ(succ(zero()))]\!]_{\mathcal{N}} = \mathit{fold}^{\mathcal{N}}(succ(succ(zero()))) = 2$

2.2. Running Example: Maze

Over the course of this thesis, we use the running example of finding solutions for mazes to demonstrate various concepts and implementations. Furthermore, as this is a common benchmark for component-based synthesis, it allows for a comparison of the results of this thesis with other approaches. A maze is a grid of positions, where some positions are blocked, and some are free. A solution for a maze is a path from a start position to an end position, that only traverses free positions. We use the coordinate of $(0, 0)$ as the start position and $(n-1, n-1)$ as the end position of an $n \times n$-maze.

Definition 11: Maze

Let $n \in \mathbb{N}$, an $n \times n$*-maze* is a function $\mathcal{M} : \{0, \ldots, n-1\}^2 \to \mathbb{B}$ indicating whether a position is *free* or *blocked*.

Definition 12: Path in a Maze

Given an $n \times n$-maze $\mathcal{M}$, a *path* in $\mathcal{M}$ is a finite sequence of positions $(x_0, y_0), \ldots, (x_l, y_l)$ such that:

- for each $i \in \{0, \ldots, l\}$ we have $(x_i, y_i) \in \mathsf{dom}(\mathcal{M})$ and $\mathcal{M}(x_i, y_i) = \mathit{true}$,
- for each $i \in \{0, \ldots, l-1\}$ we have $|x_i - x_{i+1}| + |y_i - y_{i+1}| = 1$.

Definition 13: Solution for a Maze

A *solution* for an $n \times n$-maze $\mathcal{M}$ is a path $p_0, \ldots, p_l$ such that $p_0 = (0, 0)$ and $p_l = (n-1, n-1)$.

A combinatory term, that represents a path to a maze, can be considered a sequence of moves, where each move is represented by a combinator. Consider the following combinators Up, Down, Left, Right, that represent the moves in the respective direction, alongside a combinator Start, that represents the start of the path. A path in a maze is then represented as a combinatory term, where each combinator is applied to a subpath starting from Start, and a solution is a path that ends at $(n-1, n-1)$.

Following the algebraic approach, we use the following signature,

$$
\begin{aligned}
\mathit{Maze} = (\{\mathit{path}\}, \emptyset, \{\textsc{Start} &: 1 \to \mathit{path}, \\
\textsc{Up} &: \mathit{path} \to \mathit{path}, \\
\textsc{Down} &: \mathit{path} \to \mathit{path}, \\
\textsc{Left} &: \mathit{path} \to \mathit{path}, \\
\textsc{Right} &: \mathit{path} \to \mathit{path}\})
\end{aligned}
$$

to construct a *Maze*-algebra *Path* as follows:

$$
\begin{aligned}
\mathit{Path} = ((\mathit{path} : (\mathbb{N} \times \mathbb{N})^*), \{&\textsc{Start}_{\mathit{Path}}() = ((0, 0)), \\
\textsc{Up}_{\mathit{Path}}(((x_0, y_0), \ldots (x_l, y_l))) &= ((x_0, y_0), \ldots, (x_l, y_l), (x_l, y_l - 1)), \\
\textsc{Down}_{\mathit{Path}}(((x_0, y_0), \ldots (x_l, y_l))) &= ((x_0, y_0), \ldots, (x_l, y_l), (x_l, y_l + 1)), \\
\textsc{Left}_{\mathit{Path}}(((x_0, y_0), \ldots (x_l, y_l))) &= ((x_0, y_0), \ldots, (x_l, y_l), (x_l - 1, y_l)), \\
\textsc{Right}_{\mathit{Path}}(((x_0, y_0), \ldots (x_l, y_l))) &= ((x_0, y_0), \ldots, (x_l, y_l), (x_l + 1, y_l))\}
\end{aligned}
$$

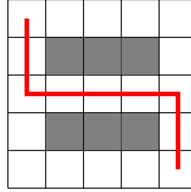

Figure 1.: 5×5 Maze With a Solution Marked in Red

Example 8: Solution for a Maze

Consider the maze and solution in Figure 1. The position $(0, 0)$ is in the top-left corner. White spaces mark a free position and gray spaces mark a blocked position. The given solution can be represented as the following combinatory term:

$$M = \textsc{Down}(\textsc{Down}(\textsc{Right}(\textsc{Right}(\textsc{Right}(\textsc{Right}(\textsc{Down}(\textsc{Down}\,\textsc{Start})))))))$$

Interpreting this term as a path, we get the following sequence of positions:

$$[\![M]\!]_{Path} = ((0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (4, 3), (4, 4))$$

While Example 8 encodes a valid solution for the maze in Figure 1, at no point do we check whether the path is valid, or that the path is a solution. A term such as Up Start would be a valid combinatory term, but neither a valid path nor a valid solution for the maze. In the following chapters, we introduce a type-based approach to combinatory logic, that allows us to encode constraints on the structure of combinatory terms.

3. Finite Combinatory Logic

In this chapter, we present the Finite Combinatory Logic with intersection types and subtyping (FCL($\cap, \leq$)), originally introduced by Rehof and Urzyczyn [40]. For this, we define intersection types and give a subtyping relation and give small examples to illustrate the introduced concepts. Furthermore, we show how the running example of solutions for mazes can be encoded in FCL. Finally, we give an algorithm for component-based synthesis using inhabitation in FCL.

Inhabitation in FCL is a well explored approach for component-based synthesis, that sees practical application in many fields [48, 3, 37, 9, 32, 33]. This chapter does not contain any new results, definitions are based on previously published work [40, 3, 4]. We present the current state of FCL, as it is a foundation for the type theory we introduce in the following chapter. FCL is presented here in a way, that is tailored to the needs of this thesis. For a more detailed introduction to FCL, we refer to the works this chapter is based on.

Combinatory Logic adds a type system to combinatory terms, that allow to specify how combinatory terms can be applied to one another. Finite Combinatory Logic is a restriction of Combinatory Logic, that disallows type variables and substitutions. While this makes the logic less expressive, it also makes inhabitation decidable [40].

In this thesis, when we refer to Finite Combinatory Logic (FCL), we implicitly refer to the logic with intersection types and subtyping, unless otherwise noted.

3.1. Intersection Types

Types in combinatory logic specify how combinatory terms can be applied to one another. We use intersection types with covariant constructors as the type language for combinatory logic. Intersection types are introduced by Coppo and Dezani-Ciancaglini in 1980 [12] and refined in 1983 by Barendregt, Coppo and Dezani-Ciancaglini [2], the extension to covariant constructors is introduced by Bessai, Rehof and Düdder in [4].

Definition 14: Intersection Type

Given an enumerable set of unary constructor symbols $\mathbb{C}$, we define an *intersection type* as follows:

$$\mathbb{T} \ni \sigma, \tau ::= (\sigma \to \tau) \mid (\sigma \cap \tau) \mid c(\tau) \mid \omega$$

with $c \in \mathbb{C}$ and ω as the universal type.

We consider the arrow $\to$ as right-associative, so that $\rho \to \sigma \to \tau$ is equivalent to $\rho \to (\sigma \to \tau)$ and intersection as associative, commutative and idempotent. Furthermore, the arrow to binds stronger than the intersection, so that $\rho \to \sigma \cap \tau$ is equivalent to $(\rho \to \sigma) \cap \tau$. As with combinatory terms, we also omit the outermost parentheses. We use lower case Greek letters to denote types.

We use c instead of $c(\omega)$ if c is a constructor symbol.

Finally, we use $\sigma \times \tau$ as a shorthand for $\pi_1(\sigma) \cap \pi_2(\tau)$, where π_1 and π_2 are constructor symbols.

Given an intersection type in the form of $\sigma_1 \to \ldots \to \sigma_n \to \tau$, we call $\sigma_1, \ldots, \sigma_n$ *argument types* and τ a *target*. Note, that a target for a type is not unique. Consider the type $\sigma_1 \to \sigma_2 \to \tau$. The target of this type can be either τ or $\sigma_2 \to \tau$ with associated argument types σ_1 and σ_2, or just σ_1, respectively.

The intuition behind intersection types is as follows. A term, that is of type $\sigma \cap \tau$, is both of type σ and of type τ. Following this intuition, we allow a term of type $\sigma \cap \tau$ to be used as at any position, were a term of type σ or of type τ is required. For example, if we have a term M of type $\sigma \to \tau$ and a term N of type $\sigma \cap \rho$, the application $M\,N$ should be possible and typed as τ. Additionally, we make use of the ω type. The intuition of ω is, that it corresponds to the empty intersection. Therefore, a term with type $\omega \to \rho$ accepts any term, that can be given a type as its first parameter.

Following this intuition, we use the BCD-subtyping introduced in [2] and extended to use constructor symbols in [4] We include a relation $S \subseteq \mathbb{C} \times \mathbb{C}$ of explicit subtypes of constructor symbols. This can be used to model a taxonomy of constructor symbols, where some constructor symbols are more specific than others. We do not investigate interactions with explicit subtypes in this thesis, but included them for completeness, since they are part of previous iterations of the synthesis framework. More in-depth investigations can be found in [3].

Definition 15: BCD-Subtyping

Given a relation of explicit subtypes $S \subseteq \mathbb{C} \times \mathbb{C}$, we define the BCD-subtyping relation as follows:

$$c(\sigma) \leq_S c'(\sigma) \qquad \text{if } (c, c') \in S \tag{1}$$
$$\sigma \leq_S \omega \tag{2}$$
$$\omega \leq_S \omega \to \omega \tag{3}$$
$$\sigma \cap \tau \leq_S \sigma \tag{4}$$
$$\sigma \cap \tau \leq_S \tau \tag{5}$$
$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq_S \sigma \to (\tau \cap \rho) \tag{6}$$
$$c(\sigma) \cap c(\tau) \leq_S c(\sigma \cap \tau) \tag{7}$$
$$\sigma \leq_S \tau_1 \cap \tau_2 \qquad \text{if } \sigma \leq \tau_1 \text{ and } \sigma \leq \tau_2 \tag{8}$$
$$\sigma' \to \tau' \leq_S \sigma \to \tau \qquad \text{if } \sigma \leq \sigma' \text{ and } \tau' \leq \tau \tag{9}$$
$$c(\sigma) \leq_S c(\sigma') \qquad \text{if } \sigma \leq \sigma' \tag{10}$$

Corollary 1

Let $S, S' \subseteq \mathbb{C} \times \mathbb{C}$ be two relations of explicit subtypes with $S \subseteq S'$, then for all intersection types σ and τ, $\sigma \leq_S \tau \Rightarrow \sigma \leq_{S'} \tau$.

We use $\leq$ as a shorthand for $\leq_\emptyset$.
As per Corollary 1, a subtyping statement, that holds for $\leq$ hold for $\leq_S$ for any S.

Lemma 2

The following subtyping rules are derivable by Definition 15:

1. $c(\sigma \cap \tau) \leq c(\sigma) \cap c(\tau)$
2. $\omega \leq \sigma \to \omega$
3. $\sigma \to (\tau \cap \rho) \leq (\sigma \to \tau) \cap (\sigma \to \rho)$

Proof.

1. $c(\sigma \cap \tau) \leq c(\sigma)$ by eqs. (4) and (10) and $c(\sigma \cap \tau) \leq c(\tau)$ by eqs. (5) and (10), therefore $c(\sigma \cap \tau) \leq c(\sigma) \cap c(\tau)$ by eq. (10).
2. $\omega \to \omega \leq \sigma \to \omega$ by eqs. (2) and (9), therefore $\omega \leq \sigma \to \omega$ by eq. (3).

3. $\sigma \to (\tau \cap \rho) \leq \sigma \to \tau$ by eqs. (5) and (9) and $\sigma \to (\tau \cap \rho) \leq \sigma \to \rho$ by eqs. (4) and (9), therefore $\sigma \to (\tau \cap \rho) \leq (\sigma \to \tau) \cap (\sigma \to \rho)$ by eq. (8).

□

We consider two types τ and σ to be equal ($\tau = \sigma$), iff $\tau \leq \sigma$ and $\sigma \leq \tau$. When considering syntactic equality, we use the notation $\tau \equiv \sigma$.

Corollary 2

The following equalities hold.

1. $c(\sigma \cap \tau) = c(\sigma) \cap c(\tau)$
2. $\omega = \sigma \to \omega$
3. $\sigma \to (\tau \cap \rho) = (\sigma \to \tau) \cap (\sigma \to \rho)$

Definition 16: Combinator Repository

Given a set of combinators $\mathcal{C}$, a *combinator repository* is a function $\Gamma : \mathcal{C} \to \mathbb{T}$.

We frequently give Γ as a right unique relation and use the colon to denote the type of a combinator. In similar fashion, we use $C : \tau \in \Gamma$ to denote, that $\Gamma(C) = \tau$.

Depending on the context, we also call Γ a *type assignment* or a *type environment*. Since we are ultimately concerned with the context of synthesis, we use the term *combinator repository*, as this terminology is used in that context.

3.2. Type System

In combinatory logic, types given to combinators are implicitly polymorphic. Unfortunately, this leads to undecidable type checking and inhabitation [12]. We instead focus on the monomorphic variant of combinatory logic, called *Finite Combinatory Logic* (FCL). While inhabitation in FCL still is EXPTIME-complete [40], in practice this is deemed to be performant enough.

Definition 17: Finite Combinatory Logic

Given a set of combinators equipped with a combinator repository (Γ), we define the *finite combinatory logic* as the following set of typing judgements:

$$\frac{C : \tau \in \Gamma}{\Gamma \vdash C : \tau}\ (\mathit{Var})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\,N) : \tau}\ (\to E)$$

$$\frac{\Gamma \vdash M : \tau \quad \tau \leq \sigma}{\Gamma \vdash M : \sigma}\ (\leq)$$

In the past, the following rules were explicitly included in the type system:

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau}\ (\cap I)$$

$$\frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma}\ (\cap E_l) \quad \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \tau}\ (\cap E_r)$$

However, these rules can be derived from the rules above, and are therefore redundant.

- The $(\cap E)$ rules can be directly derived by applying the $(\leq)$ rule with eq. (4) and eq. (5) respective.
- $(\cap I)$ can be derived by the following Lemma 4.

The following lemma is necessary for the proof of Lemma 4.

Lemma 3

Given a combinator repository Γ, the following statements hold:

1. Given a combinator C, if $\Gamma \vdash C : \tau$, then there exists a ρ with $\rho \leq \tau$ such that $C : \rho \in \Gamma$.
2. Given a combinatory term $M = M_1\,M_2$, if $\Gamma \vdash M : \tau$, then there exists a type ρ such that $\Gamma \vdash M_1 : \rho \to \tau$ and $\Gamma \vdash M_2 : \rho$.
3. Given a combinator C and combinatory terms $M_1, \ldots, M_n$, if $\Gamma \vdash C\,M_1\,\ldots\,M_n : \tau$, then there exists types $\rho, \sigma_1, \ldots, \sigma_n$, such that $\Gamma \vdash M_i : \sigma_i$ for $i \in \{1, \ldots, n\}$ and $C : \rho \in \Gamma$ with $\rho \leq \sigma_1 \to \ldots \to \sigma_n \to \tau$.

Proof.

1. The derivation only consists of $(\leq)$ rules and one (Var) rule. Via the side condition of the $(\leq)$ rule and the transitivity of $\leq$, the statement follows.
2. The derivation consists of some (possibly none) $(\leq)$ rules, followed by a $(\to E)$ rule. Each branch of the $(\to E)$ rule is followed by some (possibly none) $(\leq)$ rules.
3. Induction over n:

Base case $\Gamma \vdash c\, M_1 : \tau$**:** The derivation for $\Gamma \vdash C\, M_1 : \tau$ consists of some (possible none) $(\leq)$ rules, leading to the following $(\to E)$ rule application:

$$\frac{\overline{\Gamma \vdash C : \sigma_1 \to \gamma} \quad \overline{\Gamma \vdash M_1 : \sigma_1}}{\Gamma \vdash C\, M_1 : \gamma}\ (\to E)$$

with the side condition of $\gamma \leq \tau$. By (1) it follows, that $C : \sigma_1 \to \gamma \in \Gamma$ and by eq. (9) we have $\sigma_1 \to \gamma \leq \sigma_1 \to \tau$.

Inductive case $\Gamma \vdash c\, M_1, \ldots M_{n+1}$**:** We have the following induction hypothesis:

IH: If $\Gamma \vdash C\, M_1, \ldots, M_n : \tau$, then there exists types $\rho, \sigma_1, \ldots, \sigma_n$, such that $\Gamma \vdash M_i : \sigma_i$ for $i \in \{1, \ldots, n\}$ and $C : \rho \in \Gamma$ with $\rho \leq \sigma_1 \to \ldots \to \sigma_n \to \tau$.

From (2) we know that τ is in the form $\sigma_{n+1} \to \tau'$ and $\Gamma \vdash M_{n+1} : \sigma_{n+1}$.

□

Lemma 4

Given a combinator repository Γ and a combinatory term M, if $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\Gamma \vdash M : \sigma \cap \tau$.

Proof. By induction on M:

Base case $M = C$**:** For $\Gamma \vdash C : \sigma$ and $\Gamma \vdash C : \tau$, there needs to be ρ with $C : \rho \in \Gamma$ and $\rho \leq \sigma$ and $\rho \leq \tau$. Via eq. (8) we have $\rho \leq \sigma \cap \tau$ and therefore $\Gamma \vdash C : \sigma \cap \tau$.

Inductive case $M = M_1\, M_2$**:** We have the induction hypotheses:

IH1: $\Gamma \vdash M_1 : \sigma_1$ and $\Gamma \vdash M_1 : \tau_1$ implies $\Gamma \vdash M_1 : \sigma_1 \cap \tau_1$.

IH2: $\Gamma \vdash M_2 : \sigma_2$ and $\Gamma \vdash M_2 : \tau_2$ implies $\Gamma \vdash M_2 : \sigma_2 \cap \tau_2$.

Given $\Gamma \vdash M_1\, M_2 : \sigma$, there must exist a ρ_σ with $\Gamma \vdash M_1 : \rho'_\sigma \to \rho_\sigma$ and $\Gamma \vdash M_2 : \rho'_\sigma$ and $\rho_\sigma \leq \sigma$. Given $\Gamma \vdash M_1\, M_2 : \tau$, there must exist a ρ_τ with $\Gamma \vdash M_1 : \rho'_\tau \to \rho_\tau$ and $\Gamma \vdash M_2 : \rho'_\tau$ and $\rho_\tau \leq \tau$.

Since $\Gamma \vdash M_1 : \rho'_\sigma \to \rho_\sigma$ and $\Gamma \vdash M_1 : \rho'_\tau \to \rho_\tau$, we have from **IH1**, that $\Gamma \vdash M_1 : (\rho'_\sigma \to \rho_\sigma) \cap (\rho'_\tau \to \rho_\tau)$. Since $\Gamma \vdash M_2 : \rho'_\sigma$ and $\Gamma \vdash M_2 : \rho'_\tau$, we have from **IH2**, that $\Gamma \vdash M_2 : \rho'_\sigma \cap \rho'_\tau$. Since $(\rho'_\sigma \to \rho_\sigma) \cap (\rho'_\tau \to \rho_\tau) \leq (\rho'_\sigma \cap \rho'_\tau) \to (\rho_\sigma \cap \rho_\tau)$ the following type derivation holds:

$$\cfrac{\cfrac{\cfrac{\text{via } \mathbf{IH1}}{\Gamma \vdash M_1 : (\rho'_\sigma \to \rho_\sigma) \cap (\rho'_\tau \to \rho_\tau)}}{\Gamma \vdash M_1 : (\rho'_\sigma \cap \rho'_\tau) \to (\rho_\sigma \cap \rho_\tau)}\ (\leq) \quad \cfrac{\text{via } \mathbf{IH2}}{\Gamma \vdash M_2 : \rho'_\sigma \cap \rho'_\tau}}{\cfrac{\Gamma \vdash M_1\, M_2 : \rho_\sigma \cap \rho_\tau}{\Gamma \vdash M_1\, M_2 : \sigma \cap \tau}\ (\leq)}\ (\to E)$$

□

Example 9: Combinator Repository and Typing Judgement

Take the combinator A with the following combinator repository:

$$\Gamma = \{\mathrm{A} : (x \to y) \cap x\}$$

We form the following type derivation to show, that the term A A is of type y:

$$\dfrac{\dfrac{\dfrac{\mathrm{A} : (x \to y) \cap x \in \Gamma}{\Gamma \vdash \mathrm{A} : (x \to y) \cap x}\,(\mathit{Var})}{\Gamma \vdash \mathrm{A} : x \to y}\,(\leq) \quad \dfrac{\dfrac{\mathrm{A} : (x \to y) \cap x \in \Gamma}{\Gamma \vdash \mathrm{A} : (x \to y) \cap x}\,(\mathit{Var})}{\Gamma \vdash \mathrm{A} : x}\,(\leq)}{\Gamma \vdash \mathrm{A}\,\mathrm{A} : y}\,(\to E)$$

We revisit the running example of solutions for mazes. By including a family of $\textsc{Free}_{x,y}$ combinators, we encode the layout of the maze, and by requiring a movement combinator to move to a free position, we encode Definition 12.

Example 10: Combinator Repository for Mazes in FCL

Take the combinators $\textsc{Up}, \textsc{Down}, \textsc{Left}, \textsc{Right}, \textsc{Start}$ as well as a family of $\textsc{Free}_{x,y}$ combinators, with the following combinator repository:

$$\begin{aligned}
\Gamma_{\mathcal{M}} = \{ & \\
\textsc{Up} : & \bigcap_{(a,b)\in\{0..5\}^2} \mathit{free}(a \times (b-1)) \to \mathit{path}(a \times b) \to \mathit{path}(a \times (b-1)), \\
\textsc{Down} : & \bigcap_{(a,b)\in\{0..5\}^2} \mathit{free}(a \times (b+1)) \to \mathit{path}(a \times b) \to \mathit{path}(a \times (b+1)), \\
\textsc{Left} : & \bigcap_{(a,b)\in\{0..5\}^2} \mathit{free}((a-1) \times b) \to \mathit{path}(a \times b) \to \mathit{path}((a-1) \times b), \\
\textsc{Right} : & \bigcap_{(a,b)\in\{0..5\}^2} \mathit{free}((a+1) \times b) \to \mathit{path}(a \times b) \to \mathit{path}((a+1) \times b), \\
\textsc{Start} : & \; \mathit{path}(0 \times 0) \\
\} \cup & \\
\{\textsc{Free}_{x,y} : & \; \mathit{free}(x \times y) \mid (x,y) \in \{0 \ldots 5\}^2 \\
& \setminus \{(1,1),(2,1),(3,1),(1,3),(2,3),(3,3)\}\}
\end{aligned}$$

The combinators UP, DOWN, LEFT, RIGHT represent the moves in the respective direction, FREE represents the layout of the maze in Figure 1 and START represents the start of the path.
We give the following typing judgement for a term, that represents a path from position $(0, 0)$ to position $(4, 4)$:

$$\Gamma \vdash \textsc{Down}\,\textsc{Free}_{4,4}\,(\textsc{Down}\,\textsc{Free}_{4,3}\,(\textsc{Right}\,\textsc{Free}_{4,2}\,(\textsc{Right}\,\textsc{Free}_{3,2}\,(\\ \textsc{Right}\,\textsc{Free}_{2,2}\,(\textsc{Right}\,\textsc{Free}_{1,2}\,(\textsc{Down}\,\textsc{Free}_{0,2}\,(\\ \textsc{Down}\,\textsc{Free}_{0,1}\,\textsc{Start}))))))) : \mathit{path}(4 \times 4)$$

Neither the big intersection, nor the addition and subtraction are part of the type language. The big intersection is shorthand for the intersection of all possible positions, and both the addition and subtraction operations need to be unfolded manually on the constructor symbols.

Terms, that can be constructed according to Example 10 encode valid paths and terms, that have the type $\mathit{path}(4 \times 4)$ are valid solutions for the maze in the form of Figure 1. It is easy to see that this can be generalized to any $n \times n$-maze by constructing a combinator repository with the layout encoded as the FREE combinators. Since the FREE combinators are part of the term structure, we need to adapt the *Maze*-signature and algebra of Section 2.2 to include them.

$$\begin{aligned} \mathit{Maze}_{\mathrm{FCL}} = (\{\mathit{path}, \mathit{free}\}, \emptyset, \{\textsc{Start} : 1 \to \mathit{path}, \\ \textsc{Up} : \mathit{free} \times \mathit{path} \to \mathit{path}, \\ \textsc{Down} : \mathit{free} \times \mathit{path} \to \mathit{path}, \\ \textsc{Left} : \mathit{free} \times \mathit{path} \to \mathit{path}, \\ \textsc{Right} : \mathit{free} \times \mathit{path} \to \mathit{path}\} \cup \\ \{\textsc{Free}_{x,y} : \mathit{free} \mid (x, y) \in \{0 \dots 5\}^2 \setminus ((1,1), (2,1), (3,1), (1,3), (2,3), (3,3))\}) \end{aligned}$$

$$
\begin{aligned}
Path_{\mathrm{FCL}} = ((path : (\mathbb{N} \times \mathbb{N})^*, free : \mathbb{N} \times \mathbb{N}), \{ \\
\textsc{Start}_{path}() &= (0, 0) \\
\textsc{Up}_{path}((x, y), (p_0, \dots, p_l)) &= (p_0, \dots, p_l, (x, y)) \\
\textsc{Down}_{path}((x, y), (p_0, \dots, p_l)) &= (p_0, \dots, p_l, (x, y)) \\
\textsc{Left}_{path}((x, y), (p_0, \dots, p_l)) &= (p_0, \dots, p_l, (x, y)) \\
\textsc{Right}_{path}((x, y), (p_0, \dots, p_l)) &= (p_0, \dots, p_l, (x, y))\} \cup \\
\{\textsc{Free}_{x,y} = (x, y) \mid (x, y) \in \{0 \dots 5\}^2 \setminus ((1, 1), (2, 1), (3, 1), (1, 3), (2, 3), (3, 3))\})
\end{aligned}
$$

We see, that both the signature and the combinator repository give the combinators a specification of how they can be applied to each other, but since the types for the function symbols in the signature are more limited, than intersection types, the signature is ony an approximation for legal terms. In the context of this thesis, however, this does not pose a problem, since the synthesis is based on the types. Using the $Maze_{\mathrm{FCL}}$-signature, we can build terms such as $\textsc{UpFree}_{0,0}(\textsc{UpFree}_{0,0}\textsc{Start})$, that do not comply with $\Gamma_{\mathcal{M}}$. On the other hand, we can specify combinator repositories for terms, that cannot be modeled using a signature. The combinator A in Example 9 cannot be given a type as a function symbol, since we cannot model multiple possible arities for a function symbol.

While using the algebraic approach does not encompass all possible repositories, it still captures a wide range of possible repositories. It especially aligns itself well with an implementation in the Python programming language, as we can see in Chapter 5.

3.3. Synthesis in FCL

Type-based synthesis in combinatory logic is the process of finding all combinatory terms, that have a specified type, relative to a set of typed combinators. This is different from approaches like *satisfiability modulo theories* (SMT), where the goal is to find a single solution, that satisfies some specification. We also differ from traditional synthesis, that given a specification constructs a program *ex nihilo*, as we use a set of combinators as building blocks.

Synthesis at its core is an extension of the *inhabitation* problem, that in itself is a decision problem.

Definition 18: Inhabitation

Given a combinator repository Γ and a type τ, the *inhabitation problem* is the decision problem, whether there exists a combinatory term M such that $\Gamma \vdash M : \tau$.

Definition 19: Synthesis

Given a combinator repository Γ and a type τ, the *synthesis problem* is the problem of finding all combinatory terms M such that $\Gamma \vdash M : \tau$.

We refer to the inhabitation problem via the following syntax:

$$\Gamma \vdash ? : \tau$$

In contrast, we refer to the synthesis problem via the following syntax:

$$\{M \mid \Gamma \vdash M : \tau\}$$

Lemma 5

The inhabitation problem for FCL is decidable in ExpTime.

Proof. The proof is given in [40]. □

Intuitively, the synthesis problem $\{M \mid \Gamma \vdash M : \tau\}$ can be solved by following these steps:

1. Find all combinators $C : \rho \in \Gamma$, where ρ has a target of τ.
2. For each type σ, that is an argument of ρ (relative to the target of τ), solve the synthesis problem $\{M \mid \Gamma \vdash M : \sigma\}$.
3. Construct the term by applying the terms to each other according to their types.

Example 11: Synthesis in FCL

Consider the following combinator repository:

$$\Gamma = \{X : ((b \cap d) \to c) \to e, Y : a \to (b \cap d) \to (c \cap e), Z : a \cap b \cap d\}$$

and the following synthesis problem:

$$\{M \mid \Gamma \vdash M : e\}$$

The set of solutions of this problem consist of the following combinatory terms:

$$\{X\,(Y\,Z), Y\,Z\,Z\}$$

We first identify X as possible candidate to produce a term of type e, that requires a term of type $(b \cap d) \to c$ as an input, which leads to the following synthesis problem:

$$\{M \mid \Gamma \vdash M : (b \cap d) \to c\}$$

The only possible combinatory, that can produce $(b \cap d) \to c$ is Y, which requires a term of type a as an input. We therefore need to solve the following synthesis problem:

$$\{M \mid \Gamma \vdash M : a\}$$

The only possible solution is Z. Combining the found terms, we get the term $X\,(Y\,Z)$. Another solution starts from Y and requires two terms of type a and $b \cap d$ as respective inputs, which leads to the following synthesis problems:

$$\{M \mid \Gamma \vdash M : a\}$$

and

$$\{M \mid \Gamma \vdash M : b \cap d\}$$

The combinator Z fulfills both synthesis problems, leading to the term $Y\,Z\,Z$.

Considering Example 11, we see, that the arity of combinators is not fixed. For optimization purposes, we consider types of combinators in the combinator repository, stratified by their arity. Moreover, we "unwrap" the intersections in the targets of a type, to better mark which arguments lead to which target. For this, we present the concepts of *paths*, *organized types* and *multi-arrows*.

Definition 20: Path

A *path* π is an intersection type, that contains no intersection in its target. A path is defined as follows:

$$\mathbb{T}_\pi^\omega \ni \pi ::= \omega \mid c(\pi) \mid \tau \to \pi$$

Definition 21: Organized Type

An *organized type* is a type ρ, that consists of an intersection of paths $\pi_1, \ldots, \pi_n$, $\rho = \pi_1 \cap \ldots \cap \pi_n$. We call the set $\mathbb{P}(\rho) = \{\pi_1, \ldots, \pi_n\}$ the paths of ρ.

Every intersection type can be represented as an organized type using the following function:

$$
\begin{aligned}
organize(\omega) &= \omega \\
organize(c(\omega)) &= c(\omega) \\
organize(c(\rho)) &= c(\pi_1) \cap \ldots \cap c(\pi_n) && \text{with } organzie(\rho) \equiv \pi_1 \cap \ldots \cap \pi_n \\
organize(\sigma \to \omega) &= \omega \\
organize(\sigma \to \rho) &= \sigma \to \pi_1 \cap \ldots \cap \sigma \to \pi_n && \text{with } organize(\rho) \equiv \pi_1 \cap \ldots \cap \pi_n \\
organize(\sigma \cap \tau) &= organize(\sigma) \cap organize(\tau)
\end{aligned}
$$

Lemma 6

Given an intersection type ρ,

- $organize(\rho)$ is an organized type and
- $\rho = organize(\rho)$.

Proof.

- By induction on ρ, each case of *organize* only constructs an organized type.
- By induction on ρ, each case of *organize* is an equality derivable by Definition 15 (See Corollary 2)

□

We use $\mathbb{P}(\rho)$ to denote the paths of the organized type $organized(\rho)$

Example 12: Organized Types

Let a, b, c be constructor symbols, then the following types are organized:

- a
- $(a \to b) \cap (b \to c)$
- $(a \cap b \to b) \cap c \to a$

Multi-arrows are a way to represent the arity of a type, by explicitly separating the arguments from the target type.

Definition 22: Multi-Arrow

A *multi-arrow* is a tuple $m \in \mathbb{T}^* \times \mathbb{T}$. The function $marrows_{\cup}(\pi)$ maps a path π to its set of multi-arrows and is defined as follows:

$$marrows_{\cup}(\sigma_1 \to \ldots \to \sigma_n \to \pi) = \{(\sigma_1, \ldots, \sigma_l), \sigma_{l+1} \to \ldots \to \sigma_n \to \pi \mid l \in \{0, \ldots, n\}\}$$

Since each occurrence of a combinator is used with a specific arity, we partition the set of multi-arrows into sets of multi-arrows of the same arity.

Definition 23: Multi-Arrow Partition

Given an intersection type ρ, the *multi-arrow partition* of ρ is the set of sets of multi-arrows $marrows(\rho)$ obtained by partitioning the set $\{marrows_{\cup}(\pi) \mid \pi \in \mathbb{P}(\rho)\}$ by the size of the first projection.

Example 13: Multi-Arrows

Let a, b, c be constructor symbols, then the following types have the following multi-arrow partitions:

$$\begin{aligned}
marrows(a) &= \{\{((), a)\}\} \\
marrows((a \to b \to c) &= \{\{((), a \to b \to c)\}, \\
&\quad \{((a), b \to c)\}, \{((a, b), c)\}\} \\
marrows((a \cap (b \to c)) \to (a \cap (b \to c)) &= \{\{((), ((a \cap (b \to c)) \to a)), \\
&\qquad ((), ((a \cap (b \to c)) \to b \to c))\}, \\
&\quad \{((a \cap (b \to c)), a), \\
&\qquad ((a \cap (b \to c)), b \to c)\}, \\
&\quad \{((a \cap (b \to c), b), c)\}\}
\end{aligned}$$

The inhabitation procedure does also make use of an algorithm for the *minimal covering* problem, that is a generalization of the *minimal set covering* [34]:

Definition 24: Minimal Set Covering

Given a set S and a set of sets C, the *minimal covering* problem is the problem of finding a minimal set $C' \subseteq C$ such for each $s \in S$, such that $s \in c$ for some $c \in C'$.

Definition 25: Minimal Covering

Given a set S a set C and a relation $\preceq \subseteq S \times C$, the *minimal covering* problem is the problem of finding a minimal set $C' \subseteq C$ such for each $s \in S$, such that $s \preceq c$ for some $c \in C'$.

We do not give an explicit algorithm for the minimal covering problem. A standard algorithm for the minimal set covering problem [35] can be adapted to the minimal covering problem by replacing the containment check with the relation check.

Often times, the set of inhabitants is infinite, as we can see in the following example:

Example 14: Infinitely Many Inhabitants

Consider the following combinator repository:

$$\Gamma = \{X : a \cap (a \to a)\}$$

The set of inhabitants of type a are all terms in the form where X is repeated an arbitrary number of times, therefore the set of inhabitants is infinite.

The (possible infinite) set of inhabitants of a type τ relative to a combinator repository Γ form a regular tree language [40]. Regular tree languages can be represented as a regular tree grammar. A combinatory term $C\, M_1 \ldots M_n$ can be considered as a tree, with C as the root and $M_1 \ldots M_n$ as the children of the root.

Definition 26: Regular Tree Grammar

A *regular tree grammar* is a tuple $G = (N, \Sigma, S, R)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,
- $S \in N$ is the start symbol,
- R is a finite set of rules of the form $A \mapsto f\, B_1 \ldots B_n$, where $A, B_1 \ldots B_n \in N$ and $f \in \Sigma$.

We usually only give the set of rules R for a regular tree grammar, since the other components are usually clear from the context.

We do not give a formal definition of a derivation step, but rather an intuitive definition. A derivation step replaces a non-terminal symbol, occurring as a child in a tree, with the right-hand hand side of a rule, that has the non-terminal symbol as the left-hand hand

side. We say that a word is derivable from a regular tree grammar, if there exists a finite sequence of derivation steps, starting from the start symbol and ending in the word.

Example 15: Derivation of a Regular Tree Grammar

Consider the combinator repository Γ from Example 9 and the type $\tau = y$. The set $\{M \mid \Gamma \vdash M : \tau\}$ is given by the following regular tree grammar:

$$y \mapsto A\, x$$
$$x \mapsto A$$

We derive the term $A\, A$ as follows:

$$y \Rightarrow A\, x \Rightarrow A\, A$$

We see that types used in the type assignment are used as the set of non-terminal symbols, while the combinators are used as terminal symbols.

3.3.1. Inhabitation

An algorithm to solve Definition 18 is given in [40] as an algorithm for an alternating Turing machine. While using a alternating Turing machine is useful for complexity analysis, it is impractical for implementation. We instead present a deterministic algorithm, that also solves Definition 19.

Since we are interested in finding combinators, that produce a certain type (or a supertype thereof), we consider the following relation, that compares the target of a multi-arrow to a type.

$$((\sigma_1, \ldots, \sigma_n), \tau) \leq_S^{tgt} \pi \text{ iff } \tau \leq_S \pi$$

We also consider the following functions, that we do not give an explicit algorithm for:

minimal_covers$(S, C, \preceq)$ A function that computes all minimal covers (Definition 25). We consider multiple minimal covers, since the minimal cover in respect to $\preceq$ is not necessary unique.

maximal_elements$(S, \preceq)$ A function returns computes the maximal elements of a set S in respect to a relation $\preceq$.

prune(G) A function that removes all rules from a regular tree grammar, that are not reachable from the start symbol.

INHABIT (Algorithm 2) computes a regular tree grammar, that represents the set of inhabitants of a type τ relative to a combinator repository Γ [3, 40]. It calls SUBQUERIES (Algorithm 1), to compute the minimal number of types, that need to be inhabited for a combinator to produce its target type. INHABIT differs from the problems stated in Definitions 18 and 19 in one important aspect. If the type ω is queried, the algorithm will deem it as not inhabitable. This differs from the theory, as ω is inhabitable by any combinatory term. Since this produces all terms, and therefore no useful information, we ignore this case, as this makes inhabitation simpler.

Algorithm 1: SUBQUERIES: Generating subqueries for a set of multi-arrow

Input: A set of multi-arrows M, an intersection type τ and a relation S of explicitly given subtypes on constructors

Output: A set of intersection types

Function SUBQUERIES(M, τ, S)**:**

$\quad$ $\mathit{covers} \leftarrow \mathit{minimal_covers}(M, \mathbb{P}(\tau), \leq_S^{tgt})$

$\quad$ $\mathit{subqueries} \leftarrow \emptyset$

$\quad$ **for** $\{((\sigma_1^1, \ldots, \sigma_n^1), \tau^1), \ldots ((\sigma_1^m, \ldots, \sigma_n^m), \tau^m)\} \in \mathit{covers}$ **do**

$\quad\quad$ $\mathit{subqueries} = \mathit{subqueries} \cup \{(\sigma_1^1 \cap \ldots \cap \sigma_1^m, \ldots, \sigma_n^1 \cap \ldots \cap \sigma_n^m)\}$

$\quad$ **return** $\mathit{maximal_elements}(\mathit{subqueries}, \leq_S)$

Example 16: Run of INHABIT for FCL

Consider the synthesis problem in Example 11. We show a step-by-step computation of a regular tree grammar using INHABIT$(\Gamma, e, \emptyset)$.

1. We initialize $G = \emptyset$, $\mathit{queue} = \{e\}$, $\mathit{seen} = \emptyset$.

2. The only element in the queue is e, so we start with $\rho = e$ and mark e as seen.

3. We iterate over the combinators in Γ:

    - $X : ((b \cap d) \to c) \to e$, we have the following sets of multi-arrows:

        – $\{((), ((b \cap d) \to c) \to e)\}$: Since $\mathbb{P}(e) = \{e\}$ and $((b \cap d) \to c) \to e \not\leq e$, the set $\{e\}$ cannot be covered and the set of subqueries is empty.

Algorithm 2: INHABIT: Generating a regular tree grammar

Input: A combinator repository Γ, an intersection type τ and a relation S of explicitly given subtypes on constructors
Output: A regular tree grammar G
Function INHABIT(Γ, τ, S)**:**

```
G ← ∅
queue ← {τ}
seen ← ∅
for ρ ∈ queue \ seen do
    if ρ = ω then
        continue
    seen ← seen ∪ {ρ}
    for C : ρ' ∈ Γ do
        for n-ary ∈ marrow(ρ') do
            args ← SUBQUERIES(n-ary, ρ, S)
            if |args| > 0 then
                for (σ_1, ..., σ_m) ∈ args do
                    G ← G ∪ {ρ ↦ C σ_1 ... σ_m}
                    queue ← queue ∪ {σ_1, ..., σ_m}
return prune(G)
```

- $\{(((b \cap d) \to c), e)\}$: The target set $\{e\}$ can be covered by the set of multi-arrows (since $e \le e$), so the set *args* of subqueries is $\{((b \cap d) \to c)\}$. We add a rule

$$e \mapsto X\,((b \cap d) \to c)$$

and add $(b \cap d) \to c$ to the queue.

- $Y : a \to (b \cap d) \to (c \cap e)$, we have the following sets of multi-arrows:
  - $\{((), a \to (b \cap d) \to e), ((), a \to (b \cap d) \to c)\}$: Since both $a \to (b \cap d) \to e)$ and $a \to (b \cap d) \to c$ are not a subtype of e, the set $\{e\}$ cannot be covered and the set of subqueries is empty.
  - $\{((a), (b \cap d) \to e), ((a), (b \cap d) \to c)\}$: The target set cannot be covered, so the set of subqueries is empty.
  - $\{((a, b \cap d), e), ((a, b \cap d), c)\}$: The set $\{e\}$ can be covered by the multi-arrow $((a, b \cap d), e)$, so the set *args* of subqueries is $\{(a, b \cap d)\}$. We add a rule

$$e \mapsto Y\,a\,(b \cap d)$$

and add a and $b \cap d$ to the queue.

- $Z : a \cap b \cap d$, no set of multi-arrows can cover the target set $\{e\}$.

4. We continue with the next element of the queue, that we did not already see, we take $(b \cap d) \to c$ as the next element and iterate over the combinators in Γ:
   - $X : ((b \cap d) \to c) \to e$, no set of multi-arrows can cover the target set $\{(b \cap d) \to c\}$.
   - $Y : a \to (b \cap d) \to (c \cap e)$, we have the following sets of multi-arrows:
     - $\{((), a \to (b \cap d) \to e), ((), a \to (b \cap d) \to c)\}$: The target set cannot be covered, so the set of subqueries is empty.
     - $\{((a), (b \cap d) \to e), ((a), (b \cap d) \to c)\}$: The target set can be covered by $((a), (b \cap d) \to c)$, so the set of subqueries is $\{(a)\}$. We add a rule

$$((b \cap d) \to c) \mapsto Y\,a$$

and add a to the queue.

  - $\{((a, b \cap d), e), ((a, b \cap d), c)\}$: The target set cannot be covered, so the set of subqueries is empty.
- $Z : a \cap b \cap d$, no set of multi-arrows can cover the target set.

5. We take the next element a from the queue and iterate over the combinators in Γ:
   - $X : ((b \cap d) \to c) \to e$, no set of multi-arrows can cover the target set $\{a\}$.
   - $Y : a \to (b \cap d) \to (c \cap e)$, no set of multi-arrows can cover the target set $\{a\}$.
   - $Z : a \cap b \cap d$, we have the following sets of multi-arrows:
     - $\{((), a), ((), b), ((), d)\}$ can cover the target set $\{a\}$, with the multi-arrow $((), a)$. Since the list of arguments is empty, the set of subqueries is $\{()\}$. We add a rule

$$a \mapsto Z$$

6. We take the next element $b \cap d$ from the queue and iterate over the combinators in Γ:
   - $X : ((b \cap d) \to c) \to e$, no set of multi-arrows can cover the target set $\{b \cap d\}$.
   - $Y : a \to (b \cap d) \to (c \cap e)$, no set of multi-arrows can cover the target set $\{b \cap d\}$.
   - $Z : a \cap b \cap d$, we have the following sets of multi-arrows:
     - $\{((), a), ((), b), ((), d)\}$ can cover the target set $\mathbb{P}(b \cap d) = \{b, d\}$, with the multi-arrow $((), b)$ and $((), d)$. Since the list of arguments is empty, the set of subqueries is $\{()\}$. We add a rule

$$(b \cap d) \mapsto Z$$

7. There are no more elements in the queue, that are not seen. All rules are reachable from the start symbol, so we return the following regular tree grammar:

$$\begin{aligned} G = \{e &\mapsto X\,((b \cap d) \to c), \\ e &\mapsto Y\,a\,(b \cap d), \\ ((b \cap d) \to c) &\mapsto Y\,a, \\ a &\mapsto Z, \\ (b \cap d) &\mapsto Z\} \end{aligned}$$

We now derive the following terms from the grammar:

$$\begin{aligned} &e \Rightarrow X\,((b \cap d) \to c) \Rightarrow X\,(Y\,a) \Rightarrow X\,(Y\,Z) \\ &e \Rightarrow Y\,a\,(b \cap d) \Rightarrow Y\,Z\,(b \cap d) \Rightarrow Y\,Z\,Z \end{aligned}$$

We also give the regular tree grammar for the running maze example.

Example 17: Regular Tree Grammar for Maze Solving

Consider the combinator repository Γ from Example 10 and the type $\mathit{path}(4 \times 4)$. We give the regular tree grammar computed by $\textsc{Inhabit}(\Gamma, \mathit{Path}, \emptyset)$.

$$\begin{aligned} R = \{\mathit{path}(4 \times 4) &\mapsto \textsc{Down}\,\mathit{free}(4 \times 4)\,\mathit{path}(4 \times 3), \\ \mathit{path}(4 \times 4) &\mapsto \textsc{Left}\,\mathit{free}(4 \times 4)\,\mathit{path}(3 \times 4), \\ \mathit{path}(4 \times 3) &\mapsto \textsc{Down}\,\mathit{free}(4 \times 3)\,\mathit{path}(4 \times 2), \\ \mathit{path}(4 \times 3) &\mapsto \textsc{Up}\,\mathit{free}(4 \times 3)\,\mathit{path}(4 \times 4), \\ \mathit{path}(3 \times 4) &\mapsto \textsc{Left}\,\mathit{free}(3 \times 4)\,\mathit{path}(2 \times 4), \\ \mathit{path}(3 \times 4) &\mapsto \textsc{Right}\,\mathit{free}(3 \times 4)\,\mathit{path}(4 \times 4), \\ \mathit{path}(4 \times 2) &\mapsto \textsc{Down}\,\mathit{free}(4 \times 2)\,\mathit{path}(4 \times 1), \\ &\vdots \\ \mathit{path}(0 \times 0) &\mapsto \textsc{Start}\} \cup \\ \{\mathit{free}(x \times y) &\mapsto \textsc{Free}_{x,y} \mid (x, y) \in \{0, \dots, 5\}^2 \\ &\setminus \{(1,1), (2,1), (3,1), (1,3), (2,3), (3,3)\}\} \end{aligned}$$

For brevity, we only give the rules for the first few positions. It is important to note, that while rules to blocked positions are added to the grammar, they

are not productive since they lack a corresponding FREE combinator, so they are removed in the pruning step. Similarly, rules from blocked positions are added to the grammar and pruned, since there is no path to the blocked position.

3.3.2. Enumeration

The algorithm presented in Algorithm 2 computes a regular tree grammar, that represents the set of inhabitants of a type τ relative to a combinator repository Γ. We give a simple algorithm, that lazily enumerates a regular tree grammar bottom-up. We use the keyword **yield** to denote the output of a term, while the algorithm continues to run. Algorithm 3 outputs a sequence of terms, that are derivable from a regular tree grammar. We use the mapping *terms* from non-terminal symbols to sets of terms, to store the terms already found, and build new terms for a rule $A \mapsto f\, B_1, \ldots B_n$ by building a product over the already found terms for $B_1 \ldots B_n$. If A is the starting symbol, the term is outputted. We use a Boolean variable *have_new_terms* to determine, if the algorithm has found new terms in the current iteration, if not, the algorithm terminates.

There are other ways to enumerate a regular tree grammar, such as *Feat* [25] that was used in prior implementations [3], but this does not align well with the extension we introduce in Chapter 4.

We show a step-by-step example of the enumeration of the regular tree grammar.

Example 18: Run of ENUMERATE for FCL

We enumerate the regular tree grammar shown in Example 16 using Algorithm 3.

1. We initialize *terms* with $\{e \mapsto \emptyset, (b \cap d) \to c \mapsto \emptyset, a \mapsto \emptyset, b \cap d \mapsto \emptyset\}$ and set *have_new_terms* to true and start the main loop.

2. Resetting *have_new_terms* to false, we start iterating over the rules of the regular tree grammar:

   a) We start with the rule $e \mapsto X\, ((b \cap d) \to c)$. Since no terms for $(b \cap d) \to c$ are found, the product is empty and no new terms are found.

   b) The same holds for $e \mapsto Y\, a\, (b \cap d)$ and $(b \cap d) \mapsto Y\, a$.

   c) The rule $a \mapsto Z$ does not require any arguments, so the product yields exactly the empty tuple. We add the term Z to the set of terms for a and set *have_new_terms* to true.

Algorithm 3: ENUMERATE: Enumerating a regular tree grammar

Input: A regular tree grammar G

Output: A sequence of terms

```
Function ENUMERATE(G = (N, Σ, S, R)):
    terms ← {n ↦ ∅ | n ∈ N}
    have_new_terms ← True
    while have_new_terms do
        have_new_terms ← False
        for s ↦ C B_1 ... B_n ∈ R do
            for (M_1, ..., M_n) ∈ Π_{i=1}^n terms[B_i] do
                if C M_1 ... M_n ∉ terms[s] then
                    terms[s] ← terms[s] ∪ {C M_1 ... M_n}
                    have_new_terms ← True
                    if s = S then
                        yield C M_1 ... M_n
```

d) Similarly, for $(b \cap d) \mapsto Z$, we add the term Z to the set of terms for $b \cap d$.

3. We have found new terms, so we continue with the next iteration of the main loop.

   a) We start with the rule $e \mapsto X\,((b \cap d) \to c)$. Since no terms for $(b \cap d) \to c$ are found, the product is empty and no new terms are found.

   b) For the rule $e \mapsto Y\,a\,(b \cap d)$, we have $\mathit{terms}[a] = \{Z\}$ and $\mathit{terms}[b \cap d] = \{Z\}$. The tuple (Z, Z) is the only member of the product of these sets, so we add $Y\,Z\,Z$ to the terms for e. Since e is the starting symbol, we output this term.

   c) For the rule $(b \cap d) \to c \mapsto Y\,a$, we have $\mathit{terms}[a] = \{Z\}$. The product contains exactly the single arity product (Z), so we add $Y\,Z$ to the terms for $(b \cap d) \to c$.

   d) The rule $a \mapsto Z$ would produce the term Z, but this term is already in the set of terms for a, so no new terms are found.

   e) The same holds for $(b \cap d) \mapsto Z$.

4. Since we have found new terms in the last iteration, we start anew.
   a) For the rule $e \mapsto X\,((b \cap d) \to c)$, we have $\mathit{terms}[(b \cap d) \to c] = \{Y\,Z\}$. The product contains exactly the single arity product $(Y\,Z)$, so we add $X\,(Y\,Z)$ to the terms for e. Since e is the starting symbol, we output this term.
   b) No new terms are found for the other rules.
5. We start the next iteration of the main loop, but no new terms are found in this iteration.
6. Since no new terms are found, the algorithm terminates.

The terms, that are outputted by the algorithm, are $Y\,Z\,Z$ and $X\,(Y\,Z)$.

4. Finite Combination Logic with Predicates

In this chapter, we extend the Finite Combination Logic (FCL) with predicates, parameters and literals, to form the Finite Combination Logic with Predicates (FCLP) as introduced in [24]. We extend both the notion of terms and intersection types to allow for literals, introduce parameterized types and define a type system for FCLP. Many definitions and concepts of FCL naturally extend to FCLP, but we give the full definitions for completeness. As in the previous chapter, we give examples of the new concepts and show how they can be used to express specifications in a more concise and natural way. Finally, we extend the inhabitation algorithm (Inhabit, Algorithm 2) and enumeration algorithm (Enumerate, Algorithm 3) of FCL to incorporate the new additions to the type system. While much of the expressiveness, that FCLP adds, can be encoded in FCL, FCLP provides a more natural way to express certain specifications. The theoretical foundation of FCLP is based on the work in [24], which the author of this thesis contributed to. This chapter extends this foundation by the notion of the parameterized tree grammar (Definition 41), the representation of interpretations as algebras over signatures and the inhabitation and enumeration algorithm in Section 4.3.

At its core, FCLP conservatively extends FCL by allowing types in a combinator repository to be parameterized types, that allow for a limited form of dependent types in three ways: First, they can make use of variables, that range over a finite set of *literals*. Both these variables and literals can occur in types and can be understood as a stratification of that type, i.e., one type for each literal. Second, they can make use of for quantifiers for terms, that range over inhabitants of a given type. In contrast to variables introduced by literal quantifiers, variables introduced term quantifiers cannot occur as part of an intersection type. Lastly, they can make use of predicates, that can depend on these variables, and act as filters on the set of possible instantiations for the variables. Furthermore, the values of the variables are carried on to the term level.

4.1. Terms and Parameterized Types

Terms in FCLP follow the syntax of FCL, but with the addition of allowing for elements of a set of literals to occur in argument position. Literals are similar to constructor symbols in FCL, but are not applied to any arguments. Formally, a literal can be any symbol, in practice these symbols are often drawn from numbers, strings or tuples.

We allow for literals to occur in terms and types. For this, we extend the notion of combinatory terms to include literals.

Definition 27: Combinatory Terms With Literals

Given a finite set of combinators $\mathcal{C}$ and a finite set of literals $\mathcal{L}$, the set of *combinatory terms with literals* is defined by the following grammar:

$$T_{\mathcal{C}} \ni M, N ::= C \mid (M\, T)$$
$$T ::= M \mid l$$

with $C \in \mathcal{C}$ and $l \in \mathcal{L}$.

Similarly, we allow for literals to occur in types. We also allow for literal variables to occur in types.

Definition 28: Intersection Types With Literals

Given an enumerable set of unary constructor symbols $\mathbb{C}$ and a set $\mathcal{V}$ of variable names, the set of *intersection types with literals* is defined by the following grammar:

$$\mathbb{T} \ni \sigma, \tau ::= (\sigma \to \tau) \mid (\sigma \cap \tau) \mid c(\tau) \mid \omega \mid l \mid \alpha$$

with $c \in \mathbb{C}$, $\alpha \in \mathcal{V}$, ω as the universal type and $l \in \mathcal{L}$.

We use the same notation for types and terms as in FCL (Definitions 1 and 14), but with the addition of literals. Similarly, we use the same notion of subtyping as in FCL. No new rules for literals are needed, as literals are subtypes of themselves. For the remainder of this thesis, we implicitly refer to combinatory terms with literals and intersection types with literals, when we refer to terms and intersection types, respectively.

Literals are organized in groups. These groups are disjoint and identified by a unique name and are contained in a *literal repository*.

Definition 29: Literal Repository

Given a finite set of literals $\mathcal{L}$ and a finite set of identifiers $\mathcal{I}_l$, a *literal repository* $\Delta : \mathcal{L} \to \mathcal{I}_l$ is a function, that assigns to each literal to an identifier:

While formally a literal repository is a function, we usually give Δ as the inverse mapping of identifiers to sets of literals, where the sets of literals are disjoint.

We use the notation $l : t$ to denote that the literal l is in the group identified by t, when Δ is clear from the context.

We introduce parameterized types as a way to specify types that depend on literals and terms. Parameterized types add quantifiers and predicates in a prenex form to intersection types. We allow variables to range over literals and terms and to filter on the set of possible substitutions by predicates.

Definition 30: Parameterized Types

Given The following grammar defines the set of *parameterized types*:

$$\mathbb{P} \ni \varphi, \psi ::= \tau \mid \langle \alpha : t \rangle \Rightarrow \varphi \mid \langle\langle x : \tau \rangle\rangle \Rightarrow \varphi \mid P \Rightarrow \varphi$$

with τ as an intersection type, α as a literal variable, x as a term variable, P as a decidable predicate and t as an identifier of a group of literals.

Intuitively, the new constructs can be understood as follows:

Literal Quantifiers ($\langle \alpha : t \rangle \Rightarrow \varphi$): This introduces a literal variable α that ranges over the group of literals identified by t. The type φ can depend on α. In particular, α can be used both in predicates and in intersection types contained in φ.

Term Quantifiers ($\langle\langle x : \tau \rangle\rangle \Rightarrow \varphi$): This introduces a term variable x that ranges over inhabitants of the type τ. As with literal quantifiers, the type φ can depend on x. In contrast to literal quantifiers, x cannot be used in intersection types, but only in predicates.

Predicates ($P \Rightarrow \varphi$): P is a predicate, that must hold for the type φ to be valid. The predicate can depend on literal and term variables, that are introduced by the quantifiers. We do (intentionally) not specify a syntax for the predicates, but in this thesis, we give them as a first-order logic formula. To separate the formulae from the type, we add parentheses around the formulae.

We call the quantifiers *parameters*, to distinguish them from the notion of arguments as introduced in FCL. This distinction carries on to the term level, where we also call

subterms, that are derived by literal or term quantifiers, parameters and subterms, that are derived by types in argument position, arguments.

The following examples give an intuition of the form of specification that can be expressed with parameterized types.

Example 19: Parameterized Types

The following are examples of parameterized types:

- $\langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (\alpha = \beta + 1) \Rightarrow c(\alpha) \rightarrow c(\beta)$
  Given $\mathbb{N}_{\leq 3}$ as the group of natural numbers up to 3, this is a type for terms, that maps terms of type $c(1)$ to terms of type $c(0)$, terms of type $c(2)$ to terms of type $c(1)$ and terms of type $c(3)$ to terms of type $c(2)$, with c being a constructor symbol. In essence, this type is very similar to the intersection type $(c(1) \rightarrow c(0)) \cap (c(2) \rightarrow c(1)) \cap (c(3) \rightarrow c(2))$.

- $\langle\langle x : a \rangle\rangle \Rightarrow (\mathit{size}(x) = 3) \Rightarrow b$
  Given $\mathit{size} : T_{\mathcal{C}} \rightarrow \mathbb{N}$ as a function, that computes the size of a term, this type specifies terms, whose first argument is of type a and has size 3.

- $\langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle\langle x : a \rangle\rangle \Rightarrow (\mathit{size}(x) = \alpha) \Rightarrow c(\alpha)$
  Similar to the previous example, this type specifies terms, whose first argument is of type a and has size equal to some size ≤ 3. It carries the information about the term size in its target type.

- $\langle\langle x : a \rangle\rangle \Rightarrow \langle\langle y : a \rangle\rangle \Rightarrow (x = y) \Rightarrow b$
  This type specifies terms, whose first and second argument are of type a and equal.

We define some convenience functions for parameterized types.

Definition 31: Convenience Functions for Parameterized Types

Given a parameterized type $p_0 \Rightarrow \ldots \Rightarrow p_n \Rightarrow \tau$, we define the following functions:

$$\begin{aligned}
\mathit{params}(\varphi) &= [(x,t) \mid p \in [p_0,\ldots,p_n], p \text{ is in the form } \langle x : t \rangle \text{ or } \langle\langle x : t \rangle\rangle] \\
\mathit{params}_t(\varphi) &= [(x,t) \mid p \in [p_0,\ldots,p_n], p \text{ is in the form } \langle\langle x : t \rangle\rangle] \\
\mathit{params}_l(\varphi) &= [(x,t) \mid p \in [p_0,\ldots,p_n], p \text{ is in the form } \langle x : t \rangle] \\
\mathit{preds}(\varphi) &= [p \mid p \in [p_0,\ldots,p_n], p \text{ is a predicate}] \\
\mathit{itype}(\varphi) &= \tau
\end{aligned}$$

The functions *params*, $params_t$ and $params_l$ preserve the order of the quantifiers in the parameterized type.

Intuitively, the function *params* returns all parameters in the parameterized type, while $params_t$ and $params_l$ return only term and literal parameters, respectively. The function *preds* returns all predicates in the parameterized type and the function *itype* returns the intersection type of the parameterized type.

Definition 32: Closed Parameterized Types

A parameterized type φ is *closed*, if every occurrence of a literal variable α in φ is bound by a $\langle \alpha : t \rangle \Rightarrow \psi$ quantifiers and every occurrence of a term variable x in φ is bound by a $\langle\!\langle x : \tau \rangle\!\rangle \Rightarrow \psi$ quantifier.
If a parameterized type is not close, we call it *open*.

We also introduce the notion of literal and term substitution for parameterized types.

Definition 33: Literal Substitution

Let $l \in \mathcal{L}$ be a literal and $\alpha \in \mathcal{V}$ be a literal variable. The substitution of a literal l for a literal variable α in a parameterized type φ is defined as follows:

$$
\begin{aligned}
(\langle \alpha' : t \rangle \Rightarrow \varphi)[\alpha := l] &= \langle \alpha' : t \rangle \Rightarrow \varphi[\alpha := l] && \textit{if } \alpha' \neq \alpha \\
(\langle \alpha' : t \rangle \Rightarrow \varphi)[\alpha := l] &= \langle \alpha' : t \rangle \Rightarrow \varphi && \textit{if } \alpha' = \alpha \\
(\langle\!\langle x : \tau \rangle\!\rangle \Rightarrow \varphi)[\alpha := l] &= \langle\!\langle x : \tau[\alpha := l] \rangle\!\rangle \Rightarrow \varphi[\alpha := l] \\
(P \Rightarrow \varphi)[\alpha := l] &= P[\alpha := l] \Rightarrow \varphi[\alpha := l] \\
(\sigma \to \tau)[\alpha := l] &= \sigma[\alpha := l] \to \tau[\alpha := l] \\
(\sigma \cap \tau)[\alpha := l] &= \sigma[\alpha := l] \cap \tau[\alpha := l] \\
(c(\tau))[\alpha := l] &= c(\tau[\alpha := l]) \\
\omega[\alpha := l] &= \omega \\
l'[\alpha := l] &= l' && \text{with } l' \in \mathcal{L} \\
\alpha[\alpha := l] &= l
\end{aligned}
$$

Definition 34: Multi Literal Substitution

Let θ be a mapping from literal variables to literals. The *multi literal substitution* $subst(\varphi, \theta)$ for a parameterized type φ substitutes all occurrences of literal variables in φ according to θ.

Since literal substitutions do not depend on each other, these substitutions can be applied in any order.

Definition 35: Term Substitution

Let M be a term and x be a term variable. The substitution of a term M for a term variable x in a parameterized type φ is defined as follows:

$$
\begin{aligned}
(\langle \alpha : t \rangle \Rightarrow \varphi)[x := M] &= \langle \alpha : t \rangle \Rightarrow \varphi[x := M] \\
(\langle\!\langle x' : \tau \rangle\!\rangle \Rightarrow \varphi)[x := M] &= \langle\!\langle x' : \tau \rangle\!\rangle \Rightarrow \varphi[x := M] && \textit{if } x' \neq x \\
(\langle\!\langle x' : \tau \rangle\!\rangle \Rightarrow \varphi)[x := M] &= \langle\!\langle x' : \tau \rangle\!\rangle \Rightarrow \varphi && \textit{if } x' = x \\
(P \Rightarrow \varphi)[x := M] &= P[x := M] \Rightarrow \varphi[x := M]) \\
\tau[x := M] &= \tau
\end{aligned}
$$

As with literal substitution, we define *multi term substitution* analogously.

Definition 36: Multi Term Substitution

Let θ be a mapping from term variables to terms. The *multi term substitution* $\textit{subst}(\varphi, \theta)$ for a parameterized type φ substitutes all occurrences of term variables in φ according to θ.

The important difference is, that literal substitutions are also applied to the intersection types contained in the parameterized type, while term substitutions are only applied to the predicates. Since term variables do not depend on each other, these substitutions can be applied in any order.

We extend the notion of combinator repositories from intersection types to *closed parameterized types*.

Definition 37: Parameterized Combinator Repository

Given a finite set of combinators $\mathcal{C}$, a *parameterized combinator repository* is a function $\Gamma : \mathcal{C} \to \mathbb{P}$, that assigns a closed parameterized type to each combinator.

We use the same notation for parameterized combinator repositories as for combinator repositories in FCL (Definition 16).

Unless stated otherwise, we refer to the definitions of FCLP – as opposed to FCL – regarding intersection types, combinator repositories and combinatory terms.

4.2. Type System

To capture the intuition given in Example 19, we define the semantics of FCLP by giving the following set of rules.

Definition 38: Finite Combination Logic with Predicates (FCLP)

Given a combinator repository Γ and a literal repository Δ, we define the *finite combination logic with predicates* as the following set of typing judgments:

$$\frac{C : \varphi \in \Gamma}{\Gamma; \Delta \vdash C : \varphi}\ (\mathit{Var})$$

$$\frac{\Gamma; \Delta \vdash M : P \Rightarrow \varphi \quad P \text{ holds}}{\Gamma; \Delta \vdash M : \varphi}\ (PE)$$

$$\frac{\Gamma; \Delta \vdash : \langle \alpha : t \rangle \Rightarrow \varphi \quad \Delta(l) = t}{\Gamma; \Delta \vdash M\, l : \varphi[\alpha := l]}\ (\langle\rangle E)$$

$$\frac{\Gamma; \Delta \vdash M : \langle\!\langle x : \tau \rangle\!\rangle \Rightarrow \varphi \quad \Gamma; \Delta \vdash N : \tau}{\Gamma; \Delta \vdash M\, N : \varphi[x := N]}\ (\langle\!\langle\rangle\!\rangle E)$$

$$\frac{\Gamma; \Delta \vdash M : \sigma \quad \sigma \leq \tau}{\Gamma; \Delta \vdash M : \tau}\ (\leq)$$

$$\frac{\Gamma; \Delta \vdash M : \sigma \to \tau \quad \Gamma; \Delta \vdash N : \sigma}{\Gamma; \Delta \vdash M\, N : \tau}\ (\to E)$$

As with FCL, the intersection introduction rule $(\cap I)$ and intersection elimination rule $(\cap E)$ are omitted, as they can be derived from the rules given above. Note, that the rule $(\langle\rangle E)$ transports the literal l from the type level to the term level, so a term carries information about the literal chosen in the substitution. Similarly, the rule $(\langle\!\langle\rangle\!\rangle E)$ transports the term N from the type level to the term level, acting similar to the $(\to E)$ rule.

When ignoring the literal repository Δ, the rules (Var), $(\leq)$, $(\to E)$ form exactly the rules of FCL. This makes FCLP a conservative extension of FCL.

Lemma 7

Given a combinator repository Γ, that only contains intersection types, a combinatory term M without literals, and an intersection type τ, $\Gamma; \emptyset \vdash M : \tau$ is derivable in FCLP iff $\Gamma \vdash M : \tau$ is derivable in FCL.

Proof. The proof is by induction over the respective derivations. □

4.2.1. Examples

In this subsection, we revisit the example types given in Example 19, include them in a combinator repository, give a type derivation for an example type and compare the combinator repositories modeled in FCLP with equivalent versions in FCL.

Example 20: Repository with Literal Quantifiers and Predicates

Consider the following combinator repository Γ and literal repository Δ:

$$\begin{aligned} \Delta &= \{\mathbb{N}_{\leq 3} : \{0, 1, 2, 3\}\} \\ \Gamma &= \{C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (\alpha = \beta - 1) \Rightarrow c(\alpha) \to c(\beta), \\ &\qquad C_0 : c(0)\} \end{aligned}$$

We can derive the following typing judgment:

$$\Gamma; \Delta \vdash C\,2\,3(C\,1\,2(C\,0\,1\,C_0)) : c(3)$$

We do not give the full derivation here (due to space constraints), but focus on the following subderivation, that shows the substitution of literals in this example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta \vdash C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (\alpha = \beta - 1) \Rightarrow c(\alpha) \to c(\beta)}\,(Var)}{\Gamma; \Delta \vdash C\,0 : \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (0 = \beta - 1) \Rightarrow c(0) \to c(\beta)}\,(\langle\rangle E)}{\Gamma; \Delta \vdash C\,0\,1 : (0 = 1 - 1) \Rightarrow c(0) \to c(1)}\,(\langle\rangle E) \quad 0 = 1 - 0}{\Gamma; \Delta \vdash C\,0\,1 : c(0) \to c(1)}\,(PE) \quad \dfrac{}{\Gamma; \Delta \vdash C_0 : c(0)}\,(Var)}{\Gamma; \Delta \vdash C\,0\,1\,C_0 : c(1)}\,(\to E)$$

A similar derivation can be given for the whole term.
A version of the repository modeled in FCL would be the following:

$$\begin{aligned} \Gamma_{FCL} = \{&0 : 0, \quad 1 : 1, \quad 2 : 2, \quad 3 : 3, \\ &C : (0 \to 1 \to c(0) \to c(1)) \\ &\qquad \cap (1 \to 2 \to c(1) \to c(2)) \\ &\qquad \cap (2 \to 3 \to c(2) \to c(3)), \\ &C_0 : c(0)\} \end{aligned}$$

While the above example can be modeled in FCL, the use of literals and quantifiers allow for a more concise and natural way to express the specification. Note, that in FCL we use the numbers $0, 1, 2, 3$ both as type constructors, that are implicitly applied to ω, as well as combinators.

Example 21: Repository With Term Quantifiers and Predicates

Let $size : T_{\mathcal{C}} \to \mathbb{N}$ be the function, that returns the length of a term. Consider the following combinator repository Γ:

$$\Gamma = \{C : \langle\langle x : a \rangle\rangle \Rightarrow (size(x) = 3) \Rightarrow b, \\ X : a \cap (a \to a)\}$$

We derive the following typing judgment

$$\Gamma; \emptyset \vdash C\,(X\,(X\,X) : b$$

by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\Gamma;\emptyset \vdash C : \langle\langle x : a\rangle\rangle \Rightarrow (size(x) = 3) \Rightarrow b}\,(Var) \quad \dfrac{\dots}{\Gamma;\emptyset \vdash X\,(X\,X) : a}}{\Gamma;\emptyset \vdash C\,(X\,(X\,X)) : (size(X\,(X\,X)) = 3) \Rightarrow b}\,(\langle\langle\rangle\rangle E) \quad size(X\,(X\,X)) = 3}{\Gamma;\emptyset \vdash C\,(X\,(X\,X)) : b}\,(PE)$$

When modeled in FCL, the combinator repository would be:

$$\Gamma_{FCL} = \{C : a(2) \to b, \\ X : a(0) \cap (a(0) \to a(1)) \cap (a(1) \to a(2))\}$$

This example shows both the use of term quantifiers and predicates without the need of a literal repository. Most importantly, while it is possible to model this example in FCL, the fact, that the first parameter to C has length 3 is not obvious from the type of C. Additionally, the fact that the length of a term with type a is important must be encoded in the type of X, while the restriction only occurs as a subterm starting with a C. This makes it harder to specify, if an X-term of different length is required in the same repository.

Example 22: Repository With Literal and Term Quantifiers and Predicates

Let $size : \mathbb{C} \to \mathbb{N}$ be the function, that returns the length of a term.
Consider the following combinator repository Γ and literal repository Δ:

$$\Delta = \{\mathbb{N}_{\leq 3} : \{0, 1, 2, 3\}\} \\ \Gamma = \{C : \langle \alpha : \mathbb{N}_{\leq 3}\rangle \Rightarrow \langle\langle x : a\rangle\rangle \Rightarrow (size(x) = \alpha) \Rightarrow c(\alpha), \\ X : a \cap (a \to a)\}$$

We derive the following typing judgment

$$\Gamma; \Delta \vdash C\, 2\, (X\, X) : c(2)$$

with the accompanying derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \Delta \vdash C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle\!\langle x : a \rangle\!\rangle \Rightarrow (\mathit{size}(x) = 2) \Rightarrow c(\alpha)}\,(\mathit{Var})}{\Gamma; \Delta \vdash C\, 2 : \langle\!\langle x : a \rangle\!\rangle \Rightarrow (\mathit{size}(x) = 2) \Rightarrow c(2)}\,(\langle\rangle E) \qquad \dfrac{\dots}{\Gamma; \Delta \vdash X\, X : a}}{\Gamma; \Delta \vdash C\, 2\, (X\, X) : (\mathit{size}(X\, X) = 2) \Rightarrow c(2)}\,(\langle\!\langle\rangle\!\rangle E)}{\Gamma; \Delta \vdash C\, 2\, (X\, X) : c(2)}\,(PE)$$

When modeled in FCL, the combinator repository would be:

$$\begin{aligned} \Gamma_{FCL} = \{ & 0 : 0, \quad 1 : 1, \quad 2 : 2, \quad 3 : 3, \\ & C : (3 \to a(3) \to a(3)) \cap (2 \to a(2) \to c(2)) \cap (1 \to a(1) \to c(1)), \\ & X : a(1) \cap (a(1) \to a(2)) \cap (a(2) \to a(3)) \} \end{aligned}$$

This example shows how the use of literal variables interacts with term variables in predicates. We also see, that to encode the same example in FCL, the type of every combinator must be aware of the length of their respective inhabited terms. It is easy to see, that when using more complex predicates and larger literal repositories, the model in FCL becomes unwieldy. In fact, there are even specifications, that can be expressed in FCLP, but not in FCL.

Example 23: Repository With Equality Constraints

Consider the following combinator repository Γ:

$$\begin{aligned} \Gamma = \{ & C : \langle\!\langle x : a \rangle\!\rangle \Rightarrow \langle\!\langle y : a \rangle\!\rangle \Rightarrow (x = y) \Rightarrow b, \\ & X : a \cap (a \to a) \} \end{aligned}$$

We derive the following typing judgment

$$\Gamma; \emptyset \vdash C\ X\ X : b$$

by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma; \emptyset \vdash C : \langle\!\langle x : a \rangle\!\rangle \Rightarrow \langle\!\langle y : a \rangle\!\rangle \Rightarrow (x = y) \Rightarrow b}\,(\mathit{Var})}{\Gamma; \emptyset \vdash C\ X : \langle\!\langle y : a \rangle\!\rangle \Rightarrow (X = y) \Rightarrow b}\,(\langle\!\langle\rangle\!\rangle E) \qquad X = X}{\Gamma; \emptyset \vdash C\ X\ X : (X = X) \Rightarrow b}\,(\langle\!\langle\rangle\!\rangle E)}{\Gamma; \emptyset \vdash C\ X\ X : b}\,(PE)$$

Example 23 shows a specification, that can be expressed in FCLP, but not in FCL, since the set of inhabitants in FCL form a regular tree language [40, Corollary 11], that cannot express equality constraints on subterms [11].

Finally, we model our running example of solutions for mazes in FCLP.

Example 24: Repository for Maze Solving in FCLP

Let UP, DOWN, LEFT, RIGHT, START be the combinators for moving in a maze. Consider the following combinator repository Γ and literal repository Δ:

$$\begin{aligned}
\Delta = \{ & \\
& pos : \{0,\dots,5\}^2 \setminus \{(1,1),(2,1),(3,1),(1,3),(2,3),(3,3)\} \\
& \} \\
\Gamma = \{ & \\
\textsc{Up} : & \langle \alpha : pos \rangle \Rightarrow \langle \beta : pos \rangle \Rightarrow (\beta = (\pi_1(\alpha), \pi_2(\alpha) - 1)) \Rightarrow path(\alpha) \to path(\beta), \\
\textsc{Down} : & \langle \alpha : pos \rangle \Rightarrow \langle \beta : pos \rangle \Rightarrow (\beta = (\pi_1(\alpha), \pi_2(\alpha) + 1)) \Rightarrow path(\alpha) \to path(\beta), \\
\textsc{Left} : & \langle \alpha : pos \rangle \Rightarrow \langle \beta : pos \rangle \Rightarrow (\beta = (\pi_1(\alpha) - 1, \pi_2(\alpha))) \Rightarrow path(\alpha) \to path(\beta), \\
\textsc{Right} : & \langle \alpha : pos \rangle \Rightarrow \langle \beta : pos \rangle \Rightarrow (\beta = (\pi_1(\alpha) + 1, \pi_2(\alpha))) \Rightarrow path(\alpha) \to path(\beta), \\
\textsc{Start} : & \; pos(0,0) \\
& \}
\end{aligned}$$

Where π_1 and π_2 are the projections on the first and second component of a pair. The following typing judgement is derivable:

$$\begin{aligned}
\Gamma; \Delta \vdash & \textsc{Down}\,(4,3)\,(4,4)\,(\textsc{Down}\,(4,2)\,(4,3)\,(\textsc{Right}\,(3,2)\,(4,2)\,(\\
& \quad \textsc{Right}\,(2,2)\,(3,2)\,(\textsc{Right}\,(2,1)\,(2,2)\,(\textsc{Right}\,(2,0)\,(2,1)\,(\\
& \qquad \textsc{Down}\,(1,0)\,(2,0)\,(\textsc{Down}\,(0,0)\,(1,0)\,\textsc{Start})))))) : pos(4,4)
\end{aligned}$$

Comparing the combinator repository in Example 24 with the one in FCL (Example 10), we see that we do not need a shorthand notation to iterate over all possible positions for the movement combinators, but can include this inside the type syntax. Another crucial difference is, that we do not need to encode the maze as a part of Γ. Therefore, the combinators can be used in a more general context. The maze itself is encoded in the literal repository Δ. This makes reusing the combinator repository for different problem instances easier. Especially this last point is a recurring pattern in the use of FCLP: The combinator repository Γ defines a general problem, and the literal repository Δ represents an input to that problem.

As with the terms of Example 10, we need to update the *Maze*-signature and algebra to include literals.

$$\begin{aligned} Maze_{\text{FCLP}} = (\{path\}, (pos : \mathbb{N}^2 \setminus \{(1,1),(2,1),(3,1),(1,3),(2,3),(3,3)\}), \\ \{\textsc{Start} : 1 \to path, \\ \textsc{Up} : pos \times pos \times path \to path, \\ \textsc{Down} : pos \times pos \times path \to path, \\ \textsc{Left} : pos \times pos \times path \to path, \\ \textsc{Right} : pos \times pos \times path \to path\}) \end{aligned}$$

$$\begin{aligned} Path_{\text{FCLP}} = (Path_{\text{FCLP},path} = (\mathbb{N}^2)^*), \{\textsc{Start}^{Path_{\text{FCLP}}}() = ((0,0)) \\ \textsc{Up}^{Path_{\text{FCLP}}}(\alpha, \beta, (p_0, \ldots, p_l)) = (p_0, \ldots, p_l, \alpha) \\ \textsc{Down}^{Path_{\text{FCLP}}}(\alpha, \beta, (p_0, \ldots, p_l)) = (p_0, \ldots, p_l, \alpha) \\ \textsc{Left}^{Path_{\text{FCLP}}}(\alpha, \beta, (p_0, \ldots, p_l)) = (p_0, \ldots, p_l, \alpha) \\ \textsc{Right}^{Path_{\text{FCLP}}}(\alpha, \beta, (p_0, \ldots, p_l)) = (p_0, \ldots, p_l, \alpha)\}) \end{aligned}$$

We see, that the notion of literals aligns well with the notion of base sorts of the signature.

4.3. Synthesis in FCLP

In this section, we discuss the synthesis problem for FCLP. It follows a similar structure as the synthesis problem for FCL, but with the addition of literals, quantifiers, and predicates. First, we define the inhabitation problem for FCLP. Notably, we do not inhabit parameterized types, but only intersection types.

Definition 39: Inhabitation in FCLP

Given a combinator repository Γ, a literal repository Δ and an intersection type τ, the *inhabitation problem* for FCLP is the decision problem, whether there exists a combinatory term M such, that $\Gamma; \Delta \vdash M : \tau$.

We refer to the inhabitation problem for FCLP via the following syntax:

$$\Gamma; \Delta \vdash ? : \tau$$

Build on the inhabitation problem, we define the synthesis problem for FCLP.

Definition 40: Synthesis in FCLP

Given a combinator repository Γ, a literal repository Δ and an intersection type τ, the *synthesis problem* for FCLP is the problem, of finding all combinatory terms M, such that $\Gamma; \Delta \vdash M : \tau$.

We refer to the synthesis problem for FCLP via the following syntax:

$$\{M \mid \Gamma; \Delta \vdash M : \tau\}$$

Unfortunately, the inclusion of predicates in the type system makes the synthesis problem in general undecidable, even if each predicate is decidable.

Lemma 8

Inhabitation for FCLP is undecidable, but semi-decidable.

Proof. The proof is given in [24, Theorems 19 and 20]. □

This does not pose a big problem in the context of practical synthesis, since the semi-decidability of the inhabitation problem is sufficient for practical applications. There is, however, a decidable fragment of the inhabitation problem, that is decidable, by limiting the predicates to equality and disequality constraints [24]. We do not discuss this fragment in this thesis, since the limitation of predicates is deemed too restrictive for practical applications.

Similar to FCL, we don't synthesize the (possible infinite) set of inhabitants directly, but instead generate a finite representation of this set. In contrast to FCL, we do not use a regular tree grammar, since the set of inhabitants is not a regular tree language [24, 11]. We will, however, use a similar approach to generate a finite representation of the set of inhabitants, by using a tree grammar, that is annotated with a set of predicates for each rule, in addition to meta-information about names of parameters. We do not examine the theoretical properties of this type of grammar in this thesis.

Definition 41: Parameterized Tree Grammar

A *parameterized tree grammar* is a tuple $G = (N, \Sigma, \mathcal{L}, S, R)$, where

- N is a finite set of non-terminal symbols,
- Σ is a finite set of terminal symbols,
- $\mathcal{L}$ is a finite set of literals,
- $S \in N$ is the start symbol,

- R is a finite set of rules of the form $A \mapsto f\, p_0 \, \ldots \, p_n \, B_1 \, \ldots \, B_m \lhd P$, with non-terminals $A, B_1, \ldots, B_n \in N$, $f \in \mathcal{F}$, each pair p_i either a pair of a term variable name and non-terminal symbol $(x_i : C_i)$ or a literal value $l_i \in \mathcal{L}$ and P a set of predicates.

Parameterized Tree Grammars are a closely related to definite clause grammars used the programming language Prolog, in that they extend traditional grammars and allow for predicates to be included in the grammar [38].

Similar to regular tree grammars, we do not give a formal definition of the derivation relation for parameterized tree grammars, but instead give an intuitive explanation. As with regular tree grammars, a derivation step in a parameterized tree grammar replaces a non-terminal symbol occurring as children with the right-hand side of a rule. The inclusion of term parameters (the pairs $(x_i : y_i)$) and predicates, requires special treatment. We gather all predicates in a derivation step and evaluate them at the end of a derivation. Instead of replacing term parameters directly by the right-hand side of a rule corresponding to y_i, we gather these as substitutions in a context and replace the term parameter by its variable name. The derivation relation is also lifted to the context. This context is used to substitute the variable names in the resulting term and the gathered predicates. A derivation is only valid, if all predicates in the rules hold.

Formally, this derivation relation can be modeled over a tuple (θ, P, M), where θ is a mapping of term variables to terms, P is a set of predicates and M is a term using the terminals and non-terminals.

Example 25: Derivation and Parameterized Tree Grammar With Trivial Predicates

Consider the repositories of Example 20 and the following synthesis request:

$$\{M \mid \Gamma; \Delta \vdash M : c(3)\}$$

The parameterized tree grammar of the inhabited terms is:

$$\begin{aligned} G = \{c(3) &\mapsto C\, 2\, 3\, c(2) \lhd \{2 = 3 - 1\} \\ c(2) &\mapsto C\, 1\, 2\, c(1) \lhd \{1 = 2 - 1\} \\ c(1) &\mapsto C\, 0\, 1\, c(0) \lhd \{0 = 1 - 1\} \\ c(0) &\mapsto C_0 \lhd \emptyset\} \end{aligned}$$

We derive the term $C\,2\,3\,(C\,1\,2\,(C\,0\,1\,C_0))$ as follows:

$$\begin{aligned}\emptyset, \emptyset, c(3) &\Rightarrow \emptyset, \{2 = 3-1\}, C\,2\,3\,c(2)\\ &\Rightarrow \emptyset, \{2 = 3-1, 1 = 2-1\}, C\,2\,3\,(C\,1\,2\,c(1))\\ &\Rightarrow \emptyset, \{2 = 3-1, 1 = 2-1, 0 = 1-1\}, C\,2\,3\,(C\,1\,2\,(C\,0\,1\,c(0)))\\ &\Rightarrow \emptyset, \{2 = 3-1, 1 = 2-1, 0 = 1-1\}, C\,2\,3\,(C\,1\,2\,(C\,0\,1\,C_0))\end{aligned}$$

Since all predicates hold, the derivation is valid.

We do not need to concern ourself with literal variables in the derivation, since they are anready resolved in the unrolling step.

Example 26: Derivation and Parameterized Tree Grammar With Non-trivial Predicates

Consider the repositories of Example 22 and the following synthesis request of

$$\{M \mid \Gamma; \Delta \vdash M : c(2)\}$$

The intermediate representation of the inhabited terms is:

$$\begin{aligned}G = \{c(2) &\mapsto C\,2\,(x : a) \lhd \{\mathit{size}(x) = 2\},\\ a &\mapsto X \lhd \emptyset\\ a &\mapsto X\,a \lhd \emptyset\}\end{aligned}$$

We derive the term $C\,2\,(X\,X)$ as follows:

$$\begin{aligned}\emptyset, \emptyset, c(2) &\Rightarrow \emptyset, \{\mathit{size}(x) = 2\}, C\,2\,(x : a)\\ &\Rightarrow \{x : X\,a\}, \{\mathit{size}(x) = 2\}, C\,2\,x\\ &\Rightarrow \{x : X\,X\}, \{\mathit{size}(x) = 2\}, C\,2\,x\end{aligned}$$

Since the predicate $(\mathit{size}(x) = 2)[x := X\,X]$ holds, the derivation of $C\,2\,(X\,X)$ is valid.

Example 27: Derivation and Parameterized Tree Grammar With Equality Constraints

Consider the repository of Example 23 and the following synthesis request of

$$\{M \mid \Gamma \vdash M : b\}$$

The parameterized tree grammar of the inhabited terms is:

$$\begin{aligned} G = \{b &\mapsto C\,(x : a)\,(y : a) \lhd \{x = y\} \\ a &\mapsto X \lhd \emptyset \\ a &\mapsto X\,a \lhd \emptyset\} \end{aligned}$$

We derive the term $C\,X\,X$ as follows:

$$\begin{aligned} \emptyset, \emptyset, b &\Rightarrow \emptyset, \{x = y\}, C\,(x : a)\,(y : a) \\ &\Rightarrow \{x : X\}, \{x = y\}, C\,x\,(y : a) \\ &\Rightarrow \{x : X, y : X\}, \{x = y\}, C\,x\,y \end{aligned}$$

Since the predicate $(x = y)[x := X, y := X]$ holds, the derivation of $C\,X\,X$ is valid.

4.3.1. Inhabitation

To synthesize terms for a given specification, we give a procedure, that generates a parameterized tree grammar whose derivable terms are the set of inhabitants of the specification. The procedure is similar to the one given for FCL, but adds special handling for parameters and predicates. Algorithm 5 gives the procedure for generating parameterized tree grammars in FCLP. In essence, we follow Algorithm 2, but with the following changes:

- Each literal variable is unrolled to all possible values in the literal repository. This associates each combinator with a set of parameterized terms, that are the result of all possible substitutions of literal variables.
- Each intersection type in a term parameter must also be inhabited for a rule to be added to the parameterized tree grammar.
- Predicates are gathered and added to the rules of the parameterized tree grammar.
- The subtyping relation is extended to include literals. A literal l is a subtype of a literal l', iff $l = l'$. We do not need to handle literal variables, since they are resolved in the unrolling step.
- Similarly, *marrows* (Definition 22) is extended to handle literals.
- The procedure *prune* is extended to handle parameterized grammars. It prunes all rules not reachable from the start symbol, assuming all predicates hold.

The subroutine UNROLL (Algorithm 4) generates all possible substitutions for literal variables in a parameterized combinator repository and computes the respective multi-arrows. The unrolled parameterized types are represented as a tuple (p, P, ρ), where p is a list of parameters, P is a set of predicates and ρ is a multi-arrow.

Algorithm 4: UNROLL: Generating and Applying Substitutions for Literals

Input: A parameterized combinator repository Γ and a literal repository Δ

Output: A mapping Γ', that maps combinators to a set of tuples (p, P, τ), where p is a list of parameters, P is a set of predicates and τ is a multi-arrow.

```
Function UNROLL(Γ, Δ):
    Γ' ← ∅
    for C : φ ∈ Γ do
        Θ ← {∅}
        Φ ← ∅
        for (x, t) ∈ params_l(φ) do
            Θ ← {θ ∪ {x : t} | θ ∈ Θ, l ∈ Δ(t)}
        for θ ∈ Θ do
            parameters ← []
            for (x, t) ∈ params(φ) do
                if x is a literal variable then
                    parameters ← parameters + [θ(x)]
                else
                    parameters ← parameters + [(x, subst(t, θ))]
            predicates ← {subst(p, θ) | p ∈ preds(φ)}
            Φ ← Φ ∪ {(parameters, predicates, marrows(subst(itype(φ)), θ))}
        Γ' ← Γ' ∪ {C : Φ}
    return Γ'
```

Example 28: Run of INHABIT for FCLP

Consider the repositories of Example 20 and the following synthesis request:

$$\{M \mid \Gamma; \Delta \vdash M : c(3)\}$$

We evaluate $\text{INHABIT}(\Gamma, \Delta, c(3), \emptyset)$ step-by-step and compute the respective parameterized tree grammar.

Algorithm 5: Inhabit: Generating a Parameterized Tree Grammar

Input: A parameterized combinator repository Γ, a literal repository Δ, an intersection type τ and a relation S of explicitly given subtypes on constructors

Output: A parameterized tree grammar G

Function Inhabit$(\Gamma, \Delta, \tau, S)$**:**

- $G \leftarrow \emptyset$
- $queue \leftarrow \{\tau\}$
- $seen \leftarrow \emptyset$
- $\Gamma' \leftarrow \textsc{Unroll}(\Gamma, \Delta)$
- **for** $\rho \in queue \setminus seen$ **do**
  - **if** $\rho = \omega$ **then**
    - **continue**
  - $seen \leftarrow seen \cup \{\rho\}$
  - **for** $C : \Phi \in \Gamma'$ **do**
    - **for** $(p, P, \rho') \in \Phi$ **do**
      - $p_0, \ldots, p_n \leftarrow p$
      - $term_params_types \leftarrow \{\upsilon \mid (x, \upsilon) \in p \mid x \text{ is a term variable}\}$
      - **for** $n\text{-}ary \in \rho'$ **do**
        - $args \leftarrow \textsc{Subqueries}(n\text{-}ary, \rho, S)$
        - **if** $|args| > 0$ **then**
          - $queue \leftarrow queue \cup \{term_params_types\}$
          - **for** $(\sigma_1, \ldots, \sigma_m) \in args$ **do**
            - $G \leftarrow G \cup \{\rho \mapsto C\, p_0\, \ldots\, p_n\, \sigma_1\, \ldots\, \sigma_m \lhd P\}$
            - $queue \leftarrow queue \cup \{\sigma_1, \ldots, \sigma_m\}$
- **return** $prune(G)$

1. We initialize $G = \emptyset$, $\mathit{queue} = \{c(3)\}$ and $\mathit{seen} = \emptyset$.

2. We start by unrolling the literal variables in the parameterized combinator repository Γ using $\textsc{Unroll}(\Gamma, \Delta)$:

   a) For combinator $C : \langle\alpha : \mathbb{N}_{\leq 3}\rangle \Rightarrow \langle\beta : \mathbb{N}_{\leq 3}\rangle \Rightarrow (\alpha = \beta - 1) \Rightarrow c(\alpha) \to c(\beta)$, we generate all possible substitutions.

      i. Starting with the literal parameter $(\alpha, \mathbb{N}_{\leq 3})$, the associated literal values are $0, 1, 2$ and 3, so initially the set of substitutions is

      $$\Theta = \{\{\alpha : 0\}, \{\alpha : 1\}, \{\alpha : 2\}, \{\alpha : 3\}\}$$

      ii. The next literal parameter $(\beta, \mathbb{N}_{\leq 3})$ has the same possible values, we build the product of the substitutions, resulting in:

      $$\begin{aligned}\Theta = \{&\{\alpha : 0, \beta : 0\}, \{\alpha : 0, \beta : 1\}, \{\alpha : 0, \beta : 2\}, \{\alpha : 0, \beta : 3\},\\ &\{\alpha : 1, \beta : 0\}, \{\alpha : 1, \beta : 1\}, \{\alpha : 1, \beta : 2\}, \{\alpha : 1, \beta : 3\},\\ &\{\alpha : 3, \beta : 0\}, \{\alpha : 3, \beta : 1\}, \{\alpha : 3, \beta : 2\}, \{\alpha : 3, \beta : 3\}\}\end{aligned}$$

      iii. Iterating over the substitutions, we generate the following tuples for Φ, and add them associated with C to Γ':

      $$\begin{aligned}\Phi = \{&([0,0], \{0 = 0 - 1\}, \{\{((), c(0) \to c(0))\}, \{((c(0)), c(0))\}\}),\\ &([0,1], \{0 = 1 - 1\}, \{\{((), c(0) \to c(1))\}, \{((c(0)), c(1))\}\}),\\ &\ldots,\\ &([3,3], \{3 = 3 - 1\}, \{\{((), c(3) \to c(3))\}, \{((c(3)), c(3))\}\})\}\end{aligned}$$

   b) The combinator $C_0 : c(0)$ has no parameters, so we add the tuple $([], \emptyset, \{\{((), c(0))\}\})$ to Γ'.

3. We iterate over Γ' starting with C and try to inhabit the type $c(3)$.

   a) We iterate over all tuples (p, P, ρ') associated with C, starting with:

   $$p = [0,0], P = \{0 = 0 - 1\}, \rho' = \{\{((), c(0) \to c(0))\}, \{((c(0)), c(0))\}\}$$

b) Since no *n-ary* in ρ' as a target, that is a subtype of $c(3)$, we do not add any rules to G. Similarly, all but the following tuples do not generate any rules.

$$([0,3], \{0 = 3 - 1\}, \{\{((), c(0) \to c(3))\}, \{((c(0)), c(3))\}\})$$
$$([1,3], \{1 = 3 - 1\}, \{\{((), c(1) \to c(3))\}, \{((c(1)), c(3))\}\})$$
$$([2,3], \{2 = 3 - 1\}, \{\{((), c(2) \to c(3))\}, \{((c(2)), c(3))\}\})$$
$$([2,3], \{3 = 3 - 1\}, \{\{((), c(3) \to c(3))\}, \{((c(3)), c(3))\}\})$$

c) Since no term parameters are present in the parameters, the only types, that are added to the queue are $c(0), c(1), c(2)$ and $c(3)$ respectively. The following rules are added to the grammar:

$$c(3) \mapsto C\,0\,3\,c(0) \lhd \{0 = 3 - 1\}$$
$$c(3) \mapsto C\,1\,3\,c(1) \lhd \{1 = 3 - 1\}$$
$$c(3) \mapsto C\,2\,3\,c(2) \lhd \{2 = 3 - 1\}$$
$$c(3) \mapsto C\,3\,3\,c(3) \lhd \{3 = 3 - 1\}$$

4. We add rules for $c(1)$ and $c(2)$ in a similar manner.

5. The rules for $c(0)$ include the following additional rule:

$$c(0) \mapsto C_0 \lhd \emptyset$$

The parameterized tree grammar for the inhabited terms is:

$$\begin{aligned}
G = \{&c(3) \mapsto C\,0\,3\,c(0) \lhd \{0 = 3 - 1\}, c(3) \mapsto C\,1\,3\,c(1) \lhd \{1 = 3 - 1\},\\
&c(3) \mapsto C\,2\,3\,c(2) \lhd \{2 = 3 - 1\}, c(3) \mapsto C\,3\,3\,c(3) \lhd \{3 = 3 - 1\},\\
&c(2) \mapsto C\,0\,2\,c(0) \lhd \{0 = 2 - 1\}, c(2) \mapsto C\,1\,2\,c(1) \lhd \{1 = 2 - 1\},\\
&c(2) \mapsto C\,2\,2\,c(2) \lhd \{2 = 2 - 1\}, c(2) \mapsto C\,3\,2\,c(3) \lhd \{3 = 2 - 1\},\\
&c(1) \mapsto C\,0\,1\,c(0) \lhd \{0 = 1 - 1\}, c(1) \mapsto C\,1\,1\,c(1) \lhd \{1 = 1 - 1\},\\
&c(1) \mapsto C\,2\,1\,c(2) \lhd \{2 = 1 - 1\}, c(1) \mapsto C\,3\,1\,c(3) \lhd \{3 = 1 - 1\},\\
&c(0) \mapsto C\,0\,0\,c(0) \lhd \{0 = 0 - 1\}, c(0) \mapsto C\,1\,0\,c(1) \lhd \{1 = 0 - 1\},\\
&c(0) \mapsto C\,2\,0\,c(2) \lhd \{2 = 0 - 1\}, c(0) \mapsto C\,3\,0\,c(3) \lhd \{3 = 0 - 1\},\\
&c(0) \mapsto C_0 \lhd \emptyset\}
\end{aligned}$$

4.3.2. Enumeration

As with regular tree grammars, we give an algorithm that enumerates the derivable terms of a parameterized tree grammar. The algorithm is similar to the one given for regular tree grammars, but with the following differences:

- We gather the potential subterms in parameter position by generating all possible substitutions for the parameters and filter them by the predicates.
- We forward literals in parameter position directly to the term level.

Since predicates depend on the values of subterms, these subterms must be generated before the predicates can be evaluated. We use the bottom-up approach of Algorithm 3, that builds subterms first and constructs larger terms from these subterms and check the predicates when constructing the larger terms.

We give an example of the enumeration of a parameterized tree grammar.

Example 29: Run of ENUMERATE for FCLP

Consider the parameterize tree grammar of Example 26 with $S = c(2)$. We enumerate the terms of the grammar using ENUMERATE(G).
We initialize the mapping $\mathit{terms} = \{c(2) \mapsto \emptyset, a \mapsto \emptyset\}$, set *have_new_terms* to *True*, and start the main loop.

1. Setting *have_new_terms* to *False*, we start the loop over the rules of the grammar.
   a) We start with $c(2) \mapsto C\,2\,(x : a) \lhd \{\mathit{size}(x) = y\}$, and consider the term parameter $(x : a)$. Substitutions are generated based on the terms in $\mathit{terms}[a]$. Since no terms are generated yet, the set of substitutions is empty.
   b) We continue with $a \mapsto X \lhd \emptyset$. Since no parameters are present, only the empty substitution is generated. All predicates trivially hold, and we iterate over the product of all $\mathit{terms}[B_i]$. Since no B_i exists, only consider the empty tuple for $(M_1, \ldots, M_n)$. The term, we consider is $C\,N_1 \ldots N_n\,M_1 \ldots M_n = X$. Since $X \notin \mathit{terms}[a]$, we add it to that set and set *have_new_terms* to true.
   c) Lastly, we continue with $a \mapsto X\,a \lhd \emptyset$. No substitutions are generated, and we iterate over the terms of the only argument a, $\mathit{terms}[a] = \{X\}$. We consider the term $X\,X$. Since $X\,X \notin \mathit{terms}[a]$, we add it to that set and set *have_new_terms* to true.

Algorithm 6: ENUMERATE: Enumerating a Parameterized Tree Grammar

Input: A parameterized tree grammar G

Output: A sequence of terms

Function ENUMERATE($G = (N, \Sigma, \mathcal{L}, S, R)$):

- $terms \leftarrow \{n \mapsto \emptyset \mid n \in N\}$
- $have_new_terms \leftarrow True$
- **while** $have_new_terms$ **do**
  - $have_new_terms \leftarrow False$
  - **for** $s \mapsto C\, p_1, \ldots\, p_m\, B_1\, \ldots\, B_n \triangleleft P \in R$ **do**
    - $\Theta \leftarrow \{\emptyset\}$
    - **for** $p_i \in \{p_1, \ldots, p_m\}$ **do**
      - **if** $p_i = (x_i : \tau_i)$ **then**
        - $\Theta \leftarrow \{\theta \cup \{(x_i : t)\} \mid \theta \in \Theta, t \in terms[\tau_i]\}$
    - **for** $\theta \in \Theta$ **do**
      - **if** *any subst*(p, θ) *for* $p \in P$ *does not hold* **then**
        - **continue**
      - $(N_i)_{i \in \{1,\ldots,m\}} \leftarrow (\theta(x_i) \text{ if } p_i = (x_i : \tau_i) \text{ else } p_i)_{i \in \{1,\ldots,m\}}$
      - **for** $(M_1, \ldots, M_n) \in \Pi_{i=1}^{n} terms[B_i]$ **do**
        - **if** $C\, N_1 \ldots N_m\, M_1\, \ldots\, M_n \notin terms[s]$ **then**
          - $terms[s] \leftarrow terms[s] \cup \{C\, N_1\, \ldots\, N_m\, M_1\, \ldots\, M_n\}$
          - $have_new_terms \leftarrow True$
          - **if** $s = S$ **then**
            - **yield** $C\, N_1\, \ldots\, N_m\, M_1\, \ldots\, M_n$

2. Since *have_new_terms* is true, we start the loop again and set *have_new_terms* to false.

   a) We consider the rule $c(2) \mapsto C\,2\,(x : a) \triangleleft \{\mathit{size}(X) = 2\}$. Since $\mathit{terms}[a] = \{X, X\,X\}$, we generate the substitutions $\Theta = \{\{(x : X)\}, \{(x : X\,X)\}\}$ and iterate over them.

      i. For the first substitution $(x : X)$, the predicate $\mathit{size}(X) = 2$ does not hold, so we continue with the next substitution.

      ii. For the second substitution $(x : X\,X)$, the predicate $\mathit{size}(X\,X) = 2$ holds. We set (N_1, N_2) to $(2\,X\,X)$ and consider only the empty tuple for $(M_1, \ldots, M_n)$, since no arguments are present. Since the term $C\,2\,(X\,X)$ is not in $\mathit{terms}[c(2)]$, we add it to that set, set *have_new_terms* to true and since $c(2) = S$, we yield this term.

3. The rules for a continue to generate new terms in the form $X(X\ \ldots (X\,X) \ldots)$, so *have_new_terms* will be set to true and the procedure runs indefinitely. There will, however, be no new terms for $c(2)$.

Example 29 shows, that it is possible for the enumeration algorithm presented in Algorithm 6 to run indefinitely, even if the set of inhabitants is finite. Obviously, the enumeration runs indefinitely, if the set of inhabitants is infinite, but even when the set of inhabitants is finite it is not possible to guarantee termination, if the set of inhabitants is finite without restricting the predicates. This is a direct result of the undecidability of inhabitation in FCLP and the use of term predicates. One way to mitigate this problem in practice is to restrict the number of terms generated for each entry in the *terms* mapping. While this does guarantee termination, it does not guarantee completeness.

When comparing the enumeration in the context of FCLP to the enumeration of FCL, the most significant difference is the addition of predicates. Without predicates, term parameters act exactly like arguments and literals can be encoded as nullary constructor symbols, that are inhabited by exactly one nullary combinator, that is the literal itself. Using a generate-and-test approach in FCL, we can emulate a similar behavior to FCLP, by firstly generating all possible terms and then filtering them by predicates externally. While this approach works in many cases, the inclusion of predicates, that are evaluated during enumeration, can lead to a significant speed-up compared to the generate-and-test approach.

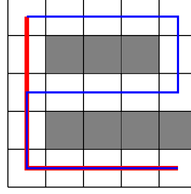

Figure 2.: 5 × 5 Maze With Exactly two Loop-Free Solutions

Example 30: Loop-Free Solutions in a Maze

Consider finding solutions for the maze depicted in Figure 2, that do not visit a position more than once. In FCL, we simply synthesize all solutions for the maze and filter them afterward. Using FCLP, we can add a term predicate to each combinator, that checks, that no subterm has visited the current position before:

$$\begin{aligned}
\Gamma = \{\textsc{Up} : {}& \langle \alpha : \mathit{pos} \rangle \Rightarrow \langle \beta : \mathit{pos} \rangle \Rightarrow (\beta = (\pi_1(\alpha), \pi_2(\alpha) - 1)) \Rightarrow \\
& \langle\!\langle \mathit{from} : \mathit{path}(\alpha) \rangle\!\rangle \Rightarrow (\beta \notin \mathit{Path}(\mathit{from})) \Rightarrow \mathit{path}(\beta), \\
\textsc{Down} : {}& \langle \alpha : \mathit{pos} \rangle \Rightarrow \langle \beta : \mathit{pos} \rangle \Rightarrow (\beta = (\pi_1(\alpha), \pi_2(\alpha) + 1)) \Rightarrow \\
& \langle\!\langle \mathit{from} : \mathit{path}(\alpha) \rangle\!\rangle \Rightarrow (\beta \notin \mathit{Path}(\mathit{from})) \Rightarrow \mathit{path}(\beta), \\
\textsc{Left} : {}& \langle \alpha : \mathit{pos} \rangle \Rightarrow \langle \beta : \mathit{pos} \rangle \Rightarrow (\beta = (\pi_1(\alpha) - 1, \pi_2(\alpha))) \Rightarrow \\
& \langle\!\langle \mathit{from} : \mathit{path}(\alpha) \rangle\!\rangle \Rightarrow (\beta \notin \mathit{Path}(\mathit{from})) \Rightarrow \mathit{path}(\beta), \\
\textsc{Right} : {}& \langle \alpha : \mathit{pos} \rangle \Rightarrow \langle \beta : \mathit{pos} \rangle \Rightarrow (\beta = (\pi_1(\alpha) + 1, \pi_2(\alpha))) \Rightarrow \\
& \langle\!\langle \mathit{from} : \mathit{path}(\alpha) \rangle\!\rangle \Rightarrow (\beta \notin \mathit{Path}(\mathit{from})) \Rightarrow \mathit{path}(\beta), \\
\textsc{Start} : {}& \mathit{pos}(0, 0)\}
\end{aligned}$$

Where *Path* is a function, that returns the set of positions visited in a path.

Using the FCLP repository of Example 30 no subterm can be generated, that visits a position twice, we also never generate terms, that visit a position three or more times. This holds even without explicitly filtering for these terms.

Since there are infinite solutions, that include loops, the enumeration for FCL never terminates. While it yields the two depicted solutions, it generates infinitely more solutions, that all are filtered out. Using FCLP, we generate exactly two solutions and then terminate, since the predicates prevent the generation of further terms, which is detectable during enumeration.

Furthermore, the maze in Figure 2 can be generalized, by making it arbitrary wide. The number of solutions are the same, but while FCLP computes the two solutions

immediately, FCL finds the first solution, and then take an arbitrary amount of time to find the second solution.

4.3.3. Optimizations

In this subsection, we give an overview over the optimizations for the inhabitation and enumeration procedures of FCLP, as they are implemented in CLSP.

While Algorithm 5 describes the general structure and idea of the procedure implemented, we give some optimizations to the procedure, that also were implemented. These optimizations consider the unrolling of the literal variables into their possible values and the generation of the substitutions. An observation is, that the number of possible substitutions for a literal variable grow exponentially with the number of literal quantifiers in the repository. While this is not avoidable, we can reduce the number of substitutions by reducing the number of literal values in a group a specific quantifiers ranges over.

Firstly, we consider the grammar, that is generated in Example 28. We see that no predicate contains variables, and most of the predicates do not hold. We can reduce the grammar eliminating all rules, that contain predicates, that do not hold, without changing the set of derivable terms. We further recognize that all predicates in the example depend solely on literal variables, that were substituted in the UNROLL step. We can avoid generating these rules in the first place by evaluating predicates, that rely solely on literal variables in the UNROLL step and only keep the substitutions, that satisfy all predicates. This reduces the amount of substituted parametrized types the main loop in INHABIT has to consider and reduces the number of rules a parameterized tree grammar has to consider, that are known not to be valid.

Secondly, we recognize, a common pattern when modeling predicates, namely that the value of a literal variable often depends solely on the values of other literal variables. This dependency can be modeled by a predicate in the form of $(\alpha = f(\beta_1, \ldots, \beta_n))$, with $\alpha, \beta_1, \ldots, \beta_n$ literal variables. We introduce a special type of predicate in the form of $(\alpha := f(\beta_1, \ldots, \beta_n))$, that signifies, that the value of α is uniquely determined by the values of $\beta_1, \ldots, \beta_n$ and can be computed through f. Since this is a predicate, that depends solely on literal variables, it also can be evaluated in the UNROLL step. Furthermore, since the value of α is uniquely determined and can be computed, we reduce the number of candidates α needs to range over to one. We consider a variant of this predicate $(\alpha \in f(\beta_1, \ldots, \beta_n))$, that signifies, that the value of α ranges over the set of values computed by f. We call this approach *generative predicates*, meaning the predicates do not act as filters on previously generated substitutions, but as generators for substitutions. Both optimizations reduce the runtime of the UNROLL step.

We further observe that substitutions for literal variables can depend on a particular subquery. Consider the repositories of Example 20 and the synthesis request of

$$\{M \mid \Gamma; \Delta \vdash M : c(3)\}$$

For each subquery $c(\gamma)$, the only substitution for β, that leads to a valid subtype is for $\beta = \gamma$. This fact can be used to reduce the number of substitutions, inferring substitutions for variables used in target positions to the literal values in a (sub-)query. This can be done using unification on literal variables. While this requires the generation of substitutions to occur repeatedly in the main loop as opposed to once in the Unroll step, depending on the number of substitutions, this can be more efficient. Notably, such a technique is impractical for type variables, that allow for intersection types and polymorphism [21], it is trivial in the case of literal variables.

Often times a repository can be structured in a way, that the number of substitutions for a literal variable can be reduced significantly, as the following example shows.

Example 31: Comparison of Different Predicates

Consider the following literal repository,

$$\Delta = \{\mathbb{N}_{\leq 3} : \{0, \ldots, 3\}\}$$

the following combinator repositories,

$$\begin{aligned}
\Gamma_0 = \{&C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (1 = \alpha - \beta) \Rightarrow c(\alpha) \to c(\beta)\\
&C_0 : c(0)\}\\
\Gamma_1 = \{&C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (\beta := \alpha - 1) \Rightarrow c(\alpha) \to c(\beta)\\
&C_0 : c(0)\}\\
\Gamma_2 = \{&C : \langle \alpha : \mathbb{N}_{\leq 3} \rangle \Rightarrow \langle \beta : \mathbb{N}_{\leq 3} \rangle \Rightarrow (\alpha := \beta + 1) \Rightarrow c(\alpha) \to c(\beta)\\
&C_0 : c(0)\}
\end{aligned}$$

and the synthesis requests

$$\begin{aligned}
&\{M \mid \Gamma_0; \Delta \vdash M : c(4)\},\\
&\{M \mid \Gamma_1; \Delta \vdash M : c(4)\} \text{ and}\\
&\{M \mid \Gamma_2; \Delta \vdash M : c(4)\}
\end{aligned}$$

Obviously, all three request in Example 31 are equivalent in the sense, that they lead to the same set of inhabitants $\{C\,4\,3(C\,3\,2\,(C\,2\,1(C\,1\,0\,C_0)))\}$ but have different runtime characteristics.

The first request enumerates all possible substitutions for α. Using unification, we infer, that for each subquery $c(\gamma)$, β is fixed to γ. Each possible value for α is enumerated, but all but one are rejected.

The second request also enumerates all 100 possible substitutions for α. While β is fixed to $\alpha - 1$, it is also fixed to γ for each subquery $c(\gamma)$. Using a predicate, that sets the value directly for variables, that occur in the target type, does not reduce the number of possible substitutions.

The last request, however, neither enumerates α nor β. The value of β is fixed by inference, and the value of α is computed by $\beta + 1$.

Further optimization can be applied when constructing the terms. In the algorithm presented, we construct the terms by building a product over all possible terms for each argument, and only consider the terms, that have not been generated yet. This can consider the same terms multiple times. We can avoid this by remembering which terms have been generated in the last iteration and only construct terms in a way, that they have at least one newly created term as a subterm.

5. Implementation

A library for synthesis via inhabitation in FCLP as described in chapter 4 has been implemented in the Python programming language[1]. The library is available at `https://github.com/tudo-seal/clsp-python` and is called Combinatory Logic Synthesizer with Predicates (CLSP) [22]. It is based on a prior implementation of synthesis in finite combinatory logic with intersection types, called *Combinatory Logic Synthesizer ((CL)S)* [6], with large parts rewritten to accommodate the new specification language of FCLP (parameterized types) and includes improvements to the core algorithm. There are several versions of (CL)S, that have been implemented in different programming languages, such as Scala, F#, Prolog and Python. The implementation at hand is based on the Python implementation. It requires Python in version 3.10 or higher with no additional libraries. The code is fully type annotated and tested using the `unittest`[2] framework. Type checking, code quality and unit testing is checked using GitHub Actions on each commit. The implementation is licensed under the Apache License 2.0[3]. While it is not available on the Python Package Index (PyPI)[4], it can be added as a dependency to a Python project by adding the git repository to the `requirements.txt` file or `pyproject.toml` of the project.

Since FCLP is a conservative extension of FCL, the implementation tries to be as compatible as reasonable possible with the existing Python implementation to act as a drop-in replacement, and thereby limit the number of changes to the codebase for existing applications.

In general, we are not directly interested in synthesizing terms, but interested in synthesizing structures in a specific domain. These can range from programs, to computer-aided design (CAD) assemblies, to simulation models for material flow, to sets of solutions for algorithmic problems, to certificates for specific solutions for a problem, to name a few. The approach CLSP takes is language agnostic in the sense, that it does not care about the specifics of the domain, but only about the structure of solutions in the domain.

[1] `https://python.org`
[2] `https://docs.python.org/3/library/unittest.html`
[3] `https://www.apache.org/licenses/LICENSE-2.0`
[4] `https://pypi.org`

Implementation and specification of combinators, as well as literals, are domain-specific and need to be provided by the user of the library.

As stated in Chapter 1, two main goals of this implementation, when compared to the prior implementation, are the following:

First, most previous implementations were not focused on the speed of the synthesis process, but on the correctness of the implementation. Therefore, they employ a model based on a verified inhabitation machine described in [3], that is not optimized for speed. In CLSP, we use a more straightforward approach, that is more efficient. Furthermore, we have seen that many specifications that can be expressed in FCL, need to be encoded using the ad-hoc polymorphism the intersection types provide. Using the finite form of polymorphism using literal variables that FCLP allows, we can employ techniques such as unification to improve the runtime of inhabitation.

Second, the usage of this library should be more user-friendly. The goal of synthesis using combinatory logic is, that a domain expert creates a set of combinators, and gives each combinator a specification. Unfortunately, the specification language of the previous implementation was required a profound understanding of the underlying theory, its user-friendliness was therefore limited. While it can be argued, that using a tool based on type theory always requires some understanding of this theory, we want to mitigate this as much as possible, by providing a more natural way of creating those specifications. For this reason, an embedded domain-specific language (eDSL) was developed, that can be read similar to traditional programming languages. Additionally, a common technique using previous implementations is to dynamically generate the repository, based on a specific problem instance. While this is still possible (due to the compatibility with the existing implementation), we want to offer a different approach, where the combinator repository is static for a given problem, and each problem instance varies only in the literal repository. In fact, a combinator repository can be seen as a logical program, with the literal repository as an input and the synthesis result as the set of solutions. The interpretation of the combinatory repository as a logical program is not new, it was stated the introduction of component-based synthesis by Rehof in [39]. The usage of the eDSL complements this view.

This chapter describes the architecture of the implementation, shows how the theoretical concepts introduced in chapter 4 are implemented, and gives examples of how to use the library.

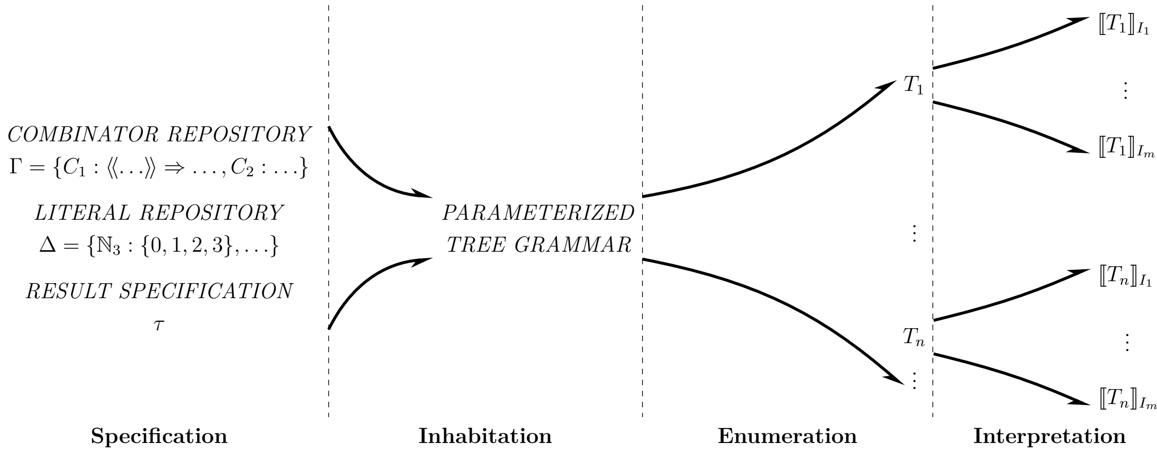

Figure 3.: Architecture of the Synthesis Process in CLSP

5.1. Architecture

The core architecture follows closely the theoretical foundations, we introduced in chapter 4. We need a combinator repository Γ, a literal repository Δ and a target intersection type τ to generate terms, that have type τ in the context of Γ and Δ. We then interpret these terms in one or more use-case dependent domain.

Synthesis using CLSP is done in the following broad steps (see Figure 3):

Specification: A domain expert provides a combinator repository Γ, a literal repository Δ and a target type τ.

Inhabitation: This step builds a parameterized tree grammar, representing all terms inhabiting τ using the combinators in Γ and literals in Δ.

Enumeration: Given the grammar of the previous step, this enumerates all (possible infinite) terms inhabiting τ.

Interpretation: This step is run on each term generated in the enumeration step. This step calls the implementation of each combinator in a term, to compute a result in the targeted domain.

The separation of the steps allows for a modular implementation, where each step can be replaced or extended with custom implementations.

While currently only one implementation of an enumeration exists, it is possible in the future to implement different enumeration strategies, that fit specific use-cases better. The current enumeration strategy of CLSP is a bottom-up approach, where terms are

generated by combining smaller terms, that inhabit subtypes of the target type, as laid out in Subsection 4.3.2. This aligns well with the implementation of predicates, that depend on subterms, and can therefore only be computed in the enumeration when the subterms are known.

The interpretation step is usually implemented by giving each combinator a specific implementation in the targeted domain. In the most basic case, each combinator is equipped with a default implementation, that is used in the interpretation step, but the system allows for switching the implementation of combinators and even having multiple implementations for the combinators in the repository.

A typical invocation of the library is shown in Listing 1. The inhabitation step creates an object of class `FiniteCombinatoryLogic`. This is a slight misnomer, as the class is not limited to finite combinatory logic, but handles the finite combinatory logic with predicates. The name is kept for compatibility reasons with the existing implementation, so that this new implementation can be used as a drop-in replacement.

5.2. Data Types

In this section, we describe the data types, that are used in the implementation of CLSP and how they relate to the theoretical concepts introduced in chapter 4. We go into some implementation details, as this allows for a more in-depth understanding of the library and how to use it, but we refrain from going into detail, when the specifics of the implementation is not meant to be used by the user of the library directly.

5.2.1. Combinators

Combinators can be any `Hashable`[5] Python object. There are two approaches for implementing the semantic of a combinator, either a combinator is itself a `Callable`[6] object (e.g. a function) or a combinator is a placeholder, that gets replaced by an implementation in the interpretation step (e.g. a string). Both approaches have advantages and disadvantages. The latter approach is more flexible, as it allows a combinatory term to be interpreted in different ways, depending on the context, while the former approach is more simple and streamlined when the use-case only calls for one single interpretation.

[5] `https://docs.python.org/3/library/collections.abc.html#collections.abc.Hashable`
[6] `https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable`

```python
from clsp import FiniteCombinatoryLogic, enumerate_terms, interpret_term

def main() -> None:
  # Define the synthesis request
  gamma = ...
  delta = ...
  target_type = ...
  # Assuming we have two interpretations
  interpretation1 = ...
  interpretation2 = ...

  # Inhabitation
  result = FiniteCombinatoryLogic(repository=gamma, literals=delta)
              .inhabit(target_type)

  interpreted_terms0 = []
  interpreted_terms1 = []
  interpreted_terms2 = []
  # Enumerate the first 100 terms
  for term in enumerate_terms(result, target_type, max_count=100):
    # Interpret the term in the default and both additional
    # interpretations and save the results
    interpreted_terms0.append(interpret_term(term))
    interpreted_terms1.append(interpret_term(term, interpretation1))
    interpreted_terms2.append(interpret_term(term, interpretation2))
```

Listing 1: Example Usage of the CLSP Library

5.2.2. Intersection Types

Intersection types are represented as custom Python classes. Earlier versions of (CL)S – in particular the version, that is implemented in Scala – use the built-in type system of the implementation language as part of the typing system. That version uses a notion of *native* and *semantic* types, where native types are the built-in types of the language and semantic types are built using custom data types. Combinators are typed using a native type, that corresponds to the implementation of the combinator, intersected with a semantic type, thereby assuring, that resulting terms were well-typed in relation to the implementation language. The implementation described in this thesis does not make any use of this distinction for several reasons:

1. While the type system of Python described in PEP 484[7] supports advanced typings, such as union types, it does not support intersection types. Therefore, we would need to encapsulate the native types of Python in the custom type system.

2. Since the domain of the synthesis is not necessarily the implementation language, but can be arbitrary, the type system of the implementation language does not add substantial certainty to the correctness of the implementation.

3. We allow the interpretation of combinators to be manyfold, a native type for one interpretation might not necessarily be fitting for another interpretation.

In addition to the representation as Python classes, operators are defined for each type constructor, to allow for a more concise representation of types. Table 1 shows the type constructors and their Python representation, as well as the operator used to represent the type constructor. We give some notes on the choice of the operators:

- The operator `**` is used, as it is the only Python operator, that is left-associative and Python does not allow changing the associativity of operators.

- The second argument of the `Literal` class is the group t of the literal $l : t$. While this is not necessary for the type system, since literal groups are considered disjoint, this is needed in the implementation, where no such requirement is present.

- The second argument of `Constructor` is optional. If none is given, the argument of ω is assumed.

[7] `https://www.python.org/dev/peps/pep-0484/`

Type Constructor	Python Class	Operator
$\sigma \to \tau$	`Arrow(`σ`, `τ`)`	σ `**` τ
$\sigma \cap \tau$	`Intersection(`σ`, `τ`)`	σ `&` τ
$c(\tau)$	`Constructor('c', `τ`)`	`c@`τ
ω	`Omega()`	–
l	`Literal('l', 't')`	–
α	`LVar('`α`')`	–

Table 1.: Type Constructors and Their Python Representation

Each type constructor inherits from an abstract base class[8] called `Type`. When a type object is created, the following two properties are computed at creation time: `is_omega`, `organized`, `size` and `free_vars`, that contain the information if a type is equal to ω (According to the rules of Definition 15), a copy of the term, that is organized (see Definition 21), its size and a set of literal variables, that are contained in the type. Type objects are immutable. This provides the benefit, that they can be used as keys in dictionaries and elements in sets, which is necessary for caching strategies employed in the inhabitation algorithm. Unfortunately, this also means, that substitution of type variables always creates a new object, which has significant runtime implications. For this reason, substitution in the implemented procedure is delayed as long as possible, and only applied once at the end.

Example 32: Intersection Types in CLSP

We give some examples of intersection types and their representation as Python objects, as well as their operator notation.

- The type $a \to b \to c$ is represented as
  `Arrow(Constructor('a'), Arrow(Constructor('b'),`
  `Constructor('c')))`
  and can be written as
  `Constructor('a') ** Constructor('b') ** Constructor('c')`.
- The type $a \cap b \cap c$ is represented as
  `Intersection(Intersection(Constructor('a'), Constructor('b')),`
  `Constructor('c'))`

[8] https://docs.python.org/3/library/abc.html

and can be written as
`Constructor('a') & Constructor('b') & Constructor('c')`.

- The type $a \to b \cap c$ is represented as
  `Intersection(Arrow(Constructor('a'), Constructor('b')), Constructor('c'))`
  and can be written as
  `Constructor('a') ** Constructor('b') & Constructor('c')`.

- The Term $a(b \cap c) \to \omega$ is represented as
  `Arrow(Constructor('a', Intersection(Constructor('b'), Constructor('c'))), Omega())`
  and can be written as
  `('a'@(Constructor('b') & Constructor('c'))) ** Omega()`.

5.2.3. Parameterized Terms and Predicates

Both literal ($\langle\alpha : t\rangle \Rightarrow \varphi$) and term quantifiers ($\langle\!\langle x : \tau\rangle\!\rangle \Rightarrow \varphi$) are implemented by the same Python class `Param`, that is applied to either another `Param` object representing φ or a `Type` object if φ is an intersection type. The distinction between literal and term quantifiers is made automatically whether it has been given a group identifier t or an intersection type τ. Types are represented as Python objects, as mentioned in the section above, while groups identifiers are represented as strings. Since types and strings are disjoint, there exists no overlap and the choice is deterministic.

Predicates decide whether a given set of substitutions is valid or invalid. They are implemented as Python functions, that take a context as an argument and return a Boolean value. The context is a dictionary mapping variable names to their values, that represents the substitutions of literal and term variables, that were introduced in the type before the predicate. In contrast to the theoretical definition of predicates, we do not require the predicates to be decidable, since this cannot be enforced in the implementation. Predicates, as implemented in CLSP, are bound to a quantifier, thereby separating them into two groups: predicates, that are bound to literal variables and predicates, that bound to term variables. We allow predicates, that are bound to literal variables to depend solely on other literal variables, and predicates, that are bound to term variables to depend on both literal and term variables. One notable consequence of this design is, that we can distinguish between predicated, that depend only on literal variables and predicates, that depend on both literal and term variables. This restriction

does not affect the expressiveness of the system, as we can always introduce the term variables last, and combine all predicates in a single predicate using a conjunction and add it to that term variable, but it allows the system to handle pure;y literal predicates differently from term predicates for optimization purposes.

There is a special treatment for predicates of quantifiers, that are in the form of $\langle\alpha : t\rangle \Rightarrow (\alpha := f(\beta_1, \ldots, \beta_n)) \Rightarrow \varphi$ or $\langle\alpha : t\rangle \Rightarrow (\alpha \in f(\beta_1, \ldots, \beta_n)) \Rightarrow \varphi$, where $\beta_1, \ldots, \beta_n$ are literal variables, f is either a function, that returns a literal of group t or a set of literals of group t. These special forms are capsuled in an object of class `SetTo`, and compute the result of $f(\beta_1, \ldots, \beta_n)$ to set the value of α directly or let α range over the result. There are two Boolean options to the `SetTo` class, `multi_value` and `override`. If `multi_value` is set, then f should return an `Iterable`[9] and the predicate is treated as the $\in$-case, otherwise the return value of f is taken as the only possible value of α. If `override` is set, the requirement, that f should return a member of the group t is relaxed; any result is bound to α, otherwise the substitution is discarded, if f does not return a value of the group t. The default value for both options is `False`.

It is possible for a parameter to have no predicate associated with it, in this case a function, that always returns `True` should be used (`lambda ctx: True`). The class `Param` is implemented as a `dataclass`[10], that takes a name, a group identifier or a type, one or more predicates and an inner parameter as an input, as shown in Listing 2.

Since creating parameterized types in such a manner is cumbersome, an embedded domain-specific language (eDSL) is provided, that allows for a more concise definition of parameterized types and better mirrors the theoretical notation. The eDSL works by chaining methods, that add quantifiers and predicates to a `DSL` object. The eDSL is constructed using the following methods and constructors:

- `DSL(cache = False: bool, infer = True: bool)`: This initializes the eDSL, and gathers parameters, predicates and finally an intersection type to build a parameterized type. All further components are methods, that called in this object, manipulate this object and return the object itself, to allow for chaining. The meaning of the `cache` and `infer` parameters is explained in Section 5.3.

- `Use(name: str, group: str | Type)`: This adds a literal or term quantifier. As with the `Param` class, we do not explicitly differentiate between these. The differentiation is made implicitly by the `group` argument.

[9] https://docs.python.org/3/glossary.html#term-iterable
[10] https://docs.python.org/3/library/dataclasses.html

```
@dataclass
class Param:
    name: str
    group: Type | str
    predicate: (
        Sequence[Callable[[MutableMapping[str, Any]], bool] | SetTo]
        | Callable[[dict[str, Any]], bool]
        | SetTo
    )
    inner: Param | Type

@dataclass
class SetTo:
    compute: Callable[[dict[str, Any]], Any]
    override: bool = field(default=False)
    multi_value: bool = field(default=False)
```

Listing 2: Definition of the Classes `Param` and `SetTo`

- `With(predicate: Callable[..., bool], raw: bool = False)`: This adds a predicate to the last added quantifier. Predicates added this way differ slightly from predicates added via the `Param` class. A predicate added via the `With` method is not passed a dictionary with substitutions by default, but rather the values of literal variables themselves. For this, the names of the arguments of the function `predicate` are analyzed the predicate is called with the values of the literal variables, that match the names of the arguments. Using the `raw` parameter, the behavior can be changed to match the behavior of the `Param` class. This is useful, when the types are built dynamically and the names of the arguments are not known beforehand.

- `As(set_to: Callable[..., Any], raw: bool = False, multi_value: bool = False, override: bool = False)`: This takes a function `set_to` and wraps it in a `SetTo` object to set the value of the last added quantifier. Concerning its `raw` argument, it has the same behavior as `With`. Its `multi_value` and `override` arguments are forwarded to the `SetTo` object.

- `In(inner: Type)`: This capsules the inner intersection type of the parameterized type. In contrast to the other methods, this does not return the `DSL` object, but constructs the parameterized type and returns it.

If a quantifier is associated with multiple predicates, the predicates are combined, meaning that a substitution must satisfy all predicates to be valid. This hold true both for predicates introduced by the `With` and the `As` method.

Example 33: Parameterized Types as Param Objects and eDSL

We give some examples how parameterized types are constructed using the `Param` class and the eDSL.

- The type $\langle\alpha : \mathbb{N}_3\rangle \Rightarrow \langle\beta : \mathbb{N}_3\rangle \Rightarrow (\alpha \geq \beta) \Rightarrow c(\alpha) \to c(\beta)$ is represented as

```
Param('alpha', 'N3', lambda ctx: True,
  Param('beta', 'N3', lambda ctx: ctx['alpha'] >= ctx['beta'],
    ('c' @ LVar('alpha')) ** ('c' @ LVar('beta'))))
```

  and can be written as

```
DSL()
  .Use('alpha', 'N3')
  .Use('beta', 'N3')
  .With(lambda alpha, beta: alpha >= beta)
  .In(('c' @ LVar('alpha')) ** ('c' @ LVar('beta')))
```

- The type $\langle\alpha : \mathbb{N}_3\rangle \Rightarrow \langle\beta : \mathbb{N}_3\rangle \Rightarrow (\beta = \alpha + 1) \Rightarrow c$ is represented as

```
Param('alpha', 'N3', lambda ctx: True,
  Param('beta', 'N3', SetTo(lambda ctx: ctx['alpha'] + 1),
    Constructor('c')))
```

  and can be written as

```
DSL()
  .Use('alpha', 'N3')
  .Use('beta', 'N3')
  .As(lambda alpha: alpha + 1)
  .In(Constructor('c'))
```

  While the predicate $(\beta = \alpha + 1)$ could be implemented as a `With` predicate, it is both more concise and more efficient to use the `As` method, as this gets special treatment in the inhabitation algorithm.

- The type $\langle \alpha : \mathbb{N}_3 \rangle \Rightarrow \langle \beta : \mathbb{N}_3 \rangle \Rightarrow \langle \gamma : \mathbb{N}_3 \rangle \Rightarrow (\gamma \in \{0, \beta\}) \Rightarrow c$ is represented as

```
Param('alpha', 'N3', lambda ctx: True,
  Param('beta', 'N3', lambda ctx: True,
    Param('gamma', 'N3',
      SetTo(lambda ctx: {0, ctx['alpha'], ctx['beta']},
        multi_value=True),
      Constructor('c'))))
```

  and can be written as

```
DSL()
  .Use('alpha', 'N3')
  .Use('beta', 'N3')
  .Use('gamma', 'N3')
  .As(lambda alpha, beta: {0, alpha, beta}, multi_value=True)
  .In(Constructor('c'))
```

- The type $\langle \alpha : \mathbb{N}_3 \rangle \Rightarrow \langle\langle x : c(\alpha) \rangle\rangle \Rightarrow \langle\langle y : c(\alpha) \rangle\rangle \Rightarrow (x = y) \Rightarrow c$ is represented as

```
Param('alpha', 'N3', lambda ctx: True,
  Param('x', 'c' @ LVar('alpha'), lambda ctx: True,
    Param('y', 'c' @ LVar('alpha'),
      lambda ctx: ctx['x'] == ctx['y'],
      Constructor('c'))))
```

  and can be written as

```
DSL()
  .Use('alpha', 'N3')
  .Use('x', 'c' @ LVar('alpha'))
  .Use('y', 'c' @ LVar('alpha'))
  .With(lambda x, y: x == y)
  .In(Constructor('c'))
```

- The type $\langle \alpha : \mathbb{N}_3 \rangle \Rightarrow \langle\langle x : b \rangle\rangle \Rightarrow (\mathit{size}(x) = \alpha) \Rightarrow c$ is represented as

```
Param('alpha', 'N3', lambda ctx: True,
  Param('x', Constructor('b'),
```

```
        lambda ctx: size(ctx['x']) == ctx['alpha'],
        Constructor('c')))
```

and can be written as

```
DSL()
  .Use('alpha', 'N3')
  .Use('x', Constructor('b'))
  .With(lambda alpha, x: x.size == alpha)
  .In(Constructor('c'))
```

Regarding the last two examples, it is not possible to use the `As` method or a `SetTo` predicate, as those are only possible for predicates associated with literal variables. Since the predicate depends on the value of a term variable, it cannot be associated with a literal variable.

Example 33 shows, that the eDSL is more concise and easier to read than the direct construction of the `Param` class. Additionally, the eDSL mirrors the theoretical notation more closely. In the remainder of this thesis, we use the eDSL to define parameterized types, unless we want to explicitly show internal details regarding the `Param` class. Users of the library are encouraged to use the eDSL, as it is more readable and user-friendly.

5.2.4. Repositories

A combinator repository is a dictionary that maps combinators to their parameterized types or intersection types. In contrast to the theoretical definition, we allow for a more flexible way to represent literal repositories. In the most basic form, a literal repository is a dictionary, that maps group identifiers to a finite collection of literals, e.g. a `list` or a `set`, but we allow the groups of literals to be represented by an arbitrary Python object, that either implements the `Iterable` abstract base class or the `Contains` Protocol. While the `Iterable` class is a python built-in class, that requires the implementation of the `__iter__` method, the `Contains` protocol is an addition of CLSP and requires the implementation of the `__contains__` method. The `__iter__` method is used to generate all substitutions for a variable, while the `__contains__` method is used internally to check, if a literal is a member of the group, when a literal is computed by a `SetTo` predicate or via inference. Separating the membership test from the items returned by the iterator allows for groups, that behave as if they were finite, when iterating over, but can check membership for infinite sets. This is especially useful for groups, that are only accessed via a `SetTo` predicate or are always inferred, as the actual content of the group never

needs to be enumerated. When a group, that only implements the `Contains` protocol, is iterated over, a `RuntimeError` is thrown, detailing the error. We call groups of literals, that only implement the `Contains` protocol *virtual groups*.

We give the repositories of Examples 20 to 24 as repositories in terms of CLSP. Since the combinators of these examples do not carry any semantic meaning, we use strings as combinators.

Example 34: CLSP Repository With Literal Quantifier and Predicate

Consider the repositories of Example 20. We can represent them as

```
delta = {
  'N3': {0, 1, 2, 3}
}
gamma = {
  "C": DSL()
    .Use('alpha', 'N3')
    .Use('beta', 'N3')
    .As(lambda alpha: alpha + 1)
    .In((c @ LVar('alpha')) ** (c @ LVar('beta'))),
  "C0": c @ Literal(0, 'N3')
}
```

Example 35: CLSP Repository With Term Quantifier and Predicate

Consider the repository of Example 21. We can represent it as

```
gamma = {
  "C": DSL()
    .Use('x', Constructor('a'))
    .With(lambda x: x.size == 3)
    .In(Constructor('b')),
  "X": Constructor('a') & Constructor('a') ** Constructor('a')
}
```

Example 36: CLSP Repository With Literal and Term Quantifier and Predicate

Consider the repositories of Example 22. We can represent them as

```
delta = {
  'N3': {0, 1, 2, 3}
}
gamma = {
  "C": DSL()
    .Use('alpha', 'N3')
    .Use('x', Constructor('a'))
    .With(lambda alpha, x: x.size == alpha)
    .In(c @ LVar('alpha')),
  "X": Constructor('a') & Constructor('a') ** Constructor('a')
}
```

Example 37: CLSP Repository With Equality Constraints

Consider the repository of Example 23. We can represent it as

```
gamma = {
  "C": DSL()
    .Use('x', Constructor('a'))
    .Use('y', Constructor('a'))
    .With(lambda x, y: x == y)
    .In(Constructor('b')),
  "X": Constructor('a') & Constructor('a') ** Constructor('a')
}
```

Example 38: CLSP Repository for Maze Solving

Consider the repositories of Example 24. We can represent them as

```
delta = {
  'pos': set(product(range(5), range(5)))
    .difference({(1, 1),(2, 1),(3, 1),(1, 3),(2, 3),(3, 3)})
}
gamma = {
  "Up": DSL().Use('a', 'pos').Use('b', 'pos')
    .With(lambda a, b: b == (a[0], a[1] - 1))
    .In(('pos' @ LVar('a')) ** ('pos' @ LVar('b'))),
  "Down": DSL().Use('a', 'pos').Use('b', 'pos')
    .With(lambda a, b: b == (a[0], a[1] + 1))
```

```
    .In(('pos' @ LVar('a')) ** ('pos' @ LVar('b'))),
  "Left": DSL().Use('a', 'pos').Use('b', 'pos')
    .With(lambda a, b: b == (a[0] - 1, a[1]))
    .In(('pos' @ LVar('a')) ** ('pos' @ LVar('b'))),
  "Right": DSL().Use('a', 'pos').Use('b', 'pos')
    .With(lambda a, b: b == (a[0] + 1, a[1]))
    .In(('pos' @ LVar('a')) ** ('pos' @ LVar('b'))),
  "Start": 'pos' @ Literal((0, 0), 'pos')
}
```

We also give an example of a repository, that uses a virtual group.

Example 39: CLSP Repository Using a Virtual Group

The following repository is a variant of the repository of Example 34, where the group of literals is not a finite set, but the infinite set of natural numbers, implemented as a virtual group.

```
class NaturalNumbers(Contains):
    def __contains__(self, item: int) -> bool:
        return isinstance(item, int) and item >= 0

delta = {
  'N': NaturalNumbers()
}
gamma = {
  "C": DSL()
    .Use('beta', 'N')
    .Use('alpha', 'N')
    .As(lambda beta: beta - 1)
    .In((c @ LVar('alpha')) ** (c @ LVar('beta')))
}
```

When comparing the implementation of Example 39 to the implementation of Example 34, we see that the order of the variables `alpha` and `beta` is reversed, so that the value of variable `alpha` can be computed from the value of variable `beta`. Since `beta` occurs in the target of the intersection type, its value is inferred during the inhabitation process. Therefore, the group `N` does not need to be enumerated over, and its implementation

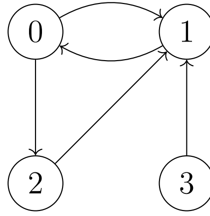

Figure 4.: A Graph for Example 40

of the `__iter__` can safely be admitted. If we implement `N` as a `set` or a `list`, the implementation would have to enumerate over all natural numbers, which would not terminate.

This technique is not only useful when modeling repositories with infinite groups, but also when modeling repositories with very large groups. A common use-case for literals is to act as state during an algorithm, that is implemented in the combinator repository. Since the set of all possible states can be very large, it is not feasible to enumerate over all possible states in each step of the algorithm. The following example shows a repository, that computes all reachable regions of a graph, starting from a given position. It uses a set of positions to represent a reachable region and to avoid visiting the same position twice. Therefore makes use of the poserset of all positions as the set of all possible states.

Example 40: CLSP Repository Modeling a Power Set

Consider the graph in Figure 4. We want to gather all positions, that are reachable from position 0. We use the graph as a singleton literal group and use the power set of all positions as the state space for possible reachable regions.

```
graph = { 0: {1, 2},  1: {0},  2: {1},  3: {1} }

def powerset(s: set[int]) -> list[tuple[int,...]]:
  return list(chain.from_iterable(combinations(s, r)
            for r in range(len(s)+1)))

class Powerset:
  def __init__(self, s: set[int]):
    self.s = s
  def __iter__(self) -> Iterator[tuple[int,...]]:
```

```
        return powerset(self.s)
    def __contains__(self, item: set[int]) -> bool:
        return item.issubset(self.s)

delta = {
    'power_pos': Powerset({p for p in graph}),
    'graph': [graph]
}

gamma = {
    "Step" : DSL()
        .Use('graph', 'graph')
        .Use('old_reach', 'power_pos')
        .Use('new_reach', 'power_pos')
        .As(lambda graph, old_reach:
            old_reach | {n for p in old_reach for n in graph[p]})
        .With(lambda old_reach, new_reach: old_reach != new_reach)
        .In(('state' @ LVar('new_reach') ** 'state' @ LVar('old_reach')))
    "End": DSL()
        .Use('reach', 'power_pos')
        .In('state' @ LVar('reach'))
}

grammar = FiniteCombinatoryLogic(gamma, delta)
    .inhabit('state' @ Literal({0}, 'power_pos'))
```

Terms synthesized by Example 40 do not only contain the maximal reachable region, but also all intermediate reachable regions.

Would Example 40 have used a `set` or a `list` to represent the group `power_pos`, it would grow exponentially with the number of positions in the graph, which makes this approach infeasible for graphs with more than a few positions. In the example presented, each literal variable only ranges over exactly one element of the group, which makes the approach feasible even for large graphs. We want to highlight the usage of the singleton literal group `graph` in the literal repository. This is a common pattern, for data flow from the literal repository to the combinator repository, for data, that does not need to be iterated over. Since the group `'graph'` only contains one element, adding a quantifier for

that group adds a negligible amount of runtime to the inhabitation process, but allows the predicates to consider specific data from the literal repository, helping the separation of data and code.

Lastly, this approach allows us to use literals groups, that are not only infinite, but inherently not enumerable, such as the set of real numbers.

5.2.5. Terms

Terms, that are generated by the synthesis, are represented as a recursive tree-like data structure, where each node is a combinator or a literal. These are returned by the enumeration step and occur as part of substitutions in predicates for term variables. They are implemented as a `dataclass Tree` with the following main attributes: `root`, `children`, `variable_names`, that correspond to the outermost combinator (or literal), its subterms, and the names of variables introduced in the parameterized type, that is associated with the outermost combinator in the combinator repository. Additionally, the size of the term is computed at creation time and stored in the `size` attribute. While the reason for the `root` and `children` attributes is obvious, the `variable_names` attribute is necessary to compute the following inferred properties, `parameters` and `arguments`. The `parameters` property is a dictionary, that maps the variable names to the corresponding subterms, that are bound to the variable, the `argument` property is a tuple over the subterms, that were synthesized for the argument types of the intersection type. While access to specific subterms is possible by traversing the tree structure using the `children` attribute and indexing using positions, the inclusion of the latter two properties allow for a more concise way of accessing those by name, which is especially useful when building term predicates.

Example 41: CLSP Terms

We give an example of the term used in Example 22 to inhabit the type $c(2)$. See the python repositories given in Example 36 for the implementation of the parameterized types.

```
Tree(root="C",
  children=(
    Tree(root=2, children=(), variable_names=[], size=1,
      parameters={}, arguments=()),
    Tree(root="X",
      children=(Tree(root="X", children=(), variable_names=[],
```

```
        size=1, parameters={}, arguments=()),),
      variable_names=[], size=2, parameters={},
      arguments=(Tree(root="X", children=(), variable_names=[],
        size=1, parameters={}, arguments=()),),
    ),
  ),
  variable_names=['alpha', 'x'],
  size=3,
  parameters={
    'alpha': Tree(root=2, children=(), variable_names=[], size=1,
      parameters={}, arguments=()),
    'x': Tree(root="X",
      children=(Tree(root="X", children=(), variable_names=[],
        size=1, parameters={}, arguments=()),),
      variable_names=[], size=2, parameters={},
      arguments=(Tree(root="X", children=(), variable_names=[],
        size=1, parameters={}, arguments=()),),
    ),
  },
  arguments=(),
)
```

It might seem redundant and memory inefficient to have multiple references to the same subterm in the tree structure (once as part of the `children` attribute and once as part of the `parameters` and `arguments` attributes), but the `parameters` and `argument` attributes are computed on demand and not stored in memory. Only the `size` attribute is computed statically beforehand, for optimization purposes.

5.2.6. Parameterized Tree Grammar

A parameterized tree grammar is returned by the inhabitation step. Since it is not designed to be interacted with by an end-user, we do not go into the implementation details. A parameterized tree grammar capsules a dictionary, mapping intersection types as non-terminals to a list of right-hand-rules, that contain a reference to a combinator as terminal symbols, the list of parameters of the combinator, that are either a reference to a term variable or a literal itself a list of term predicates and a list of intersection types representing the applied arguments to the combinator as non-terminals.

5.3. Inhabitation

The inhabitation procedure takes a combinator repository as a dictionary mapping combinators to parameterized types, a literal repository mapping group identifiers to sets of values, an explicit subtype relation as dictionary from constructor name to sets of constructor names, and an intersection type to inhabit as input. It is called by firstly creating an instance of the class `FiniteCombinatoryLogic` and then calling its `inhabit` method:

```
grammar = FiniteCombinatoryLogic(repository=gamma, subtypes=subtypes,
  literals=delta).inhabit(target)
```

The `inhabit` method implements Algorithm 5 with some minor technical differences. Creating a `FiniteCombinatoryLogic` object initializes the inhabitation algorithm and precomputes the multi-arrows of the intersection type of the parameterized types in the combinator repository. It does not yet precompute the literal substitutions, since this is done on demand during the inhabitation process to facilitate the inference of the literal substitutions. The `inhabit` method implements the remainder of Algorithm 5.

CLSP makes use of the optimizations described in Subsection 4.3.3. Instead of iterating over each literal group to compute literal substitutions, we only iterate over groups for variables, that are not computed by a `SetTo` predicate, since these are limited to the result of their corresponding predicate. Generally, literal substitutions are generated on-demand when iterating over a specific $C : \Phi \in \Gamma$ relative to a specific ρ, that needs to be inhabited. Each literal predicate is evaluated as soon as possible to discard invalid substitutions. There are three possible modes implemented for enumerating the literal substitution.

- The first mode is to compute all literal substitutions each time on demand, and using the inferred literal substitutions the same way, as the `SetTo` predicates. While this computes the literal substitutions for each combinator in each step, it is most efficient, if the amount of unbound literal variables (either through a `SetTo` predicate or as an inferred substitution) is small.

- The second mode is to compute all possible literal substitutions, honoring the `SetTo` predicates, but not the inferred literal substitutions, and cache these substitutions, to only compute them once. Then compute the inferred literal substitutions and use them to filter on the cached substitutions. This way, the substitution space is only computed once, but the algorithm still respects the inferred literal substitutions. This is the preferred mode, if the amount of unbound literal variables is large.

- The third mode is to not compute the inferred literal substitutions at all, and only use the `SetTo` predicates to filter the literal substitutions. This mode is the least effective, and should only be used for debugging purposes.

The mode can be set for each parameterized type by setting the `cache` and `infer` argument of the `DSL()` object. If the `infer` argument is false, the third mode is always used. If the `cache` argument is true, the second mode is always used. If the `cache` argument is false and the `infer` argument is true, the first mode is used, this is the default behavior.

The remainder of the implementation follows the algorithm closely. For the decision procedure of the subtyping relation, the algorithm described in [4] was used.

5.4. Enumeration

The enumeration procedure takes a parameterized tree grammar and a target type as input and returns an `Iterable` over terms that inhabit the target type. While the design of CLSP in theory allows for multiple enumeration strategies, only one strategy is currently implemented. The main idea of the bottom-up approach is shown in Algorithm 6, but the actual implementation is more complex, and utilizes many optimizations, as described in Subsection 4.3.3. We do not go into the details of each optimization, since they are mostly technical and do not change the overall structure of the algorithm. The enumeration is implemented as a generator function `enumerate_terms`, and is called as follows with a parameterize tree grammar `grammar` and target type `tau`:

```
terms = enumerate_terms(grammar, tau, max_count=n, max_bucket_size=m)
```

While the function generates an infinite number of terms, that can be limited by the Python `itertools.islice`[11] function, the `max_count` argument limits the number of terms generated, and makes the algorithm aware of this limit, so the even number of terms generated in each step is limited by that maximum. Similarly, the `max_bucket_size` argument limits the number of terms, that are saved in the *terms* dictionary of Algorithm 6 for each non-terminal n. Both arguments are optional and default to `None`, which means no limit is set, but setting limits can be yield performance improvements and better termination properties, while sacrificing completeness. In the enumeration implemented, the sequence of terms is not guaranteed to be sorted by size, but the terms are generated in a way, that each subterms is generated and outputted before the term itself, which is a property of the bottom-up approach.

[11] `https://docs.python.org/3/library/itertools.html#itertools.islice`

5.5. Interpretation

Interpretation of terms is implemented as a function, that takes a term and optionally a dictionary of interpretations for combinators and returns the interpreted term. Given a term `term` and a dictionary `interpretation`, the interpretation of the term is computed by the following function call:

```
interpreted_term = interpret_term(term, interpretation)
```

Interpretation is done bottom-up, starting with the leaves of the term and interpreting each subterm before interpreting the term itself. When a combinator is interpreted, its corresponding interpretation is called, either from the dictionary or the default interpretation of the combinator. If a combinator is not found in the dictionary, the default interpretation is used. Before calling an interpretation, it is checked, if the arity of an interpretation matches the number of arguments of the combinator. If the number of arguments is lower than the arity of the interpretation, a `partial`[12] function is created. If the number of arguments is higher than the arity of the interpretation, a `RuntimeError` is thrown. The arity of an interpretation is found using reflection on the function object using the `inspect`[13] module of Python.

Intersection types allow us to model combinators of varying arity, but Python functions do not allow for multiple specific arities. To circumvent this, we allow for variadic arguments in Python. An interpretation that is implemented using variadic arguments can be called with any number of arguments, including the arguments specified by the intersection type. Literals included in the term are interpreted as is, and are not passed to the interpretation function. Special treatment is given to interpretations, that do not implement the `Callable` protocol. These are handled as constant values and are as-is.

5.5.1. Signatures and Interpretations

While not implemented as part of CLSP, the concept of generic abstract base classes and inheritance in Python aligns well with the concept of signatures and algebras as defined in Section 2.1 and is given as a best practice to define interpretations for CLSP. A signature can be implemented as an abstract base class, that is generic over its sorts and implements each function symbol as an abstract method. An algebra can then be implemented as a subclass of the signature, setting the sorts to the desired types and implementing the abstract methods. Lastly, in the abstract signature class we can

[12] `https://docs.python.org/3/library/functools.html#functools.partial`
[13] `https://docs.python.org/3/library/inspect.html`

implement a method, that generates a dictionary, as requested by the `interpret_term` function, that maps each combinator to the corresponding method in the algebra.

Example 42: Signatures and Interpretations for Solutions for Mazes in Python

We give an example of a signature and interpretation for the combinators of Example 38.

```
Path = TypeVar('Path')

class MazeSignature(Generic[Path], ABC):
  @abstractmethod
  def Up(self, a: tuple[int, int], b: tuple[int, int]
           , in_path : Path) -> Path: ...

  @abstractmethod
  def Down(self, a: tuple[int, int], b: tuple[int, int]
             , in_path : Path) -> Path: ...

  @abstractmethod
  def Left(self, a: tuple[int, int], b: tuple[int, int]
             , in_path : Path) -> Path: ...

  @abstractmethod
  def Right(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : Path) -> Path: ...

  @abstractmethod
  def Start(self) -> Path: ...

  def as_dict(self) -> dict[str, Callable[..., Path]]:
    return {
      "Up": self.Up,
      "Down": self.Down,
      "Left": self.Left,
      "Right": self.Right,
      "Start": self.Start
```

```
    }

class PathAlgebra(MazeSignature[list[tuple[int, int]]]):
  def Up(self, a: tuple[int, int], b: tuple[int, int]
            , in_path : list[tuple[int, int]]
            ) -> list[tuple[int, int]]:
    return in_path + [b]

  def Down(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : list[tuple[int, int]]
              ) -> list[tuple[int, int]]:
    return in_path + [b]

  def Left(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : list[tuple[int, int]]
              ) -> list[tuple[int, int]]:
    return in_path + [b]

  def Right(self, a: tuple[int, int], b: tuple[int, int]
               , in_path : list[tuple[int, int]]
               ) -> list[tuple[int, int]]:
    return in_path + [b]

  def Start(self) -> list[tuple[int, int]]:
    return [(0, 0)]
```

To illustrate the usage of a second interpretation, we give an example interpretation, that counts the number of steps taken in the maze.

```
class CounterAlgebra(MazeSignature[int]):
  def Up(self, a: tuple[int, int], b: tuple[int, int]
            , in_path : int) -> int:
    return in_path + 1

  def Down(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : int) -> int:
    return in_path + 1
```

```
  def Left(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : int) -> int:
    return in_path + 1

  def Right(self, a: tuple[int, int], b: tuple[int, int]
              , in_path : int) -> int:
    return in_path + 1

  def Start(self) -> int:
    return 0
```

An inhabited term `t` is then interpreted using the `interpret_term` function as follows:

```
path = interpret_term(t, PathAlgebra().as_dict())
length = interpret_term(t, CounterAlgebra().as_dict())
```

As mentioned in Chapter 3, the algebraic approach to interpretation does not fit all use-cases, it does, however, provide a clean and modular way to define interpretations, that can be used in the synthesis process. Especially, this implementation of the approach allows the static type checker of Python to check, that all interpretations are consistent with the signature.

6. Benchmarks

In the previous chapters, we have introduced different modeling techniques. In particular, we have introduced techniques using FCL and techniques using FCLP. We have seen two kinds of predicates, such as term predicates and literal predicates. The latter are separated into `SetTo` predicates and normal predicates. We introduced the concept of inference of literal variables and have seen three modes of handling the generation of literal substitutions.

In this chapter, we compare these different modeling approaches based on their performance. We will mostly be using the running example of finding solutions for mazes, since this example can easily scale the size of the maze, which allows us to see the scalability of the different approaches. Since FCLP is a conservative extension of FCL, CLSP can act as a synthesis framework for both FCL and FCLP. When we do not make use of the additional features of FCLP, CLSP synthesize terms according to the rules of FCL. We also compare the performance of CLSP the most recent versions of (CL)S.

All benchmarks were run on a machine with an AMD Ryzen 9 3950X CPU and 32 GiB of DDR4 RAM. The machine runs Manjaro Linux with the kernel version 6.12.1 and the benchmarks were run using CPython 3.12.7. The code for the benchmarks can be found at `https://github.com/tudo-seal/clsp-python` in the `tests/benchmarks` directory. To conduct a series of benchmarks, the `multi-benchmark` script in that directory was used. Most benchmarks are run for multiple maze sizes, up to a point where the runtime takes more than 1 minute. The results are then plotted in a graph to show the scalability of the different approaches. We did not run the benchmarks multiple times to get an average runtime, as we are not interested in the exact runtime, but in the scalability of the different approaches. The raw benchmark results are given in Appendix A.

The `multi-benchmark` script is run by executing the following command:

```
$ python -m tests.benchmarks.multi_benchmark \
   -m <benchmark_module1> \
   -m <benchmark_module2> \
   ...
   -s <start_size> -e <end_size> -n <increment> -t <timeout>
```

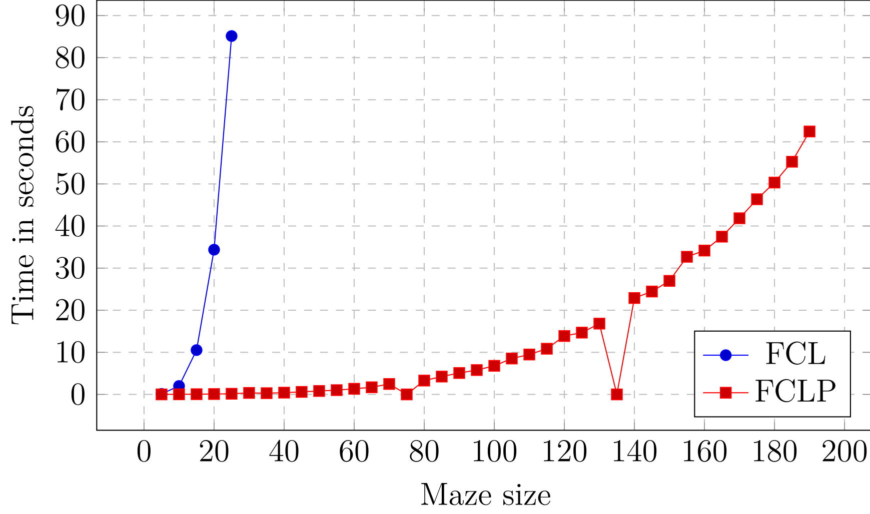

Figure 5.: Comparison of FCL and FCLP for Different Maze Sizes

6.1. Comparing FCLP to FCL

We start with a direct comparison between the FCL and FCLP approaches. We use randomly generated mazes of different sizes to see how the different approaches scale. The mazes are generated by the same seed, so that we can compare the results directly. For this comparison, we use a repository using the approach used in Example 10 to find solutions for mazes in FCL and the following repositories for an $n \times n$ maze $\mathcal{M}$ in FCLP:

$$
\begin{aligned}
\Delta &= \{pos : \{p \in \{0, \ldots n\}^2 \mid \mathcal{M}(p) = \mathit{true}\}\} \\
\Gamma &= \{\textsc{Up} : \langle \beta : pos \rangle \Rightarrow \langle \alpha : pos \rangle \Rightarrow (\alpha := (\pi_1(\beta), \pi_2(\beta) + 1) \\
&\qquad \Rightarrow \langle\!\langle \mathit{from} : path(\alpha) \rangle\!\rangle \Rightarrow path(\beta), \\
&\quad \textsc{Down} : \langle \beta : pos \rangle \Rightarrow \langle \alpha : pos \rangle \Rightarrow (\alpha := (\pi_1(\beta), \pi_2(\beta) - 1) \\
&\qquad \Rightarrow \langle\!\langle \mathit{from} : path(\alpha) \rangle\!\rangle \Rightarrow path(\beta), \\
&\quad \textsc{Left} : \langle \beta : pos \rangle \Rightarrow \langle \alpha : pos \rangle \Rightarrow (\alpha := (\pi_1(\beta) + 1, \pi_2(\beta)) \\
&\qquad \Rightarrow \langle\!\langle \mathit{from} : path(\alpha) \rangle\!\rangle \Rightarrow path(\beta), \\
&\quad \textsc{Right} : \langle \beta : pos \rangle \Rightarrow \langle \alpha : pos \rangle \Rightarrow (\alpha := (\pi_1(\beta) - 1, \pi_2(\beta)) \\
&\qquad \Rightarrow \langle\!\langle \mathit{from} : path(\alpha) \rangle\!\rangle \Rightarrow path(\beta), \\
&\quad \textsc{Start} : pos(0, 0)\}
\end{aligned}
$$

The FCLP repository differs from the repositories introduced in Example 24 in that the order of the variables α and β is swapped, so that the value of α is uniquely determined

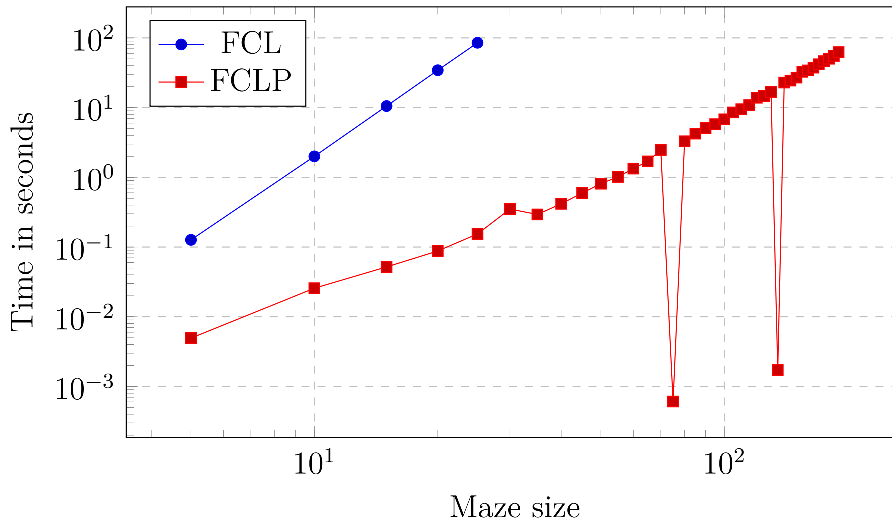

Figure 6.: Comparison of FCL and FCLP for Different Maze Sizes (Logarithmic Scale)

by the value of β and β can be inferred during inhabitation, and that there are no variables in the types, that need to range over their respective group. The impact of this change is evaluated in Section 6.3. We also give the intersection type of each movement combinator not as an arrow type, but only as the target type. We incorporate the source part of the arrow type as a term parameter. When term parameters are used in such a way, they are not considered to be part of the generated multi-arrows. Since we never want to inhabit types in the form of $path(\alpha) \to path(\beta)$, but only types in the form of $path(\alpha)$, we do not need to generate multi-arrows for these types. The impact of this change is evaluated in Section 6.4

The FCL repository is implemented in the file

```
benchmark_maze_vanilla_cls.py
```

and the FCLP repository is implemented in the file

```
benchmark_maze_posparams_setto_freel_tuple_reversed.py
```

The results of the benchmark is shown in Figures 5 and 6. We see, that the FCLP approach is significant faster than the FCL approach for all maze sizes. Notebly, there are two dips in the runtime of the FCLP approach: One at 75 and one at 135. This is due to the fact, that the random maze generator generates mazes that are not solvable, which causes the FCLP approach to exit immediately, because of the early discarding of subtitutions in the FCLP approach. For comparison, the FCL approach takes 12157

seconds (∼3 hours and 22 minutes) for a maze of size 75. It does not detect that the maze is unsolvable and tries to find solutions.

Concerning the scalability, the log-log plot in Figure 6 shows, that both approaches scale polynomial with the size of the maze, but the runtime of the FCLP apporach has a lower slope than the FCL approach, signifying a lower exponent.

6.2. Comparing Term and Literal Predicates to Generate-and-Test

Both term predicates and literal predicates can be used to impose restrictions on the synthesized terms, as well as approaches based on generating all possible terms and filtering out unwanted terms afterwards. In this section, we compare the performance of the three approaches in different scenarios. We use the loop-free maze solving problem as a benchmark, as introduced in Example 30. Larger versions of the maze retain exactly two solutions by enlarging the upper blocked part of the maze. We consider five different repositories, each corresponding to a different approach to restrict the solutions:

- The first repository uses term predicates, that filter out solutions containing loops in the enumeration. The repository is implemented in the file

  `benchmark_maze_loopfree_term.py`

- The second repository adds a literal, that keeps track of the already visited positions in the maze, and only generates substitutions that do not visit a position twice. The repository is implemented in the file

  `benchmark_maze_loopfree_literal.py`

- The third repository uses term predicates, but utilizes a simple cache for the result of the computation of the visited positions. The repository is implemented in the file

  `benchmark_maze_loopfree_cache_term.py`

- The forth repository uses the same approach as the second repository, but implements the powerset as a virtual group (See Subsection 5.2.4). The repository is implemented in the file

  `benchmark_maze_loopfree_virtual_literal.py`

- The last repository uses the generate-and-test approach. It firstly computes all solutions, and filters the solutions externally, that contain at least one loop. Since the number of potential solutions is infinite, we manually stop the enumeration after finding the two loop-free solutions. The repository is implemented in the file

  `benchmark_maze_loopfree_generate_and_test.py`

The results are plotted in Figure 7. Neither results for the non-virtual literal repository, nor the generate-and-test repository are shown in the figure, since their runtimes far exceed the runtimes of the other repositories. The non-virtual literal repository took 0.1 second for a maze of size 5, 30 seconds for a maze of size 6 and the computation did not fit in 32 GiB of RAM for size 7. The generate-and-test repository took 0.5 second for a maze of size 5, 16 seconds for a maze of size 6, 56 seconds for a maze of size 7 and 513 seconds for a maze of size 8. The repositories for cached and non cached term predicates differ only in the `@functools.cache`[1] annotation on the `visited` function, that extracts the visited positions from a given term. The cache is not shared between invocations of different maze sizes. While the runtimes for the approach using literal predicates and virtual groups is slower than the optimized approach using term predicates, the approach using literal predicates is still faster than the non-optimized approach using term predicates. The advantage of the approach using literal predicates and virtual groups is, that the relevant data – in this case the visited positions – is directly accessable as input of a literal predicate, while the term predicate needs to extract the relevant data from the term. Depending on the complexity of the extraction, an approach based on literal predicates and virtual groups can be faster than an approach based on term predicates.

As we can seen in Figure 8, the runtime of all approaches scale polynomial. The virtual literal approach scales roughly the same as the non-cached term predicate approach. The non-cached term predicate approach scales significantly worse than the other approaches. Notably the runtime of the non-cached term predicate approach starts out faster than the virtual literal apporach.

As shown in Section 2.2, a term that is solely composed of movement combinators carries enough information to construct a path. Considering the results of the benchmarks in Figure 7, that shows the improved runtime when using term predicates, we investigate the performance of an approach, that only uses term predicates. The repository for

[1] `https://docs.python.org/3/library/functools.html#functools.cache`

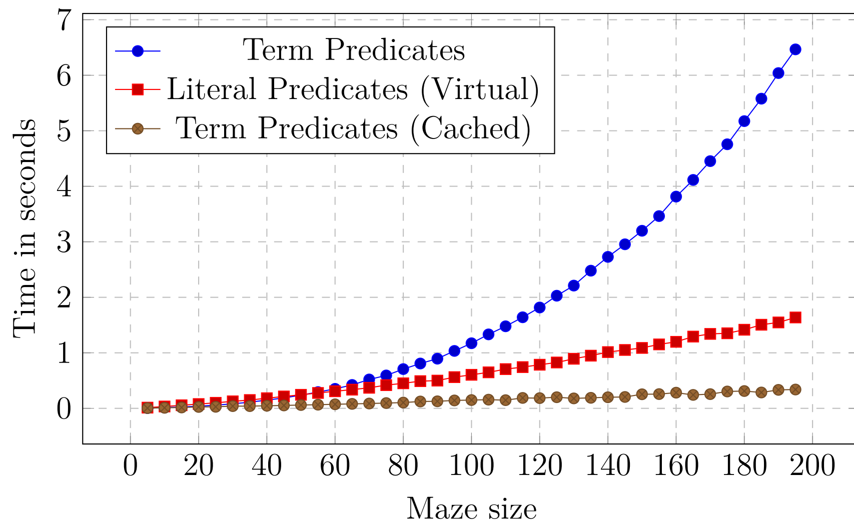

Figure 7.: Runtime for Loop-Free Solutions Using Literal, Term and Cached Term Predicates

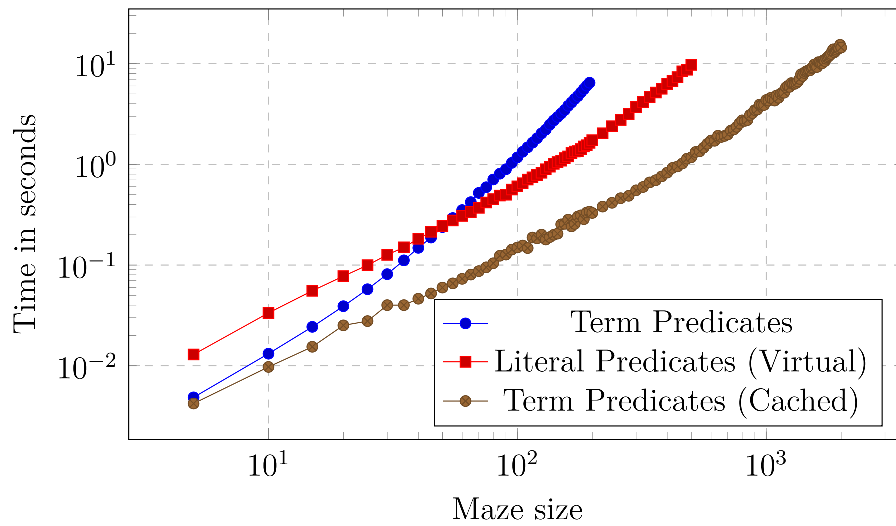

Figure 8.: Runtime for Loop-Free Solutions Using Literal, Term and Cached Term Predicates (Logarithmic Scale)

Maze size	Runtime in seconds
5	0.048
6	0.468
7	4.791
8	35.542
9	318.296

Table 2.: Runtime of the Approach Using Only Term Predicates

this approach is implemented in the file `benchmark_maze_posparams_onlytermparam.py`. The type of each movement combinator is in the form of

$$\langle\langle \mathit{from} : \mathit{path} \rangle\rangle \Rightarrow (\mathcal{M}(\mathit{current_position}(\mathit{from})) = \mathit{true}) \Rightarrow \mathit{path}$$

where *current_position* computes the current position of the path by traversing the movement combinators. We add a combinator

$$\textsc{End} : \langle\langle \mathit{from} : \mathit{path} \rangle\rangle \Rightarrow (\mathit{current_position}(\mathit{from}) = (n-1, n-1)) \Rightarrow \mathit{end}$$

that checks, if the current position is the end position of the maze, the synthesis request changes to

$$\{M \mid \Gamma, \emptyset \vdash M : \mathit{end}\}$$

The results in Table 2 show, that even on small maze sizes, the runtime is significantly slower than the approach using literals and literal predicates. More over, since filtering is done in the enumeration step, the runtime scales with the amount of terms generated. In this benchmark, ten solutions are generated. When generating 20 solutions for a maze of size six, the runtime increases to 1.804 seconds.

We conclude, that both literal predicates and term predicates are useful tools for filtering solutions. Both are faster, than using a traditional generate-and-test approach, but there is no overall “best” approach, it depends on the actual use-case, which approach is more suitable.

6.3. Inference of Literal Variables

In Section 5.3, we have seen that the inference of literal variables can be handled in different ways. We list three different modes of handling the generation of literal substitutions. In

this section, We compare the performance of the three different approaches using two different repositories for each approach. Both repositories are based on the repository in Section 6.1 for finding solutions for mazes, but differ in the order of literals for the parameterized types of the movement combinators. The first repository uses the structure introduced in Section 6.1 with the order of the literal variables reversed, while the second repository uses the order introduced in Example 10. We use the following modes for each repository:

- The first mode is using the default mode, meaning `cache=False` and `infer=True`. The repositories are implemented in the files
  - `benchmark_maze_posparams_setto_freel_tuple_reversed.py`
  - `benchmark_maze_posparams_setto_freel_tuple.py`
- The second mode is using `cache=True` and `infer=True`. The repositories are implemented in the files
  - `benchmark_maze_posparams_setto_freel_tuple_reversed_cache.py`
  - `benchmark_maze_posparams_setto_freel_tuple_cache.py`
- The third mode is using `infer=False`. The repositories are implemented in the files
  - `benchmark_maze_posparams_setto_freel_tuple_reversed_noinfer.py`
  - `benchmark_maze_posparams_setto_freel_tuple_noinfer.py`

The results are plotted in Figure 9. We did not include the results for the approaches using `infer=False` and `infer=True, cache=True` for the reversed repositories, since they do not differ from the results of the non-reversed variant. It should be of no surprise, that the approach using Inference, `SetTo` and the reversed order of literal variables is the fastest. Evaluating the data, we make the following observations:

- When using caching, or no inference, the order of the literal variables does not matter. The runtime is the same for both orders.
- Caching is faster than non caching in the case, when at least one literal variable need to range over its literal group, but significantly slower, if no variable needs to range over its literal group.
- Inference can be even slower than not using inference at all, when inference and `SetTo` restrict the same variable.

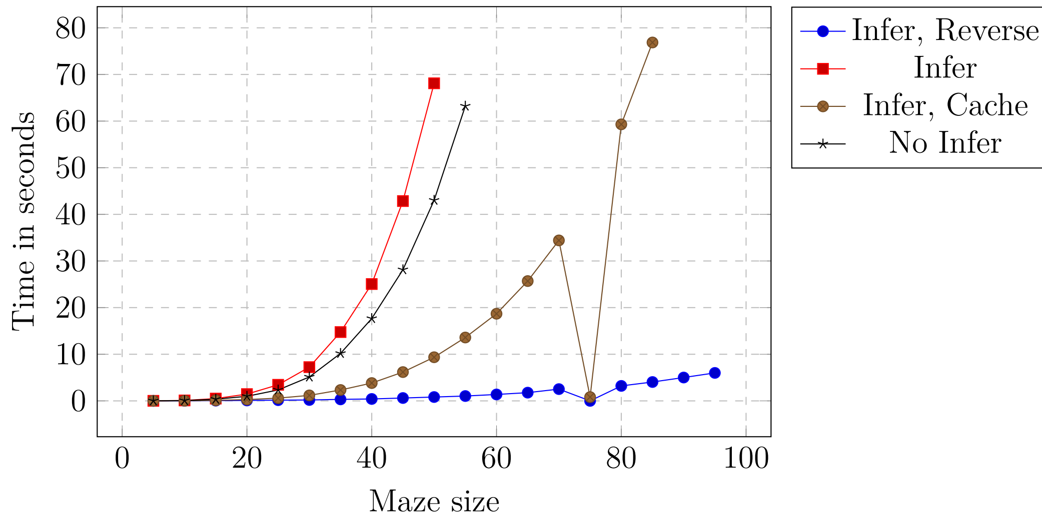

Figure 9.: Comparing Different Strategies of Filtering Substitutions

- No approach leads to a asymtotically faster runtime, all approaches scale polynolial with the size of the maze.

6.4. Source as Parameter

Next, we examine the performance impact of using the source of an arrow type as a term parameter. We compare the performance of the repository shown in Section 6.1 with a version, that uses $path(\alpha) \to path(\beta)$ instead of $\langle\langle from : path(\alpha)\rangle\rangle \Rightarrow path(\beta)$. We run the comparison once with inference and once without inference. We only consider a size of 95 for the run with inference and a size of 30 for the run without inference, since these sizes yield runtimes of similar lengths. The repository with the approach using the source as parameter is implemnted in the files

- `benchmark_maze_posparams_setto_freel_tuple_reversed.py`
- `benchmark_maze_posparams_setto_freel_tuple_reversed_noinfer.py`

the repository with the approach using the source in the intersection type is implemented in the files:

- `benchmark_maze_setto_freel_tuple_reversed.py`
- `benchmark_maze_setto_freel_tuple_reversed_noinfer.py`

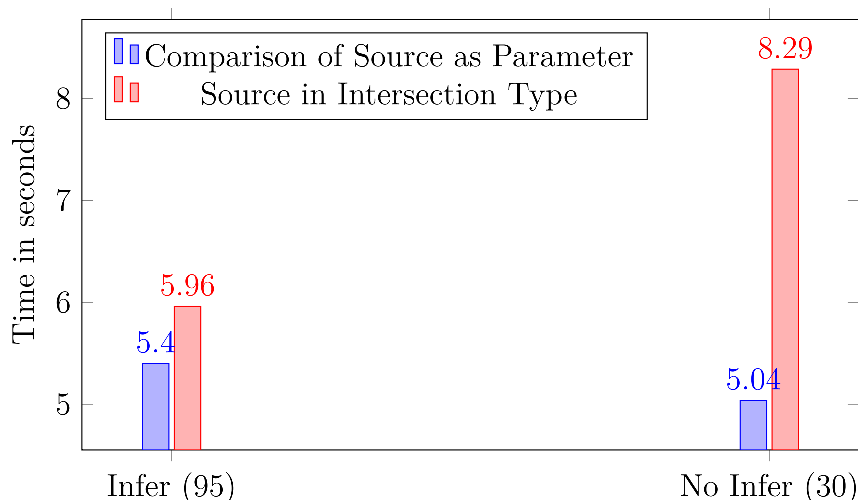

Figure 10.: Source as Parameter and Source in Intersection Type

The results are shown in Figure 10. We see, that the approach using the source as parameter is faster than the approach using the source in the intersection type, but the difference is more significant, when inference is not used. This is due to the fact, that the inference leads to a smaller amount of possible substitutions and therefore a smaller amount of candidates, that need to be subtype checked against. Since having the source not be part of the intersection type leads to fewer multi-arrows and therefore fewer subtype checks, the difference is more pronounced when inference is not used.

6.5. Comparing Python Implementations

While CPython is the default implementation of the Python language, there are other implementations. One of the most popular alternative implementations is PyPy[2], which is a Just-In-Time (JIT) compiler for Python and regularly outperforms Python. CLSP is fully compatible with PyPy, helped by the fact, that it does not rely on external libraries. While this does not test features of CLSP, the impact of using a more performant interpreter is a useful information for users of CLSP, that want to use it in a production environment. We use the benchmark of the FCLP repository detailed in Section 6.1 to compare the performance of PyPy in version 7.3.17 and CPython in version 3.12.7.

The results are shown in Figure 11. We see, that PyPy is significantly faster than CPython for all maze sizes. The difference is more pronounced for larger maze sizes,

[2] `https://www.pypy.org/`

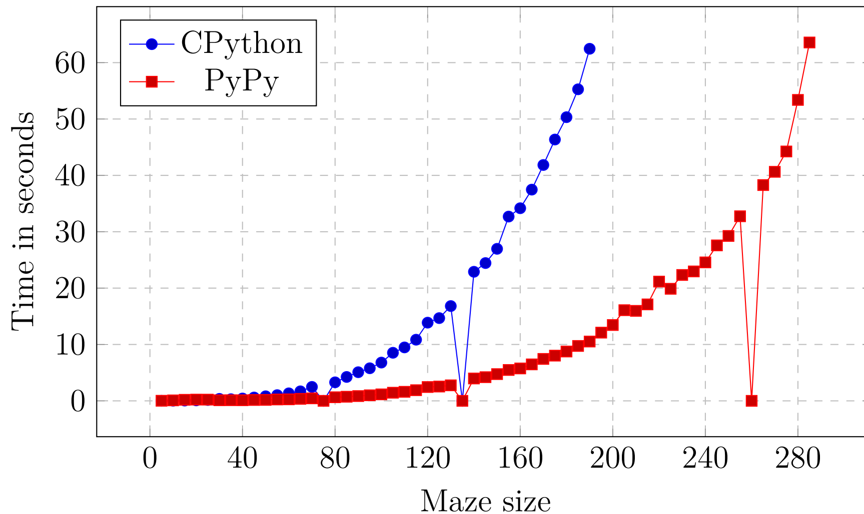

Figure 11.: Comparison Between PyPy and CPython

since the JIT compiler of PyPy can optimize the code better, when it has more data to work with.

6.6. Comparing CLSP to Previous (CL)S Implementations

Lastly, we compare the performance of CLSP to previous implementations of (CL)S, specifically the Scala implementation [5], the F# implementation [20] and the previous Python implementation [6]. Since finding solutions for mazes is a common benchmark, there exists a repository for this problem already implemented in most previous implementations. For the benchmark of CLSP, we use the repository detailed in Section 6.1. For the benchmark of the Scala version, we use the repository implemented in the following file in the `cls-scala` Project, that defines a 30×30 maze:

```
examples/src/main/scala/org/combinators/
    cls/examples/LabyrinthBenchmark.scala
```

For the F# version, we use the repository implemented as part of the subproject `cls-fsharp-experiments` inside the `cls-fsharp` project, with a 30×30 maze. For the Python version, we use the repository implemented in the following file in the `cls-python` Project, that defines a 10×10 maze:

```
tests/benchmark_labyrinth.py
```

Implementation	Runtime in seconds	Maze size
CLSP	0.351	30
Scala	74.286	30
F#	43.244	30
Python	297.13	10

Table 3.: Comparison of CLSP to Previous Implementations

While the maze size of the Python version is significantly smaller than the other versions, it gives us a lower bound on the runtime of the Python version. All benchmarks were run on the same machine as the benchmarks for CLSP. The benchmark results are shown in Table 3. We clearly see, that CLSP is multiple orders of magnitude faster than all previous versions of (CL)S, even when taking into account the different sizes.

7. Parallelized Inhabitation

When reviewing the inhabitation algorithm for both FCLP and FCL (Algorithm 5 and Algorithm 2 respective) we can see that both algorithms follow a classical producer-consumer pattern. In each iteration of the algorithm, a target type is consumed from the *queue* and several new types are produced and added to the *queue*.

This pattern lends itself to parallelization, by separating the loop from the main algorithm and spawning multiple workers. Each worker takes a type from the *queue* and produces new types, which are then added to the *queue*. Using traditional synchronization primitives, such as mutexes or semaphores, the workers can synchronize their access to the *queue*.

In this chapter, we present and evaluate a parallelized version of the inhabitation algorithm for FCLP. This version is not implemented as part of CLSP, but a prototype based on an earlier version of CLSP can be found in the main repository[1] in the branch `multiproc`.

7.1. Implementation

Unfortunately, CPython prior to version 3.13 does not support for true thread-based parallelism due to the Global Interpreter Lock (GIL). While it is possible to spawn multiple threads in Python, only one thread can execute Python code at a time, so a program in python cannot utilize multiple CPU cores. To work around this limitation, we use the `multiprocessing` module, which allows us to spawn multiple processes, each with its own Python interpreter and memory space. This way, we can utilize multiple CPU cores, but we have to be careful with the communication between the processes, as we cannot easily share memory between them. Furthermore, the `multiprocessing` module is implemented differently on Windows, Linux, and macOS. The prototype implementation is optimized for Linux, but should work on Windows and macOS as well, albeit with reduced performance. A major drawback of the `multiprocessing` module is, that each process needs its own memory space, that cannot be shared. When spawning a

[1] `https://github.com/tudo-seal/clsp-python`

new process, the memory of the parent process is copied to the child processes, which is very expensive in terms of memory and time.

We use the `SharableList`[2] class from the `multiprocessing` module to share the *queue* between the worker processes and to share the generated rules for the parameterized tree grammar. The use of the `Queue`[3] class from the `multiprocessing` module was considered, but was deemed not to be flexible enough for the use-case.

Since the `SharableList` can only contain primitive types, we have to serialize types and rules to a `bytes` object before adding them to the *queue*. Rules are serialized by using the `pickle`[4] module, which is part of the Python standard library, Types are serialized by using a custom serialization function, that converts a type to a `byte` object and back. We use the custom serialization function because the `pickle` module adds a lot of overhead both in terms of memory and time. Since we need to serialize and deserialize types very often, we need to keep the overhead as low as possible. We can use `pickle` for the rules because they are only serialized and deserialized once. We also serialize the combinators by replacing them with a Universal Unique Identifier (UUID) and storing the mapping between the UUID and the combinator in a separate dictionary in the main process.

The workers themselves take a type from the *queue*, deserialize it, and run the inner part of the loop of the original algorithm. When they produce a subquery, they serialize it and add it to the *queue* and when they compute a rule for the parameterized grammar, it is serialized and added to the list of rules. Access to the *queue* is guarded by a semaphore, that counts the number of items in the *queue*. When a worker takes a type from the *queue*, it decrements the semaphore and when it adds a type to the *queue*, it increments the semaphore. When a worker tries to take an item from an empty queue, it waits until there are elements in the queue. In addition to the semaphore for the queue access, we also globally count the number of types added to the queue and the number of types, that are completed by the workers. This way we can detect when no new types are generated, terminate the workers and carry on with the rest of the algorithm.

7.2. Evaluation

To evaluate the parallelized version of the inhabitation algorithm, we use benchmarks based on finding solutions for mazes, similar to Chapter 6. We do not only vary the size

[2] `https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.managers.SharedMemoryManager.ShareableList`

[3] `https://docs.python.org/3/library/multiprocessing.html#multiprocessing.Queue`

[4] `https://docs.python.org/3/library/pickle.html`

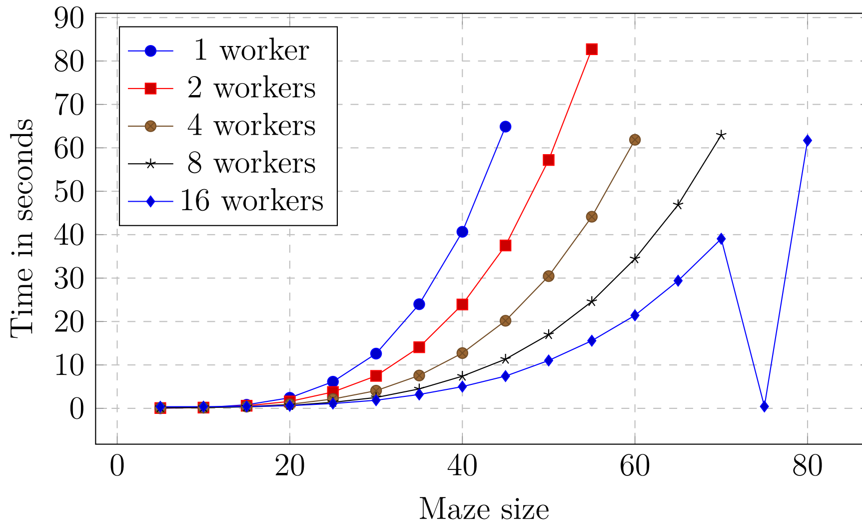

Figure 12.: Runtimes for Different Maze Sizes With no, 2, 4, 8 and 16 Workers

of the maze, but also the number of workers available to the program. We use the same hardware as in Chapter 6.

The results for 2, 4, 8 and 16 workers, as well as the results for a version without parallelization are shown in Figure 12. It is important to note that these results are not directly comparable to the results from Chapter 6, since this version of CLSP does not include many of the optimizations from Subsection 4.3.3. Instead, we compare the results of the parallelized version of this implementation to the results of the non-parallelized version this implementation is based on. Figure 13 shows select data points to better compare them. This figure also includes the runtime for 17 workers, which is not shown in Figure 12.

While we see a significant speedup when using multiple workers (Figures 12 and 13), the fact, that each process needs to hold a copy of the memory of the parent process, limits the applicability in very large repositories. Additionally, Figure 13 shows, that in very small repositories, the overhead of spawning new processes and copying the memory of the parent process to the child processes increases the runtime. Overall, the parallelized version of the inhabitation algorithm scales roughly linear with the number of workers, up to a number of workers equal to the number of CPU cores available. The benchmark hardware has 16 physical cores, so we see a significant speedup up to 16 workers, with a slight decrease in performance when using 17 workers. This observation holds true for different hardware configurations as well, a benchmark on a machine with

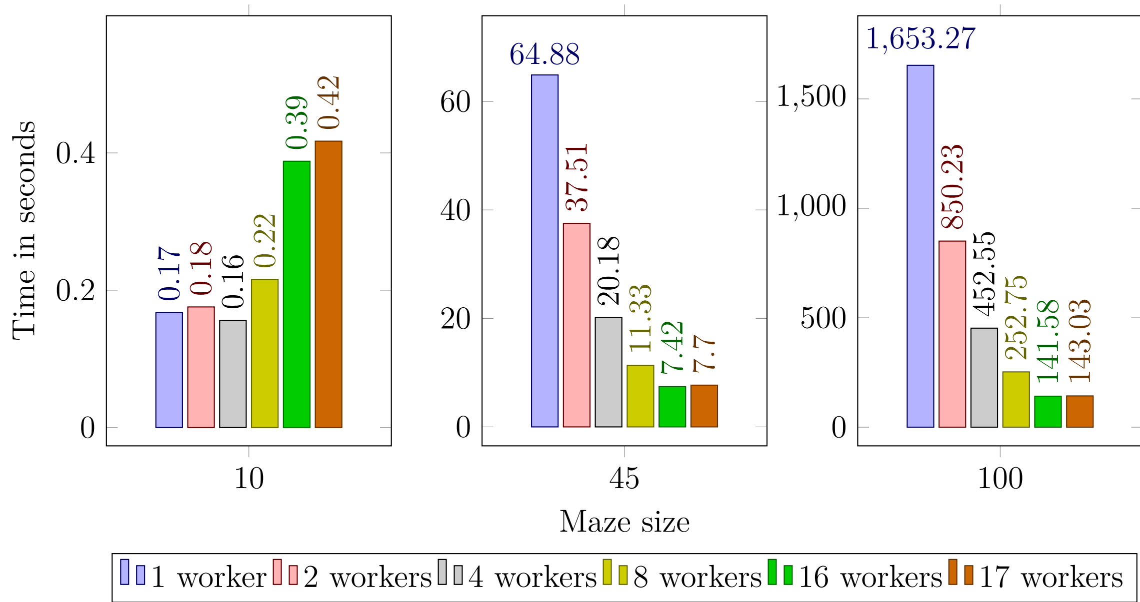

Figure 13.: Comparison Between Different Maze Sizes With 1, 2, 4, 8, 16 and 17 Workers

8 physical cores shows a significant speedup up to 8 workers, with a slight decrease in performance when using 9 workers.

7.3. Thread-based parallelism

Since version 3.13, Python supports deactivating the Global Interpreter Lock (GIL) at compile time of the interpreter[5] and therefore allows for true thread-based parallelism. While this is not the default setting, it shows that an approach of using threads for parallelism is possible using Python. Since the main bottleneck of the multiprocessing approach was memory usage and inter process communication, that is inherently tied to an approach based on processes, we can expect a thread-based approach to be more efficient and better scaling. This is a very recent development, so it is not included in the version of CLSP described in this thesis, but future versions of CLSP should strive to implement this.

[5] `https://docs.python.org/3/whatsnew/3.13.html#whatsnew313-free-threaded-cpython`

8. Applications

In this chapter, we present two applications developed using the CLSP framework. The author was involved in the development of both applications. The first application synthesizes reactive systems together with certificates for reachability and safety properties [44]. This application makes use of the fact, that a term in CLSP can be interpreted in multiple ways. Terms synthesized in this application are interpreted both as strategies for reachability and safety games, as well as Coq[1] proofs, that certify the correctness of each strategy.

In the second application, simulation models of material flow systems in the field of logistics are synthesized [49]. This is a continuation of the work done in [45]. This application makes use of the variance CLSP provides, due to enumerating the all terms for a given specification.

We briefly introduce the applications with a focus on the usage of CLSP and discuss the main results and insights relevant to the CLSP framework.

8.1. Reactive System Synthesis

A reactive system is a system that continuously interacts with its environment. Synthesis of reactive system is a classical problem in computer science going back to the works of Alonzo Church [10][27]. Traditionally, synthesis is done *ex nihilo*, i.e., the system is synthesized from scratch based on a specification. In [44], the authors present a component-based synthesis approach to synthesize reactive systems by using CLSP. More specifically, the authors use the CLSP framework to synthesize uniform memoryless strategies for a fragment of LTL, namely safety and reachability games on arenas. Together with each strategy, a Coq proof is synthesized that certifies the correctness of the strategy. Classically, correctness of reactive synthesis is done by a *correct-by-construction* approach, where the correctness of the results is guaranteed by the (proven) correctness of the system. In this work, while the synthesis process and implementation is not proven to be

[1] `https://coq.inria.fr/`

correct, the correctness of each result is guaranteed by a Coq proof, that is generated alongside each strategy.

We will briefly give an intuition of the game theoretic background and show how the CLSP framework is used in this application. An *arena* is a directed graph with two sets of vertices, V_P and V_O, representing the vertices controlled by a player and an opponent, respectively. We call the vertices in V_P *player positions* and the vertices in V_O *opponent positions*. A *memoryless uniform strategy* is a function that assigns to each position in V_P a successor position. A *play* is an infinite sequence of consecutive positions in the arena. A play is *consistent* with a strategy if the next position of V_P in the play is chosen according to the strategy. Given a set of positions U, the *reachability condition* for a play is met, if the play visits a position in U at least once. The *safety condition* is met if the play never visits a position outside U. A strategy is *uniformly winning* if all plays from positions in V_P, that are consistent with the strategy, satisfy the condition.

Intuitively, for a player position, the next position is chosen by the strategy, and for an opponent position, an adversary chooses the next position. For reachability, we want to synthesize strategies, that ensure that the player can reach a target set of position from a maximum amount of starting positions, no matter how the opponent plays. For safety, we want to synthesize strategies, that ensure that the player never leaves a set of positions, no matter how the opponent plays.

Traditionally, such strategies are computed by the attractor method, that iteratively computes a winning region and strategy for the player [50]. Using the attractor method, one can only infer strategies, that use the least number of steps to fulfill the condition. Using the CLSP framework, we synthesize all strategies, thereby covering the complete variance of strategies for a given specification. A strategy is represented as a term in CLSP, that is interpreted as a mapping from player positions to outgoing edges as well as a Coq proof, that certifies the correctness of the strategy.

The main insight of this application is that, that a strategy term in CLSP holds enough information to act as a correctness proof. Additionally, two kinds of variance are identified in this application.

1. Variance, that is bound by dependencies in the specifications. This includes choices for successors, that can (eventually) violate the condition.

2. Variance, that is unbound. This includes choices for successors, when the condition is already met, such as successors of positions, that are already in the target set.

The parameters and predicates of CLSP align themselves well to specify the first kind of variance, and archive better results, than a generate-and-test approach (See also

$$\begin{aligned}
\Gamma_R = \{\textsc{Start} : &\langle U : \hat{U}\rangle \Rightarrow R(0), \\
\textsc{Ex} : &\langle (s, T) : E_P^{\cup}\rangle \Rightarrow \langle n_+ : \mathbb{N}\rangle \Rightarrow \langle n : \mathbb{N}\rangle \Rightarrow (n = n_+ - 1) \Rightarrow \\
&\langle\!\langle M : R(n)\rangle\!\rangle \Rightarrow (s \notin |M| \wedge T \cap |M| \neq \emptyset) \Rightarrow R(n_+), \\
\textsc{All} : &\langle (s, T) : E_O^{\cup}\rangle \Rightarrow \langle n_+ : \mathbb{N}\rangle \Rightarrow \langle n : \mathbb{N}\rangle \Rightarrow (n = n_+ - 1) \Rightarrow \\
&\langle\!\langle M : R(n)\rangle\!\rangle \Rightarrow (s \notin |M| \wedge T \subseteq |M|) \Rightarrow R(n_+), \\
\textsc{Fin} : &\langle (V_P, V_O, E) : \hat{\mathcal{A}}\rangle \Rightarrow \langle n : \mathbb{N}\rangle \Rightarrow \langle\!\langle M : R(n)\rangle\!\rangle \Rightarrow \\
&\Big((\forall v \in V_P \setminus |M|.\{t \mid (v, t) \in E\} \cap |M| = \emptyset) \wedge \\
&(\forall v \in V_O \setminus |M|.\{t \mid (v, t) \in E\} \not\subseteq |M|)\Big) \Rightarrow \mathsf{REACH}\}
\end{aligned}$$

Figure 14.: Repository to Compute Strategies for Reachability

Chapter 6). For the second kind of variance, it was found that a straightforward product is more efficient than using CLSP. For that reason, only the first kind of variance is computed using CLSP, while the second kind is computed as part of the interpretations, so a term in CLSP does not depict a single strategy, but a family of strategies.

Two repositories are given in the paper, one for reachability and one for safety games. The repositories encode an algorithm for reachability and safety games respectively, the input data of the algorithm – the arena – is provided as part of the literal repository. This way, the combinator repository can be oblivious to the actual arena, and the same combinator repository can be used for arbitration arenas, archiving a separation of data and code, that is not possible using an approach based on FCL.

The repository for reachability is depicted in Figure 14. $|M|$ computes the winning region associated with a combinatory term M as follows:

$$\begin{aligned}
|\textsc{Start}\, U| &= U \\
|\textsc{Ex}\, (s, T)\, n_+\, n\, M| &= \{s\} \cup |M| \\
|\textsc{All}\, (s, T)\, n_+\, n\, M| &= \{s\} \cup |M| \\
|\textsc{Fin}\, \mathcal{A}\, n\, M| &= |M|
\end{aligned}$$

The literal groups $\hat{U}$ and $\hat{\mathcal{A}}$ are singleton sets of the initial set of positions and the arena, respectively, $E_P^{\cup}$ and $E_O^{\cup}$ contain tuples of player and opponents positions, together with all their adjacent positions in the arena.

Terms consist of a Start combinator, that initializes the winning region with the initial set of positions, repeated Ex and All combinators, that add positions to the

winning region, if their condition is met, and a FIN combinator, that checks, that the region is maximal. These terms contain enough information to generate strategies and certificates. We give the interpretation as strategies as.

Definition 42: Strategy interpretation [44, Definition 10]

Let $\mathcal{A} = (V_P, V_O, E)$ be an arena. Given a region $U \subseteq V$, let $strats_{\mathcal{A}}(U)$ be the *set of all strategies* on $\mathcal{A}$ restricted to player positions in the region U:

$$strats_{\mathcal{A}}(U) = \prod_{s \in U \cap V_P} \Big(E \cap (\{s\} \times V) \Big)$$

Given a region $U \subseteq V$ and combinatory term M with $\Gamma_R, \Delta^U_{\mathcal{A}} \vdash M : \mathsf{REACH}$, the set $\sigma_{\mathcal{A}}(M)$ contains *strategies* represented by M:

$$\begin{aligned}
\sigma_{\mathcal{A}}(\textsc{Start}\, U) &= strats_{\mathcal{A}}(U) \\
\sigma_{\mathcal{A}}(\textsc{Ex}\,(s,T)\, n_+\, n\, M) &= \{\sigma \cup \{(s,t)\} \mid \sigma \in \sigma_{\mathcal{A}}(M), t \in T \cap |M|\} \\
\sigma_{\mathcal{A}}(\textsc{All}\,(s,T)\, n_+\, n\, M) &= \sigma_{\mathcal{A}}(M) \\
\sigma_{\mathcal{A}}(\textsc{Fin}\, \mathcal{A}\, n\, M) &= \{\sigma \cup \sigma' \mid \sigma \in \sigma_{\mathcal{A}}(M), \sigma' \in strats_{\mathcal{A}}(V_P \setminus |M|)\}
\end{aligned}$$

We can see, that the unbound variance is represented by the $strats_{\mathcal{A}}(U)$ function, that simply computes a product.

The repository for safety games works analogously.

8.2. Synthesis of Simulation Models

In industrial applications, simulation models are used to predict the behavior of systems. In [49], the authors present a component-based synthesis approach to synthesize simulation models for material flow systems in the field of logistics. Instead of building simulation models by hand, the CLSP framework is used to synthesize simulation models given an input of available components. Material flow systems are systems that transport material from one location to another, using, for example, conveyor belts, robots, or people. Locations themselves convert one kind of material into another kind of material. These are usually machines or workbenches. Such systems can be modeled as a directed graph, with the vertices representing locations and the edges representing the flow of material. A major challenge using the component-based approach is, that the fundamental output structure of the synthesis is a tree-like structure, while the simulation models are graphs. For this reason, a suitable subset of simulation models is chosen, that is representable as

trees, but still captures a significant portion of such systems. This subset is depicted in terms of *network terms*, that are built using the following grammar:

$$M, N ::= \textsc{Machine}(m) \mid \textsc{Seq}(M, N) \mid \textsc{Par}(M, N) \mid \textsc{Loop}(M) \mid \textsc{Opt}_X(M)$$

Materials in a material flow system are abstracted as *resources*, that are passed from one location to another.

MACHINEs are the basic building blocks of the network, that represent a single location or a single component and have a set of *input* and *output* resources. SEQ and PAR represent sequential and parallel composition of networks, respectively. A LOOP is a network, that loops arbitrary often, and $\textsc{Opt}_X$ is a family of networks over a set of resources X, that can be skipped, passing the resources X along.

For the sequential composition, the input is the input of the first network and the output is the output of the last network. Furthermore, the output of the first network must be the same as the input of the second network. For the parallel composition, the input is the input of both networks and the output is the output of both networks. A loop feeds part of its output back to its input, so the input of a loop is the input of the inner network without the output of the inner network, and the output is the output of the inner network without the input of the inner network. We require the input and output of the inner network not to be disjoint and not to be equal. The input of an optional network over X is X combined with the input of the inner network, and the output is X combined with the output of the inner network.

Combinators in CLSP are the term constructors for the network terms. The requirements for the input and output of the networks are encoded as predicates in the combinator repository. For the literal repository, the set of available machines is used. The terms synthesized by CLSP are then interpreted as XML-Files for AnyLogic[2], a simulation software, that can be used to simulate the material flow system.

The main insight of this application is the importance of normal forms for the terms. When considering the PAR and SEQ combinator, we recognize that both the compositions are associative, meaning $\textsc{Par}(M, \textsc{Par}(N, O)) = \textsc{Par}(\textsc{Par}(M, N), O)$ and $\textsc{Seq}(M, \textsc{Seq}(N, O)) = \textsc{Seq}(\textsc{Seq}(M, N), O)$. Similarly, the LOOP and $\textsc{Opt}_X$ combinator are idempotent, meaning $\textsc{Loop}(\textsc{Loop}(M)) = \textsc{Loop}(M)$ and $\textsc{Opt}_X(\textsc{Opt}_X(M)) = \textsc{Opt}_X(M)$. Since CLSP synthesizes all terms for a given specification, the same interpretation could be synthesized in multiple ways. To avoid redundancy, only normalized terms are synthesized, in which the first argument of a PAR or SEQ combinator is never a PAR or SEQ combinator itself and the inner network of a LOOP or $\textsc{Opt}_X$ combinator is never a LOOP or $\textsc{Opt}_X$ combinator itself.

[2] `https://www.anylogic.com/`

To enforce this, two approaches are considered. The first approach uses intersection types, by annotating each type, that is not a Par, Seq, Loop or Opt combinator with the type *noPar*, *noSeq*, *noLoop* and *noOpt*, respectively. The second approach uses a term predicated that checks if the first argument of a Par or Seq combinator is a Par or Seq combinator itself and if the inner network of a Loop or Opt_X combinator is a Loop or Opt_X combinator itself. Both approaches are compared in the paper, and it is found that the term predicate approach is generally faster.

9. Conclusion

We have presented a new version of the Combinatory Logic Synthesizer, called CLSP, that is based on Finite Combinatory Logic with Predicates (FCLP). We will evaluate the goals set in the introduction, particularly when compared to previous versions of (CL)S, and give an outlook on future work.

The following goals were formulated in the introduction: *User-friendliness*, *Execution speed* and *Portability*.

User-friendliness While no broad user study was done, due to the still limited user base of CLSP, we evaluate the user-friendliness of the framework based on the feedback of users, that have used the framework in the context of the applications presented in Chapter 8. The feedback is generally positive, with users specifically praising the executing speed of the framework, which made the applications feasible in the first place, which feeds into the second goal. Furthermore, the ability to directly use numerical values as part of the literal set was an improvement over previous versions of (CL)S, where numerical values had to be encoded in combinatory logic. One implication of this is a separation of code and data. While in FCL based versions of (CL)S, each problem instance was encoded in the repository, the addition of a literal repository allows the combinator repository to be fully abstract over a specific problem instance as code, and data for each instance is included in the literal repositories. Standard practice for earlier version is having a dynamic repository, that was computed for each instance. While this is still possible, it is in large parts not necessary, which simplifies usage and increases maintainability. Lastly, the eDSL is also well received, with the main critique being the lack of type checking regarding the literal groups and predicates, which is a limitation of Python itself. In general, the eDSL was seen helpful for programmers, that were not deeply familiar with intersection type theory and combinatory logic, as it allowed them to specify types in a way, that is more familiar to them. In conclusion, the user-friendliness of CLSP is an improvement over previous versions of (CL)S, but there is still room for improvement.

Execution speed The second goal of execution speed is evaluated in Chapter 6, where we compare the performance of synthesis using FCLP to synthesis using FCL. The performance of CLSP is a significant improvement over previous versions of (CL)S, especially when combining the different features, that are introduced in FCLP. This does not only hold true for the synthesis of solutions for mazes, but generally for all applications, that were built on top of CLSP, making the applications presented in Chapter 8 feasible in the first place.

Portability Lastly, the portability of CLSP is comparable to recent versions of (CL)S. Both the F# and Scala versions of (CL)S can easily be embedded into projects, that are utilizing the .Net or Java Virtual Machine, or interact with other projects using the standard functionalities of their respective Foreign Function Interface. The previous Python version of (CL)S is exactly as portable as CLSP, as it shares the following traits, namely its implementation language, Python, and the lack of external dependencies. The Python interpreter is easily embeddable into other projects using `python.h`, which is a C-API for Python, that allows for embedding Python into C/C++ projects. One such example is the integration of CLSP into Autodesk Fusion 360[1], which is a CAD software, that allows for the creation of plugins using Python. Work on combining the component-based synthesis approach of (CL)S with the CAD capabilities of Fusion 360 has been done [9] with current iterations of the tool using CLSP [8]. Furthermore, since CLSP is a conservative extension of (CL)S, it can be used as a drop-in replacement by simply replacing the library, ensuring compatibility with previous projects. The goal of portability is the reason, the parallel inhabitation presented in Chapter 7 was ultimately put on hold, as the `multiprocessing` module behaves differently on different operating systems, which makes it difficult to guarantee the same performance on all systems. Overall, while the portability of CLSP does not improve upon previous versions of (CL)S, it was already in a good spot, and the additions of FCLP did not hinder the portability of the framework.

Concluding, the goals set in the introduction were met, with the user-friendliness of the framework being improved, the execution speed being faster, and the portability being on par with previous versions of (CL)S.

While the goals set in the introduction were met, there is still room for improvement in the framework. The eDSL behaves like an untyped logical programming language. Mypy, the default type checker for Python, has support for extensions. While it is unclear

[1] `https://www.autodesk.com/products/fusion-360`

whether proper typing support utilizing Mypy, can be implemented, since the usage of the eDSL depends highly on runtime information, it is worth investigating. As a first step, a custom runtime type checker could be implemented, that aborts if the repository is not well-typed or closed.

Another area of improvement is the parallel inhabitation, that was put on hold. The current implementation uses the `multiprocessing` module due to the Global Interpreter Lock (GIL) in CPython. Recent developments in the Python interpreter, namely the removal of the GIL in CPython 3.13, allows for a more efficient parallel implementation of CLSP based on the `threading` module, that allow for a more portable version, while simultaneously reducing some overhead. While the current implementation is based on an older version of CLSP, and it is doubtful, whether code of that implementation can be reused in a thread-based implementation, the author hopes that the ideas presented in Chapter 7 help the development of such a version.

Furthermore, we want to address the current enumeration in CLSP. By design, CLSP enumerates all terms of a given specification, to cover the variance in the search space. Depending on the specification and the interpretation, this could lead to many terms, that map to identical solutions. One way to address this is the formulation of normal forms through predicates (See Section 8.2). This is however either not always possible or may increase both the complexity of the synthesis and the inherent complexity of the specification. Another approach is a sampling-based approach. Often times one is interested only in *some* solutions per term size. There were some experiments done, using random predicates to sample the search space, but the results were not deemed usable. Since the current enumeration algorithm builds new terms based on already enumerated terms, this approach drastically limits the search space. The author strongly believes that a new enumeration algorithm is necessary to address this issue. There were some experiments done, that were based on a Monte-Carlo Tree Search [13], they need however further investigation to evaluate. Other potential approaches include the Feat [25] approach, that earlier versions of (CL)S use [3]. It is however unclear how this can be implemented together with the term predicates used in CLSP.

Finally, the author acknowledges that this thesis is mainly concerned with the implementation and the practical application of CLSP. Further research on the theoretical aspects of CLSP and by extension FCLP, is still necessary. We know that inhabitation in FCLP is undecidable, and that there is a decidable fragment of FCLP, that restricts the use of predicates to equality and disequality constraints [24]. The exact complexity of inhabitation in this fragment is not known. This fragment, while enjoying good theoretical properties, was deemed too limiting for the practical applications of CLSP. However, it

should be interesting to investigate larger fragments of FCLP, that are still decidable, and can be used in practice. The author believes, that restricting the predicates to a useful subset of specialized predicates, can lead to performance improvements, like the already implemented `SetTo` predicates.

Lastly, future development on CLSP should not only be limited to the framework itself, but also to the applications, that are built on top of it. Currently, there are ongoing works to combine the component-based synthesis approach of CLSP with composable neural networks, that are used in the context of reinforcement learning.

In conclusion, the author believes that the improvements made in CLSP, as well as the potential improvements, that were outlined in this chapter, make CLSP a valuable tool for component-based synthesis, that can be used in a variety of applications.

Bibliography

[1] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics.* Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[2] Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. "A filter lambda model and the completeness of type assignment". In: *Journal of Symbolic Logic* 48.4 (Dec. 1983), pp. 931–940. DOI: `10.2307/2273659`.

[3] Jan Bessai. "A Type-Theoretic Framework for Software Component Synthesis". PhD thesis. Dortmund: TU Dortmund University, 2019. DOI: `10.17877/DE290R-20320`.

[4] Jan Bessai, Jakob Rehof, and Boris Düdder. "Fast Verified BCD Subtyping". In: *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday.* Ed. by Tiziana Margaria, Susanne Graf, and Kim G. Larsen. Cham: Springer International Publishing, 2019, pp. 356–371. ISBN: 978-3-030-22348-9. DOI: `10.1007/978-3-030-22348-9_21`.

[5] Jan Bessai, Anna Vasileva, and George Heinemann. *cls-scala.* 2021. URL: `https://github.com/combinators/cls-scala`.

[6] Jan Bessai et al. *cls-python.* 2022. URL: `https://github.com/cls-python/cls-python`.

[7] Jan Bessai et al. "Combinatory Logic Synthesizer". In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change.* Ed. by Tiziana Margaria and Bernhard Steffen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 26–40. ISBN: 978-3-662-45234-9. DOI: `10.1007/978-3-662-45234-9_3`.

[8] Constantin Chaumet. *CLS-CAD.* URL: `https://github.com/tudo-seal/CLS-CAD`.

[9] Constantin Chaumet, Jakob Rehof, and Thomas Schuster. “A knowledge-driven framework for synthesizing designs from modular components”. In: *Procedia CIRP* 128 (2024). 34th CIRP Design Conference, pp. 304–309. ISSN: 2212-8271. DOI: 10.1016/j.procir.2024.05.096.

[10] Alonzo Church. “Application of recursive arithmetic to the problem of circuit synthesis”. In: *Summaries of the Summer Institute of Symbolic Logic* 1 (1957), pp. 3–50.

[11] Hubert Comon et al. *Tree Automata Techniques and Applications*. 2008. URL: https://inria.hal.science/hal-03367725.

[12] Mario Coppo and Mariangiola Dezani-Ciancaglini. “An extension of the basic functionality theory for the λ-calculus.” In: *Notre Dame Journal of Formal Logic* 21.4 (Oct. 1980), pp. 685–693. DOI: 10.1305/ndjfl/1093883253.

[13] Rémi Coulom. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 72–83. ISBN: 978-3-540-75538-8. DOI: 10.1007/978-3-540-75538-8_7.

[14] Haskell Brooks Curry. “Grundlagen der kombinatorischen Logik”. In: *American Journal of Mathematics* 52.3 (July 1930), pp. 509–536. DOI: 10.2307/2370619.

[15] Mariangiola Dezani-Ciancaglini and J.Roger Hindley. “Intersection types for combinatory logic”. In: *Theoretical Computer Science* 100.2 (1992), pp. 303–324. ISSN: 0304-3975. DOI: 10.1016/0304-3975(92)90306-Z.

[16] Boris Düdder, Moritz Martens, and Jakob Rehof. *Prototype Implementation of an Inhabitation Algorithm for FCL($\cap, \leq$)*. Presentation at Types 2011 in Bergen, Norway. Sept. 2011.

[17] Boris Düdder et al. “Bounded Combinatory Logic”. In: *Computer Science Logic (CSL’12) - 26th International Workshop/21st Annual Conference of the EACSL*. Ed. by Patrick Cégielski and Arnaud Durand. Vol. 16. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012, pp. 243–258. ISBN: 978-3-939897-42-2. DOI: 10.4230/LIPIcs.CSL.2012.243.

[18] Boris Düdder et al. “Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Composition Synthesis”. In: *ITRS 2012* (2012). DOI: 10.4204/EPTCS.121.2.

[19] Andrej Dudenhefner. "Algorithmic Aspects of Type-Based Program Synthesis". PhD thesis. Dortmund: TU Dortmund University, 2019. DOI: `10.17877/DE290R-20108`.

[20] Andrej Dudenhefner. *cls-fsharp*. 2019. URL: `https://github.com/mrhaandi/cls-fsharp`.

[21] Andrej Dudenhefner, Moritz Martens, and Jakob Rehof. "The Intersection Type Unification Problem". In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 19:1–19:16. ISBN: 978-3-95977-010-1. DOI: `10.4230/LIPIcs.FSCD.2016.19`.

[22] Andrej Dudenhefner, Christoph Stahl, and Jan Bessai. *Combinatory Logic Synthesizer with Predicates*. 2024. URL: `https://github.com/tudo-seal/clsp-python`.

[23] Andrej Dudenhefner et al. "Finite Combinatory Logic extended by a Boolean Query Language for Composition Synthesis". In: *29th International Conference on Types for Proofs and Programs TYPES 2023–Abstracts*. 2023, p. 105. URL: `https://types2023.webs.upv.es/TYPES2023.pdf#page=111`.

[24] Andrej Dudenhefner et al. "Finite Combinatory Logic with Predicates". In: *29th International Conference on Types for Proofs and Programs (TYPES 2023)*. Ed. by Delia Kesner, Eduardo Hermo Reyes, and Benno van den Berg. Vol. 303. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Aug. 2024, 2:1–2:22. ISBN: 978-3-95977-332-4. DOI: `10.4230/LIPIcs.TYPES.2023.2`.

[25] Jonas Duregård, Patrik Jansson, and Meng Wang. "Feat: functional enumeration of algebraic types". In: *SIGPLAN Not.* 47.12 (Sept. 2012), pp. 61–72. ISSN: 0362-1340. DOI: `10.1145/2430532.2364515`.

[26] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer, 1990. ISBN: 978-1-4684-0359-6. DOI: `10.1007/978-1-4684-0357-2`.

[27] Joyce Friedman. "Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. Summaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, NJ, 1960, pp. 3–50. 3a-45a." In: *The Journal of Symbolic Logic* 28.4 (1963), pp. 289–290. DOI: `10.2307/2271310`.

[28] Jean-Yves Girard. “Interprétation functionelle et élimination des coupures dans l’arithmétique d’ordre supérieure”. PhD thesis. Université Paris VII, 1972. URL: https://cs.cmu.edu/afs/cs.cmu.edu/user/kw/www/scans/girard72thesis.pdf.

[29] George Grätzer. *Universal Algebra*. New York: Springer, 1979. ISBN: 978-0-387-77486-2. DOI: 10.1007/978-0-387-77487-9.

[30] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. “Program Synthesis”. In: *Found. Trends Program. Lang.* 4.1-2 (2017), pp. 1–119. DOI: 10.1561/2500000010.

[31] Andrew Hunt and David Thomas. *The Pragmatic Programmer: Your Journey to Mastery*. Boston, MA, USA: Addison-Wesley, 1999. ISBN: 978-0135957059. URL: https://pragprog.com/titles/tpp20/the-pragmatic-programmer-20th-anniversary-edition/.

[32] Fadil Kallat, Tristan Schäfer, and Anna Vasileva. “CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories”. In: *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*. Ed. by Giselle Reis and Haniel Barbosa. Vol. 301. EPTCS. 2019, pp. 51–65. DOI: 10.4204/EPTCS.301.7.

[33] Fadil Kallat, Tristan Schäfer, and Anna Vasileva. “CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories”. In: *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*. Ed. by Giselle Reis and Haniel Barbosa. Vol. 301. EPTCS. 2019, pp. 51–65. DOI: 10.4204/EPTCS.301.7.

[34] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9.

[35] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2018. ISBN: 978-3-662-56039-6. DOI: 10.1007/978-3-662-56039-6.

[36] Burgun Luc, Greiner Alain, and Prado Lopes Eudes. “A consistent approach in logic synthesis for FPGA architectures”. In: *Proceedings of the 1st International Conference on ASIC (ASICON)*. 1994, pp. 104–107.

[37] Dominik Mäckel, Jan Winkels, and Christin Schumacher. “Synthesis of Scheduling Heuristics by Composition and Recombination”. In: *Optimization and Learning*. Ed. by Bernabé Dorronsoro et al. Cham: Springer International Publishing, 2021, pp. 283–293. ISBN: 978-3-030-85672-4. DOI: 10.1007/978-3-030-85672-4_21.

[38] Fernando C.N. Pereira and David H.D. Warren. “Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks”. In: *Artificial Intelligence* 13.3 (1980), pp. 231–278. ISSN: 0004-3702. DOI: 10.1016/0004-3702(80)90003-X.

[39] Jakob Rehof. “Towards Combinatory Logic Synthesis”. In: *BEAT’13 - 1st International Workshop on Behavioural Types* (2013).

[40] Jakob Rehof and Paweł Urzyczyn. “Finite Combinatory Logic with Intersection Types”. In: *Typed Lambda Calculi and Applications*. Ed. by Luke Ong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 169–183. ISBN: 978-3-642-21691-6. DOI: 10.1007/978-3-642-21691-6_15.

[41] John C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium*. Ed. by B. Robinet. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 408–425. ISBN: 978-3-540-37819-8.

[42] Moses Schönfinkel. “Über die Bausteine der mathematischen Logik”. In: *Mathematische Annalen* 92 (Sept. 1924), pp. 305–316. DOI: 10.1007/BF01448013.

[43] Raymond Merrill Smullyan. *To Mock a Mockingbird and Other Logic Puzzles: Including an Amazing Adventure in Combinatory Logic*. New York: Knopf, 1985. ISBN: 0-19-280142-2.

[44] Christoph Stahl, Felix Laarmann, and Andrej Dudenhefner. “Synthesis of Reactive Systems with Reachability and Safety Certificates”. In: *Leveraging Applications of Formal Methods, Verification and Validation. 12th International Symposium, ISoLA 2024, Crete, Greece, October 27-31, 2024, Proceedings*. Vol. 15219. Lecture Notes in Computer Science. (Submitted). Springer, 2025.

[45] Robin Sutherland et al. In: *Zeitschrift für wirtschaftlichen Fabrikbetrieb* 119.3 (2024), pp. 141–145. DOI: 10.1515/zwf-2024-1030.

[46] Anna Vasileva. “User Support for Software Development Technologies”. PhD thesis. Dortmund: TU Dortmund University, 2022. DOI: 10.17877/DE290R-23150.

[47] Hao Wang. “Logic of many-sorted theories”. In: *Journal of Symbolic Logic* 17.2 (June 1952), pp. 105–116. DOI: 10.2307/2266241.

[48] Jan Winkels. “Automatisierte Komposition und Konfiguration von Workflows zur Planung mittels kombinatorischer Logik”. PhD thesis. Dortmund: TU Dortmund University, 2019. DOI: 10.17877/DE290R-20469.

[49] Jan Winkels et al. “Synthesizing Structural Variants in Discrete-Event Simulation Models using Finite Combinatory Logic with Predicates”. In: *Software Engineering and Formal Methods: 23nd International Conference, SEFM 2025, Toledo, Spain, November 10-14, 2025, Proceedings.* (In preparation). Springer-Verlag, 2025.

[50] Wieslaw Zielonka. “Infinite games on finitely coloured graphs with applications to automata on infinite trees”. In: *Theoretical Computer Science* 200.1 (1998), pp. 135–183. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(98)00009-7.

Appendix A.

Benchmark Results

We will give the raw benchmark results of the figures for the benchmarks done in Chapter 6 and Chapter 7.

n	**FCL**	**FCLP**	n	**FCL**	**FCLP**
5	0.127	0.005	105	—	8.528
10	1.999	0.026	110	—	9.480
15	10.526	0.052	115	—	10.847
20	34.377	0.088	120	—	13.871
25	85.132	0.154	125	—	14.678
30	—	0.350	130	—	16.816
35	—	0.293	135	—	0.002
40	—	0.418	140	—	22.897
45	—	0.595	145	—	24.443
50	—	0.811	150	—	26.969
55	—	1.015	155	—	32.691
60	—	1.335	160	—	34.173
65	—	1.692	165	—	37.472
70	—	2.476	170	—	41.842
75	—	0.001	175	—	46.360
80	—	3.287	180	—	50.325
85	—	4.240	185	—	55.271
90	—	5.082	190	—	62.458
95	—	5.782			
100	—	6.789			

Table 4.: Raw benchmark results in seconds for Figures 5 and 6

n	**Term**	**Literal (V)**	**Term (C)**	n	**Term**	**Literal (V)**	**Term (C)**
5	0.005	0.013	0.004	105	1.332	0.649	0.156
10	0.010	0.033	0.010	110	1.478	0.707	0.147
15	0.024	0.056	0.015	115	1.640	0.743	0.188
20	0.039	0.078	0.025	120	1.817	0.786	0.184
25	0.058	0.100	0.028	125	2.029	0.826	0.201
30	0.081	0.126	0.040	130	2.211	0.893	0.179
35	0.111	0.150	0.040	135	2.480	0.949	0.190
40	0.148	0.183	0.046	140	2.728	1.013	0.200
45	0.188	0.214	0.052	145	2.955	1.051	0.205
50	0.238	0.244	0.060	150	3.199	1.088	0.254
55	0.293	0.277	0.066	155	3.463	1.152	0.257
60	0.351	0.308	0.073	160	3.814	1.198	0.282
65	0.421	0.337	0.080	165	4.115	1.293	0.242
70	0.520	0.370	0.087	170	4.453	1.341	0.256
75	0.593	0.418	0.095	175	4.757	1.352	0.304
80	0.707	0.451	0.104	180	5.174	1.416	0.312
85	0.808	0.490	0.124	185	5.578	1.506	0.286
90	0.895	0.500	0.127	190	6.039	1.547	0.333
95	1.034	0.562	0.142	195	6.466	1.637	0.339
100	1.173	0.604	0.149				

Table 5.: Raw benchmark results in seconds for Figure 7

n	**Literal (V)**	**Term (C)**	n	**Term (C)**	n	**Term (C)**
200	1.736	0.329	820	2.703	1440	8.326
220	2.034	0.381	840	2.750	1460	8.465
240	2.389	0.415	860	3.091	1480	8.571
260	2.765	0.461	880	3.226	1500	8.592
280	3.164	0.487	900	3.457	1520	9.016
300	3.704	0.552	920	3.498	1540	9.154
320	4.157	0.596	940	3.923	1560	9.806
340	4.671	0.662	960	3.891	1580	9.364
360	5.169	0.697	980	3.887	1600	9.273
380	5.638	0.759	1000	4.354	1620	10.343
400	6.280	0.827	1020	4.294	1640	10.204
420	6.712	0.921	1040	4.562	1660	9.992
440	7.342	0.953	1060	4.433	1680	10.109
460	8.380	1.019	1080	4.297	1700	10.819
480	8.708	1.138	1100	4.824	1720	10.593
500	9.735	1.176	1120	4.582	1740	11.192
520	—	1.326	1140	5.016	1760	11.540
540	—	1.348	1160	5.159	1780	11.822
560	—	1.463	1180	5.144	1800	12.169
580	—	1.599	1200	5.815	1820	12.485
600	—	1.725	1220	6.006	1840	13.338
620	—	1.714	1240	5.879	1860	13.881
640	—	1.924	1260	6.408	1880	12.972
660	—	1.872	1280	6.324	1900	13.363
680	—	1.917	1300	6.299	1920	13.771
700	—	1.980	1320	6.441	1940	14.321
720	—	2.162	1340	6.655	1960	14.169
740	—	2.220	1360	7.056	1980	15.379
760	—	2.357	1380	7.847	2000	14.442
780	—	2.546	1400	7.551		
800	—	2.734	1420	8.093		

Table 6.: Additional raw benchmark results in seconds for Figure 8

The figure uses the same data as Figure 7 for values < 200

n	**Infer Rev.**	**Infer**	**Cache**	**No Infer**
5	0.005	0.011	0.005	0.007
10	0.018	0.097	0.025	0.065
15	0.050	0.494	0.111	0.354
20	0.091	1.455	0.275	0.987
25	0.155	3.449	0.591	2.343
30	0.191	7.236	1.178	5.126
35	0.322	14.756	2.332	10.182
40	0.411	25.056	3.823	17.663
45	0.613	42.849	6.174	28.128
50	0.823	68.104	9.358	43.025
55	1.038	—	13.590	63.220
60	1.376	—	18.687	—
65	1.763	—	25.693	—
70	2.524	—	34.413	—
75	0.001	—	0.817	—
80	3.211	—	59.303	—
85	4.045	—	76.855	—
90	5.009	—	—	—
95	5.998	—	—	—

Table 7.: Raw benchmark results in seconds for Figure 9

n	**PyPy**	n	**PyPy**	n	**PyPy**
5	0.017	105	1.444	205	16.084
10	0.087	110	1.604	210	15.963
15	0.185	115	1.892	215	17.112
20	0.222	120	2.453	220	21.159
25	0.217	125	2.539	225	19.880
30	0.110	130	2.755	230	22.331
35	0.117	135	0.000	235	22.961
40	0.109	140	3.970	240	24.562
45	0.154	145	4.201	245	27.580
50	0.170	150	4.744	250	29.252
55	0.248	155	5.478	255	32.746
60	0.272	160	5.751	260	0.001
65	0.354	165	6.470	265	38.283
70	0.447	170	7.438	270	40.631
75	0.000	175	8.023	275	44.242
80	0.640	180	8.755	280	53.387
85	0.730	185	9.739	285	63.575
90	0.846	190	10.525		
95	0.987	195	12.102		
100	1.152	200	13.453		

Table 8.: Raw benchmark results in seconds for Figure 11

The values for the CPython interpreter are the same as the FCLP approach in Figure 5.

n	**single**	**2**	**4**	**8**	**16**
5	0.015	0.070	0.111	0.197	0.371
10	0.168	0.176	0.156	0.216	0.388
15	0.824	0.602	0.400	0.341	0.450
20	2.434	1.594	0.946	0.667	0.617
25	6.133	3.767	2.146	1.394	1.112
30	12.606	7.480	4.066	2.501	1.870
35	23.985	14.077	7.565	4.479	3.194
40	40.663	23.921	12.725	7.410	5.006
45	64.884	37.513	20.175	11.326	7.425
50	—	57.198	30.453	16.995	11.027
55	—	82.713	44.135	24.600	15.578
60	—	—	61.856	34.461	21.413
65	—	—	—	46.884	29.399
70	—	—	—	62.929	39.056
75	—	—	—	—	0.436
80	—	—	—	—	61.705

Table 9.: Raw benchmark results in seconds for Figure 12