User Support for Software Development Technologies

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

Anna Vasileva

Dortmund

2022

Dekan: Prof. Dr.-Ing. Gernot Fink
Gutachter:

Prof. Dr. Jakob Rehof (Technische Universität Dortmund, Deutschland)
JProf. Dr.-Ing. Ben Hermann (Technische Universität Dortmund, Deutschland)

Acknowledgments

First, I'd like to thank my advisor, Prof. Jakob Rehof, who was always a source of inspiration, suggestions, and support during my doctoral journey. I am very thankful for not only the job and encouragement but also the discussions, guidance, new ideas, and further training in type theory. Your enthusiasm for the topic was an inspiration during my diploma thesis almost eight years ago and a basis for my initial ideas. It has been an honour to be your PhD student.
Many thanks to Prof. Ben Hermann, who was so kind to support me during this last step on my PhD journey. I would also like to thank my other committee members, Prof. Falk Howar and Prof. Peter Buchholz.
I would like to thank Prof. Petra Mutzel and Dr. Doris Schmedding for their many tips, mental support, and encouragement as mentors. and Prof. Boris Düdder for believing in me at the beginning of my PhD.
Many thanks go to my former colleague Dr. Jan Bessai for sharing an office with me and for the great time during a summer school in Ohrid, Macedonia; numerous tips, discussions, understanding, and his patience until the end.
I would like to thank the entire team of the chair for Software Engineering, Ute Joscko, Sevda Tarkun, and Dr. Lars Hildebrand for their incredible support. Thanks especially to Fadil Kallat, Felix Laarmann, Tristan Schäfer and Jan Winkels for the practical application of the visualisation and debugging support, as well as for providing complex use cases for evaluating and improving this work. I would also like to thank Moritz Roidl for the discussions, the practical use cases in the field of cyber-physical systems, the introduction to a new research field, and for enabling me to stay in Dortmund and finish my PhD.
Finally, I want to thank my family in Bulgaria and my friends for their moral support throughout this experience.
Further, I thank my boyfriend Lutz for his great moral support and calmness when things were not going well. Thank you for being so patient.

Abstract

The adoption of software development technologies is very closely related to the topic of user support. This is especially true in early phases, when the users are not familiar with the modification or the build processes of the software that has to be developed nor with the technology used for software development. This work introduces an approach to improve the usability of software development technologies represented by the Combinatory Logic Synthesizer (CL)S Framework. (CL)S is based on a type inhabitation algorithm for the combinatory logic with intersection types and aims to automatically create software components from a domain-specified repository. The framework yields a complete enumeration of all inhabitants. The inhabitation results are computed in the form of tree grammars. Unfortunately, the underlying type system allows limited application of domain-specific knowledge. To compensate for this limit, this work provides a framework for debugging intersection type specifications and filtering inhabitation results using domain-specific constraints as main aspects. The aim of the debugger is to make potentially incomplete or erroneous input specifications and decisions of the inhabitation algorithm understandable for those who are not experts in the field of type theory. The combination of tree grammars and graph theory forms the foundation of a clear representation of the computed results that informs users about the search process of the algorithm. The graphical representations are based on hypergraphs that illustrate the inhabitation in a step-wise fashion. Within the scope of this work, three filtering algorithms were implemented and investigated. The filtering algorithm integrated into the framework for user support and used for the restriction of inhabitation results is practically feasible and represents a clear improvement compared to existing approaches. It is based on modifying the tree grammars resulting from the (CL)S Framework. Additionally, the usability of the (CL)S framework is supported by eight perspectives included in a web-based integrated development environment (IDE) that provides detailed graphical and textual information about the synthesis.

Zusammenfassung

Die Einführung von Softwareentwicklungstechnologien ist sehr eng mit dem Thema der "Benutzerunterstützung" verbunden. Dies gilt primär in frühen Phasen, wenn die Benutzer weder mit den Änderungs- oder Erstellungsprozessen der zu entwickelnden Software noch mit der für die Softwareentwicklung verwendeten Technologie vertraut sind. In dieser Arbeit wird ein Ansatz zur Verbesserung der Benutzerfreundlichkeit von Softwareentwicklungstechnologien am Beispiel von dem Combinatory Logic Synthesizer (CL)S Framework vorgestellt. (CL)S basiert auf einem Typinhabitationsalgorithmus für die kombinatorische Logik mit Intersektionstypen und zielt auf die automatische Komposition von Softwarekomponenten aus einem domänenspezifischen Repository ab. Das Framework liefert eine vollständige Aufzählung von aller Inhabitanten. Die Ergebnisse der Inhabitation werden in Form von Baumgrammatiken berechnet. Leider erlaubt das zugrundeliegende Typsystem eine begrenzte Anwendung von domänenspezifischem Wissen. Um dies zu kompensieren, bietet diese Arbeit als Hauptaspekte das Debuggen von Intersektionstypen und das Filtern von Inhabitationsergebnisse unter Verwendung von domänenspezifischen Constraints. Das Ziel des Debuggers ist es, potenziell unvollständige oder fehlerhafte Eingabespezifikationen und Entscheidungen des Inhabitationsalgorithmus für Nichtexperten auf dem Gebiet der Typentheorie verständlich zu machen. Die Kombination von Baumgrammatik und Grafentheorie bildet die Grundlage für eine übersichtliche Darstellung der berechneten Ergebnisse, die den Benutzer über den Suchprozess des Algorithmus informiert. Die grafischen Darstellungen basieren auf Hypergrafen, die das Inhabitationsprozess schrittweise veranschaulichen. Im Rahmen dieser Arbeit wurden drei Filterungsalgorithmen implementiert und untersucht. Der in das Framework zur Nutzerunterstützung integrierte Filteralgorithmus, der zur Einschränkung von Bewohnungsergebnissen verwendet wird, ist praktisch umsetzbar und stellt eine deutliche Verbesserung gegenüber bestehenden Ansätzen dar. Er basiert auf einer Modifikation der Baumgrammatiken, die durch das (CL)S Framework entstanden sind. Die Benutzerfreundlichkeit des (CL)S Frameworks wird zusätzlich durch acht Perspektiven unterstützt, die in einer webbasierten integrierten Entwicklungsumgebung (IDE[1]) enthalten sind und detaillierte grafische und textuelle Informationen über die Synthese liefern.

[1] von englisch: integrated development environment

Contents

List of Figures

Chapter 1

Introduction

Modern technologies are being developed with increasing rapidity. Smart homes have become a feature of people's daily routines, and the technology for self-driving cars is already over a decade old. Notably, people's mobility continues to change, as evidenced by autonomous robots, fly-by-wire, and drive-by-wire technologies of *Industry 4.0* and smart cities [25; 118]. Related projects aim to support the interactions between people and technologies to improve efficiency and productivity. Accordingly, technologies shape people's daily (social) routines. However, technologies are pointless without users. For this reason, the acceptance and adoption of these technologies depends on their usability. The more straightforward the tools, the more advantageous they are for users.

The Combinatory Logic Synthesizer (CL)S is a synthesis framework based on the combinatory logic with intersection types [31; 103]. A type inhabitation algorithm searches for inhabitants based on a given repository and target type. The repository includes domain-specific, typed combinators. Notably, software synthesis deals not only with the automatic composition of classic software (e.g., executable software) but also with the synthesis of data structures such as LegoNXT scripts [33], Microsoft Project schedules, process plans [133], or Business Process Model and Notation (BPMN) 2.0 diagrams [31; 109]. Unfortunately, despite the broad range of possible applications, the usability of software synthesis frameworks has not been prioritised.

This work introduces a web-based integrated development environment (IDE) as user support for the component-based synthesis framework – (CL)S. The topic *usability* is familiar to the field of software development [99]. ISO 9124 [81] defines usability as follows: "the extent to which a system, product or service can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". In this case, the system is the (CL)S IDE Framework, and the target audience includes not only software developers who may have limited training in the field of type theories but also users with programming knowledge and an interest in the advantages of frameworks for software synthesis. The software synthesis with user support increases the efficiency and effectiveness of the

development process without the inexplicable (non-)solutions of the inhabitation algorithm. The basis of research in the field of user support for software development technologies is presented by the debugging of specifications for intersection types [51] and a filtering function of a complete enumeration of inhabitants based on modification of tree grammars. To support the users, this work introduces the main features of an IDE that augments the functionalities of the (CL)S, helps one understand the decisions of the algorithm, and supports the developers when their input specifications are incomplete or incorrect. The specification of the components and the variability of the synthesis software can be most troublesome. For this reason, debugging and filtering make such troubles more manageable. In this way, the (CL)S IDE Framework supports users during synthesis and assists with the discovery and comprehension of solutions. The debugging function informs users of bad specifications or errors that occur during the search process, so the filtering algorithm supports users in achieving accurate inhabitation results according to the domain specifications by reducing the synthesised solution space. The wide field of application areas of the (CL)S Framework requires use-case independence, so this independence had high priority during the development of the IDE. This work identifies the basic features required to explain the search process and detect input-specifications deficiencies. To our knowledge, no other framework currently supports the debugging of specifications for intersection types [51] nor provides a filtering support based on a modification of tree grammars.

1.1 Related Work

The automated system for verifying and synthesising programs, Leon [41], also provides a web-based IDE [18] for the user support. This framework is intended for the application of functional Scala programs. The verification rests on external satisfiability modulo theory (SMT) solvers whose installation varies by operating system, and the preformation is based on the Scala AST nodes. The synthesis is based on a given set of data structures with invariants. The (CL)S IDE provides a backend synthesis algorithm for the automatic composition of software components. The synthesis comes from type specifications that are independent of the Scala programming language. Moreover, the representation of the results is graph-oriented and a central feature of the development of the user support. In contrast, the Leon IDE provides text informing users of the results for each line of code. The graphical representation supplies an annotation of each node representing a combinator with source positions to indicate the point of origin of that combinator in the repository. Similar to Leon, the platform for program verification Why3 [42] is also based on external provers and provides an IDE with a textual output of the solvers for a certain position in the code. Certain theorem provers also have IDEs for better usability to inform for program verification: for example, the proof assistants Coq [19; 84], Isabelle [5], and LEAN [63], in which different colours indicate different information. Numerous examples of programs with IDE exist in the area of automatic verification. The examples presented here derive from web-based solutions also aiming to support users. The main difference from most web-based IDEs is the location and manipulation of the input specification, particularly using the browser. In our case, the implementation extends the inhabitation algorithm and works with the results and not with the user-specified repository, such that the developer can apply the local IDE they usually use. The main reason for the development of a web-based application is its platform independence. This approach reduces the effort of program setup without the installation of a specific framework or version. Examples of such IDE tied to a platform are JavaFX [100] and Oracle Java [89]. Numerous approaches deal with the topic of program synthesis using type theory, for instance, those of Polikarpova et al. [101], Zdancewic et al. [66], and Kuncak et al. [75]. To our knowledge, these approaches for program synthesis provide no user support at the same time.

Globular [22] offers an example of a web-based proof assistant with a graphical representation of the results. In contrast to the (CL)S IDE, users can manipulate graphs based on sting diagrams. The disadvantage of this approach is that the user must be expert in category theory. At the same time, the users of the (CL)S IDE can also be nonexperts in background theory. In the field of software synthesis, Feng et al. [64] present a graphical representation using Petri nets. They synthesise programs based on methods from abstract programming interfaces (APIs), where the nodes of the Petri nets correspond to types and where the transitions are API calls. This approach can be viewed as an analysis of a graph-reachability problem. In this case, Petri nets are well suited to the synthesis of imperative programs, which requires multiple calls of a certain procedure. The authors discuss the application of *hypergraphs* on an approach similar to that presented in [30] and they indicate the approach to be complex. Bessai [29]

explains in his work why this is not the case and how the problems can be avoided using synthesis based on a semantic type concept. Moreover, the representation of Petri nets by another kind of graphs is not novel in the literature. The authors of [87], [88], and [98] discuss such approaches for transformation. In the context of this work, for the representation of the synthesis results, a simple multi-hypergraph is sufficient, such that the additional features of the Petri nets are unnecessary.

Numerous approaches with different focal points are based on SMT. Very often, this technology is used for the synthesis [117; 72; 15]. The definition of syntactic constraints ensures certain correctness specifications. Solar-Lezama et al. [117] introduce the language for finite programs, *Sketch*. The sketching approach used for software synthesis completes sketches of programs using constraints, which consider certain domain-specified functionalities [116]. These specifications ensure the generation of solutions representing only desired components and are verified by a solver. The approach presented by Gulwani, Singh, and Polozov [74] also represents a synthesis using SMT. The users define the desired functionalities by examples (e.g., input-output specifications, demonstrations, or assertions), and a constraint solver verifies whether the programs satisfy. Other approaches dealing with SMT synthesis are introduced in [83; 123; 107; 66; 73]. In contrast to these approaches, the filtering approach presented in this work is based on already-synthesised results using combinatory logic with intersection types. The results detected by a user as trivial or undesirable can afterwards be filtered out using constraints. An alternative to the SMT approach is that of Jan Winkels [132]. He applied ScalaGraph [12] to filter the results of a synthesis of planning processes according to user-defined constraints. An investigation presented in this work demonstrates that this approach is proper to this application case but cannot be provided as domain-independent filtering in the IDE.

The restriction approach presented by Michael D. Adams and Matthew Might [14] is closely related to the filtering implementation presented in this work. Their work deals with ambiguities caused in programming languages such as YACC or C. Figure 1.1 illustrates the approach, whereby ambiguities caused by precedence, associativity, dangling `else`, and functional `if` can be managed. They show how restrictions and context-free grammars (CFGs) can be encoded using tree automata.

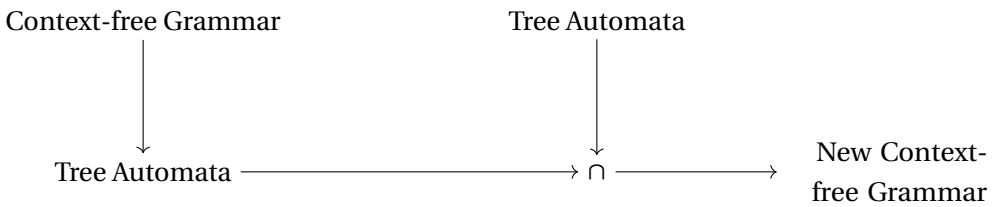

Figure 1.1: Overview of the filtering approach by Adams and Might [14]

An intersection of CFGs with restrictions makes the approach computable because intersec-

tion and difference are undecidable on CFGs. The result represents a tree automaton that considers user-defined constraints. They translate the result into context-free grammar by removing the production labels. Similar to the filtering approach presented in this work, the restrictions represented as tree automaton accept any trees except those that must be restricted. In this work, the application of intersection of the languages of tree grammars is presented as a filtering approach. CFGs and tree grammars are strongly related [50], and the intersection and difference problems between two tree grammars, in turn, are decidable [50].

The problem of enumeration terms resulting from a tree grammar has been investigated by Bessai et al. [39]. The authors have shown that the deciding emptiness and finiteness of the intersection $\mathscr{L}(\mathscr{G}) \cap NF(R)$ is EXPTIME-complete, where $\mathscr{L}(\mathscr{G})$ is the language of a regular grammar $\mathscr{G}$ and $NF(R)$ is the normal form of a rewriting system R. The approach is based on previous work by Comon and Jacquemard [50; 49] on automata with disequality constraints and pumping lemma. This approach applies to linear and nonlinear rewrite relations. It is demonstrated that the linear case is practically feasible in contrast to the nonlinear case. A prototype Haskell implementation of the algorithm is introduced and experimentally analysed. The investigation has shown that the performance of the linear case is better than in the nonlinear case. According to the authors, this approach represents an improvement of a filtering algorithm based on an SMT solver [85] presented and discussed in this work (cf. Section 3.1). At the same time, the filtering approach based on the modification of tree grammars that is introduced in Section 3.3 and is comparable with the linear case performs faster for the same use cases (cf. Section 5). These presented results are significant for the investigations introduced in this work. The practical application and the comparison of these algorithms are discussed in Section 3.3.3 and Section 5.1.

1.2 Publications

Within the scope of this dissertation project, results in the field of formal IDEs and filtering based on theorem prover are already published. We consider these publications in this section. The published results are detailed in this work and referenced accordingly.

1.2.1 User Support for the Combinator Logic Synthesizer Framework

The first publication to present a work in progress on a debugger for the (CL)S Framework was [30]. The authors outline the idea of user support for type-based tools. This paper presents the motivation and explains the question of why it is important to consider usability for the adoption of software development technologies. This publication presents a visualisation approach using hypergraphs to make the results of the inhabitation algorithm understandable for nonexperts. The presentation of different perspectives and features completes the introduction of the IDE.

1.2.2 CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories

In this paper [85], a framework combining SMT solvers and the (CL)S Framework was presented. This work aims to consider the advantages of these theories to restrict certain solutions computed by the inhabitation algorithm. Such a filtering approach is necessary because, in some situations, certain well-formed terms might be considered unsuitable results according to the context of the application case. Moreover, certain terms might also represent trivial or identical results and behaviour. The filtering approach centres on the definition of domain-specific constraints and translation of the resulting tree grammars into SMT functions. The translation contains all computed terms that comply with a given target type. The SMT solver decides which terms are satisfied and which are not. That way, solutions can be filtered out according to given domain-specific constraints.

1.2.3 Experience Report: Towards Moving Things with Types – Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis

Another publication within the scope of this dissertation was [38]. This work presents the results of an experience report based on interdisciplinary collaborative work at the Technical University of Dortmund, regarding the type-based synthesis algorithm, (CL)S, and its IDE with a logistics lab environment. The possibilities, advantages, and challenges of applying the framework in the domain of logistics are outlined and discussed. Moreover, the results show that an extension of the framework is possible with manageable effort. For example, network communication was implemented to connect the (CL)S Framework with the infrastructure in the research lab. The experiment and its lessons are detailed in this work.

1.2.4 Clean Code and Static Code Analysis

Publications in the field of clean code and static code analysis were also published by the author. They do not directly impact the results presented in this work, but they support the development of the approach and the research in the field of usability. The insights gained regarding maintainable code, namely avoidance of dead code and use of clear specifications [93], motivated the development of features included in the IDE for user support. These publications are not elaborated, but are referenced accordingly. The following research papers were published:

- **Clean Code – ein neues Ziel im Software-Praktikum** [1].
  This paper [112] presents the topic of *Static Code Analysis* and considers the importance

[1] English: "Clean Code – a new goal in the software practicum"

of high-quality code for development teams. Furthermore, the authors introduce an approach for long-term and effective integration of code-quality assurances in the software development process in student projects. The results stated here are achieved by more iterations in the software education process, as well as by didactic methods.

- **Integration von Qualitätsaspekten in einen Entwicklungsprozess**[2].
  The authors [110] introduce the results of an effective integration of *clean code* assurances in development processes. The presented findings are from an early phase of the integration process. However, successful integration of such a topic proceeds in several iteration steps. The authors discuss the acceptance of the applied metrics and thresholds, as well as their dependency on the projects' complexity.

- **Vom Clean Model zum Clean Code**[3].
  This study's [126] results correlate the quality of UML models with the quality of the developed code in software development processes in computer science education. Here, the authors discuss which of the deficiencies visible in the code can be recognised in the model: for instance, regarding naming conventions, operations with long parameter lists, *God classes* [93], or long functions and classes. Complex changes lead to the formation of chain reactions, which beginning programmers struggle with. Facing such, student motivation suffers. This demotivation should be avoided. For this reason, quality aspects must already be accounted in the modelling phase. Defects detected early can be eliminated more easily and with less effort than for defects detected too late.

- **How to Improve Code Quality by Measurement and Refactoring.**
  In this paper [127], the authors demonstrate the successful integration of measurement of code quality in the process of software development. Apart from choosing a suitable tool for code analysis and metrics, alongside proper threshold values, concepts to remove deficiencies are essential code quality requirements. In addition, several cycles of a development process are necessary to achieve a long-term and effective integration of code quality in the development process. The integration of static analysis and refactoring of program code is achieved through the *Plan-Do-Check-Act* cycle and with didactic methods in a software development course at the university.

- **Clean Java - Von Anfang an!** [4].
  This article [125]examines developers' awareness of high-quality code and the integration of code-quality aspects in the software development process in computer science education. The authors show that high motivation, clear goals, and appropriately chosen tools for static code analysis, metrics, and benchmarks are crucial in successfully anchoring clean code in the development process.

[2] English: "Integration of quality aspects into a development process"
[3] English: "From Clean Model to Clean Code"
[4] English: "Clean Java - Right from the start!"

- **Reviews – ein Instrument zur Qualitätsverbesserung von UML-Diagrammen** [5].
  This research paper [111] conceptualises how to improve the quality of model-based development. In the first phase of the analysis, the students must specify and document the software requirements using UML models. Unfortunately, the quality of these models is often very poor. The approach is based on peer reviews of the quality of models in the development teams. The reviews are based on quality aspects inspired by clean code. They appear in the form of checklists. This approach allows for careful consideration of the diagrams and for more profound discussions of the quality achieved.

1.3 Overview

This work is organized as follows: In the following, Chapter 2 discusses the relevant theoretical background. Here, the synthesis based on finite combinatory logic with intersection types and the resulting tree grammars are introduced. Moreover, the visualisation technique of tree grammars applied in this work is explained in this part. Chapter 3 outlines an essential contribution: the developed filtering approaches. This part presents the different algorithms and provides information about the advantages, disadvantages and the formal correctness of the algorithm integrated into the web-based IDE for the (CL)S Framework. Chapter 4 deals with the IDE for the (CL)S Framework. Detailed information about the implementation and the features of each perspective are presented in this part. Chapter 5 gives an overview over the evaluation and analysis of the performance of the filtering algorithms. Furthermore, a comparison to other filtering techniques is discussed. Chapter 6 outlines investigations within real development projects, presenting the advantages of the IDE, its importance for the support of developers, and lessons learned from the application cases. Finally, Chapter 7 summarises the main conclusions and discusses limitations to the results and the recommendations for further research.

[5] English: "Reviews - a tool for improving the quality of UML diagrams"

Chapter 2

Theoretical Background

This chapter introduces combinatory logic synthesis (Section 2.1). The leading theory behind the software synthesis approach presented in this work, namely finite combinatory logic with intersection types, is considered in Section 2.2, while Sections 2.3 and 2.4 elaborate tree grammars, an essential frame for this work's developments. Section 2.5 introduces the framework for composition synthesis, (CL)S Scala, which produces the tree grammars, the basis for the visualisation and filtering approaches (see Chapter 3) introduced in this work. Section 2.6 represents the theory behind the visualisation of the inhabitation results. Here, the definition, usage, and the limitations of the hypergraphs as a visualisation technique for tree grammars are discussed. The last section deals with Satisfiability Modulo Theories (SMT) that is an essential part of the filtering approach presented in Section 3.1.

2.1 Combinatory Logic Synthesis

Nowadays, software synthesis is familiar in the study of the composition of large systems. It is an approach that aims for automatically generate programs based on user specifications. Synthesis based on combinatory logic is a type-theoretical approach dealing with typed combinators. The combinators represent the user-specified software components. This approach reduces the search space supporting the handling of the complexity of software synthesis problems.

The principle of combinatory logic was first presented by Moses Schönfinkel in his work "Über die Bausteine der mathematischen Logik" [1] in 1924 [113]. This approach is older than Lambda Calculus [48]. Both techniques are equally expressive, but in contrast to Lambda Calculus, the grammar of combinatory logic is much simpler, since it does not contain bound variables. A few years later, Haskell Curry turned the idea of combinatory logic into a useful programming technique. However, in general, these systems aim to describe the general properties of programs, using these to modify other programs [78].

To describe the combinatory logic synthesis, the theory of inhabitation based on combinatory logic is introduced in the following. The decision problem of inhabitation is defined as follows:

$$\Gamma \vdash_s ? : A$$

That is, the decision problem asks for a term e in a fixed set Γ of typed combinators, with a given type A. We call a term e inhabitant with type A if $\Gamma \vdash_s e : A$ is true. The above-presented generalisation of the inhabitation problem is defined as follows: Given Γ and A, does there exist a combinatory term e with $\Gamma \vdash e : A$? [103]

In combinatory logic with simple types, each term is assigned exactly one type.

Definition 1 (Simple Types) Let $\mathbb{T}$ be a non-empty set of simple types. Simple types $A, B \in \mathbb{T}$ are defined as follows:

$$A, B ::= \alpha \mid a \mid A \to B,$$

where $\alpha \in \mathbb{V}$ is a type variable, $a \in \mathbf{B}$ is a constant, and $A \to B$ is a function type. □

Terms are defined by

$$e ::= X \mid (ee),$$

where X denotes a combinator symbol that ranges over a set $\mathbf{B}$ and where (ee) represents an application of term e to term e. In typed combinatory logic, we have a fixed set Γ that represents the typed environment and contains components of the form $(X : A)$. The combinator base contains the following Turing-complete standard combinators:

[1] English: "On the Building Blocks of Mathematical Logic"

$$\Gamma = \{\mathbf{S} : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$$
$$\mathbf{K} : (\alpha \to \beta \to \alpha)\}$$

It results in a system corresponding to Hilbert–Style systems of logic [103]. The standard combinatory logic with simple types is based on the following type rules:

- The (var) rule allows substitution of formulas into axioms. Here, S denotes a type substitution on $\Gamma(X)$.

$$\frac{}{\Gamma, X : A \vdash X : S(A)}\ (\mathsf{var})$$

- The *modus ponens* rule ($\to \mathsf{E}$) allows the application of the component specifications from a set Γ.

$$\frac{\Gamma \vdash e : A \to B \qquad \Gamma \vdash e' : A}{\Gamma \vdash (ee') : B}\ (\to \mathsf{E})$$

In combinatory logic with intersection types, terms can also be assigned to an intersection of types [22]. For combinatory logic with intersection types and fixed base **S**, **K**, the inhabitation problem is undecidable [103].

Definition 2 (Intersection Types) Intersection types, denoted by $\sigma, \tau \in \mathbb{T}$, are defined as follows:

$$\sigma, \tau ::= \omega \mid a \mid \alpha \mid \sigma \to \tau \mid \sigma \cap \tau$$

Type constants are represented by a and range over a set **B**. Type variables $\alpha, \beta, \cdots \in \mathbb{V}$ can be substituted with type constants. Furthermore, intersection types can be constructed from function types ($\sigma \to \tau$) as well as from intersection ($\sigma \cap \tau$). □

Aside from the intersection types, we consider the subtype relation ($\leq$) presented by Barendregt, Coppo, and Dezani-Ciancaglini (BCD) [23]. The subtype relation $\sigma \leq \tau$ expresses that type σ is a subtype of the type τ. This relation complicates the type system because the values of type σ can always be used instead of the values of type τ. We define the following subtyping rule as an extension of the original BCD rules [29].

Definition 3 (Subtyping Rules) The relation $\leq$ is closed under the following rules:

- The (IDEM) rule controls the idempotency.

$$\frac{}{\sigma \leq \sigma \cap \sigma}\ (\textsc{Idem})$$

- The ($\leq \omega$) rule allows the usage of a special type constant ω as the universal type.

$$\frac{}{\sigma \leq \omega} \ (\leq \omega)$$

- The $(\rightarrow \omega)$ rule controls the functions with type ω.

$$\frac{}{\omega \leq \omega \rightarrow \omega} \ (\rightarrow \omega)$$

- The (SUB) rule allows co- ($\tau_1 \leq \tau_2$) and contra-variant ($\sigma_2 \leq \sigma_1$) subtyping of functions. This rule introduces subtype polymorphism. A function type $\sigma_2 \rightarrow \tau_2$ is a supertype of any function type $\sigma_1 \rightarrow \tau_1$ if $\sigma_2 \leq \sigma_1$ and $\tau_1 \leq \tau_2$.

$$\frac{\sigma_2 \leq \sigma_1 \qquad \tau_1 \leq \tau_2}{\sigma_1 \rightarrow \tau_1 \leq \sigma_2 \rightarrow \tau_2} \ (\text{SUB})$$

- The (GLB) rule allows the usage of type σ instead of the intersection $\tau_1 \cap \tau_2$ if σ is a subtype of τ_1 and τ_2.

$$\frac{\sigma \leq \tau_1 \qquad \sigma \leq \tau_2}{\sigma \leq \tau_1 \cap \tau_2} \ (\text{GLB})$$

- The rules (LUB_1) and (LUB_2) allow for the intersection as the least upper bound. In other words, an intersection of the two types is a subtype of individual types.

$$\frac{}{\sigma \cap \tau \leq \sigma} \ (\text{LUB}_1) \qquad \frac{}{\sigma \cap \tau \leq \tau} \ (\text{LUB}_2)$$

- The rule (DIST) allows the distribution of intersections.

$$\frac{}{(\sigma \rightarrow \tau_1) \cap (\sigma \rightarrow \tau_2) \leq \sigma \rightarrow \tau_1 \cap \tau_2} \ (\text{DIST})$$

Using the rules above, the following distribution with intersection types $(\sigma_1 \rightarrow \tau_1) \cap (\sigma_2 \rightarrow \tau_2) \leq \sigma_1 \cap \sigma_2 \rightarrow \tau_1 \cap \tau_2$ can be derived.

2.2 Finite Combinatory Logic with Intersection Types

In this section, we discuss the main theory behind the software synthesis considered in this work: finite combinatory logic with intersection types, as studied by Rehof and Urzyczyn [104]. They characterise the decision problem as EXPTIME-complete. This system derives from combinatory logic with simple types [78], applied with the following two extensions:

- the provability question in a Hilbert–Style with a generalised form of the combinator base and
- BCD intersection types [23].

The synthesis problem asks whether there is a composition of functions from a set Γ of typed combinators with a given type.

Definition 4 (Applicative Terms) Let **B** be a finite set of combinators. Applicative terms $M, N \in \mathbb{A}$ are defined as follows:

$$M, N ::= c \mid (MN),$$

where $c \in \mathbf{B}$. A term $M \in \mathbb{A}$ can be constructed using named component or combinator c and application of M to N. An application is left associative, such that $((MN)P) \equiv (MNP)$ with $P \in \mathbb{A}$. □

Constants can also represent user specifications. Furthermore, types can encode functions. The user specification of the combinators is Turing-complete in general [58]. Intersection $(\sigma \cap \tau)$ in user-defined components in a repository Γ can also be used for the construction of results. This intersection is useful for representing the combination of semantic and native types. In other words, a combinator may be typed under two perspectives: a technical one for an underlying programming language and a user-centric one. Terms can also be assigned to an intersection of types [103]. Native types correspond to the type of implementation of the actual domain-specific components. For example, if the native type is `String`, the algorithm will search for any applicative term made from combinators that can be assigned to that type in the underlying programming language. Moreover, a semantical specification of the components can be applied. In this way, the type of combinators can be specified to make the synthesis more precise, according to the intended usage. Thus, it is possible to combine native types with semantic types. For example, if we have in Γ a combinator c : `String` $\cap$ $Title$, the `String` represented by this combinator can also be used as a $Title$ for something.

In finite combinatory logic with intersection types, the type inhabitation process is based on rules that assign types to combinatory terms [58]. These rules are presented in the following.

- The (var) rule controls the application of any combinator c from the typed repository Γ that has type τ using substitutions. It is defined as follows:

$$\frac{}{\Gamma, c : \tau \vdash c : \Gamma(\tau)}\ (\mathsf{var})$$

  As mentioned, the specification mechanism is Turing complete, and the type inhabitation problem is in general undecidable. For this reason, a restriction on variable substitution is needed to ensure decidability [58].

- Combinators can also have function types. The arrow elimination rule ($\rightarrow$ E) allows the application of such combinators to appropriately typed arguments to form new terms from another terms.

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}\ (\rightarrow \mathsf{E})$$

Here, we consider M as a function that takes σ as input type and returns type τ.

- Finite combinatory logic with intersection types is also closed under the subtyping rules based on BCD [23] and presented in Definition 3. Here, the BCD system is furthermore extended with type constructors, as was proposed in [90; 37]. The subtyping rule ($\leq$) is defined as follows:

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

In FCL, the following rules are derivable [29]:

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \ (\cap\mathrm{I}) \qquad\qquad \frac{}{\Gamma, M : \omega} \ (\mathbb{A}\omega)$$

2.3 Tree Grammars

In [29], an inhabitation machine is defined, which given a repository and requested type will compute a tree grammar. The soundness and completeness of the implemented inhabitation machine are proven, ensuring computed *tree grammars* are finite and exact representations of all possible solutions. According to [50], the emptiness of tree grammars is decidable in linear time, so this method also solves the type emptiness problem. The following definition of tree grammars is an unranked version of the normalised regular tree grammars defined in [50]. Compared to [29], it will encode a more compactly readable, fully uncurried version of combinator applications, where ((c M) N) becomes c(M, N).

Definition 5 (Tree Grammars) A tree grammar $\mathscr{G}$ is a 4-tuple $(S, \mathscr{N}, \mathscr{F}, R)$ with

- a start symbol $S \in \mathscr{N}$,
- a set $\mathscr{N}$ of *nonterminals*,
- a set $\mathscr{F}$ of *terminal symbols*, and
- a set R of *productions rules* of the form $\alpha_1 \mapsto \{c_1(\beta_1, \beta_2, \ldots \beta_n),\ c_2(\gamma_1, \gamma_2, \ldots \gamma_m)\}$, where $n, m \geq 0$, $\alpha_1, \beta_1, \beta_2, \ldots, \beta_n, \gamma_1, \gamma_2, \ldots, \gamma_m \in \mathscr{N}$ are nonterminal, and $c_1, c_2 \in \mathscr{F}$ are terminal symbols.

We consider unranked tree grammars without restriction on the arity of the terminal symbols; for example, we can have rules r_1 and $r_2 \in R$, such that $r_1 := \alpha_1 \mapsto c(\beta_1, \ldots, \beta_n)$ and $r_2 := \alpha_2 \mapsto c(\beta_1, \ldots, \beta_m)$, where $n, m \in \mathbb{N}$ and $n \neq m$, $c \in \mathscr{F}$, and $\alpha_1, \alpha_2 \in \mathscr{N}$. Moreover, we allow rules with arity = 0, which we abbreviate to $r := \alpha_1 \mapsto c()$. □

Definition 6 (Tree Grammar Languages) For a given tree grammar $\mathcal{G} = (\alpha, \mathcal{N}, \mathcal{F}, R)$ and target symbol $\alpha \in \mathcal{N}$, $\mathcal{L}_\alpha(\mathcal{G})$ is the least set closed under the following rules:

- if $\alpha \mapsto c \in R$ then $c \in \mathcal{L}_\alpha(\mathcal{G})$.
- if $\alpha \mapsto c(\beta_1, \beta_2, \ldots, \beta_n) \in R$ and for all $1 \leq k \leq n : t_k \in \mathcal{L}_{\beta_k}(\mathcal{G})$ then $c(t_1, t_2, \ldots, t_n) \in \mathcal{L}_\alpha(\mathcal{G})$

We define $\mathcal{L}(\mathcal{G}) = \mathcal{L}_\alpha(\mathcal{G})$ to be the language of grammar $\mathcal{G}$ for start symbol α. □

Definition 7 (Applicative Tree Grammars) An applicative tree grammar $\mathcal{G}_{ap}$ is a special case of tree grammars with

- a start symbol $S \in \mathcal{N}$
- a set $\mathcal{N}$ of *nonterminals*,
- a set $\mathcal{F}$ of *terminal symbols*,
- a set R of *productions rules* of form $\alpha_1 \mapsto c$ and $\alpha_2 \mapsto @(\beta_1, \beta_2)$, where $\alpha_1, \alpha_2, \beta_1, \beta_2 \in \mathcal{N}$ are nonterminal and $c \in \mathcal{F}$ is a terminal symbol. The operator @ is a binary operator $@ : R \times R \to R$.

We consider applicative tree grammars with restrictions on the arity of the terminal symbols; for instance, we can have application rules only with arity = 2, such that $r_1 := \alpha \mapsto @(\beta_1, \beta_2)$, where $r_1 \in R$ and $\alpha \in \mathcal{N}$, or rules with arity = 0 abbreviated to $r_2 := \alpha \mapsto c$, where $r_2 \in R$ and $c \in \mathcal{F}$. □

Definition 8 (Applicative Tree Grammar Languages) For a given applicative tree grammar $\mathcal{G}_{ap} = (\alpha, \mathcal{N}, \mathcal{F}, R)$ and target symbol $\alpha \in \mathcal{N}$, $\mathcal{L}_\alpha(\mathcal{G}_{ap})$ is the least set closed under the following rules:

- If $\alpha \mapsto c \in R$ then $c \in \mathcal{L}_\alpha(\mathcal{G}_{ap})$.
- If $\alpha \mapsto @(\beta_1, \beta_2) \in R$, $t_1 \in \mathcal{L}_{\beta_1}(\mathcal{G}_{ap})$, and $t_2 \in \mathcal{L}_{\beta_2}(\mathcal{G}_{ap})$ then $@(t_1, t_2) \in \mathcal{L}_\alpha(\mathcal{G}_{ap})$

We define $\mathcal{L}(\mathcal{G}_{ap}) = \mathcal{L}_\alpha(\mathcal{G}_{ap})$ as the language of grammar $\mathcal{G}_{ap}$ for start symbol α. □

A word $e \in \mathcal{L}_\alpha(\mathcal{G})$ for a target type $\alpha \in \mathcal{N}$ is an inhabitant e of type α, where $\mathcal{G}$ is tree grammar or applicative tree grammar. Applicative tree grammars represent terms as a curried version. Hence, in contrast to tree grammars, terms are represented by @(@(c,β_1),β_2) instead of c(β_1, β_2). The next section (Section 2.4) demonstrates that certain tree grammars and applicative tree grammars are mutually translatable.

2.4 Translation of Tree Grammars

Ranked and unranked trees are well-studied constructions that can be recognised by the same finite automata [50; 45; 69; 55]. Unranked trees are often encoded by ranked because of the application of the theory of ranked tree automata on unranked trees. In this manner, already existing algorithms can be reused. A ranked (or applicative) tree grammar can be considered the unranked tree grammar's binary presentation.

In the following, we discuss tree grammars resulting from the inhabitation machine. We show that the tree grammars and the applicative tree grammars can be translated into each other. That is, they generate equivalent languages so that they are interchangeable. Applicative tree grammars correspond to tree grammars regarding function $f_{at} : \mathscr{G}_{ap} \to \mathscr{G}$ (s. Algorithm 1). In contrast, tree grammars correspond to applicative tree grammars regarding the function $f_{ta} : \mathscr{G} \to \mathscr{G}_{ap}$ presented in Algorithm 2, which is based on the extension encoding presented in [50], Chapter 8.

Proposition 1 *Each applicative tree grammar generated by the inhabitation machine can be translated into a tree grammar.* □

Given an applicative tree grammar $\mathscr{G}_{ap} = (\tau, \mathscr{N}, \mathscr{F}, R)$, we can translate each production rule into an applicative tree grammar using algorithm T_{at} presented below (Algorithm 1). Here, the input is an applicative tree grammar represented by a set of rules of the form presented in Definition 7. The output is the encoded tree grammar. The algorithm encodes each binary rule into a production rule. Rules of the form $\alpha \mapsto c$ are translated in $\alpha \mapsto c()$ and added to the set with the production rules in line 4. Application rules are recursively handled by function COMPUTE_PRODUCTION_RULES. If we have in the applicative tree grammar rules of the form $\alpha \mapsto @(\beta_1, \beta_2)$, $\beta_1 \mapsto @(\sigma, \tau)$, $\beta_2 \mapsto y$, $\sigma \mapsto c$, $\tau \mapsto x$, where $\alpha, \beta_1, \beta_2, \sigma, \tau$ are nonterminal and c, x, y are combinators, we get the following production rules $\alpha \mapsto c(\tau, \beta_2), \tau \mapsto x$ and $\beta_2 \mapsto y$. In contrast to the first algorithm, this algorithm satisfies for applicative tree grammars, where the left branch of the tree cannot have an arbitrary height. Such a case can occur when we have a combinator with the universal type of anything ω so that the subtype rule (cf. Definition 3) produces arrows from any such target and we get $\omega \leq \omega \to \omega \leq \omega \to \omega \to \omega \leq \ldots$. In this case, the combinator can receive an arbitrarily high number of arguments. To handle this case, we insert a new rule of the form $\alpha \mapsto *()$ (line 19). Hence, from a node $*$, any number of trees is created. In this way, the algorithm encodes any applicative tree grammar to a tree grammar, in this case $T_{at}(\mathscr{G}_{ap}) = \mathscr{G}$.

Proposition 2 *Each tree grammar can be translated into an applicative tree grammar.* □

Given a tree grammar $\mathscr{G} = (\tau, \mathscr{N}, \mathscr{F}, R)$, we can translate each production rule into rules from an applicative tree grammar using the algorithm T_{ta} as shown in the following: The input of the algorithm is a tree grammar represented by a map of production rules (cf. Definition 5).

Algorithm 1 Applicative Tree Grammar Translation into Tree Grammar

```
1: function ENCODE_APPLICATIVE_TREE_GRAMMAR(𝒢_ap)
2:     treeGrammar ← ∅
3:     for all r in R do
4:         case r := α ↦ c then
5:             treeGrammar ← treeGrammar ∪ α ↦ c()
6:         case r := α ↦ @(β1, β2) then
7:             treeGrammar ← treeGrammar ∪ α ↦ COMPUTE_RIGHT-HAND_-
   SIDE (𝒢_ap, β1, β2)
8:     end for
9:     return treeGrammar
10: end function
11:
12: function COMPUTE_RIGHT-HAND_SIDE(𝒢_ap, β1 ∈ 𝒩, β2 ∈ 𝒩)
13:     ths ← ∅
14:     for all r in R do
15:         case r := β1 ↦ c ∈ R then
16:             rhs ← rhs ∪ c(β2)
17:         case r := β1 ↦ @(σ1, σ2) then
18:         if σ1 = ω then
19:             rhs ← rhs ∪ ∗()
20:         else
21:             rhs ← rhs ∪ ( COMPUTE_RIGHT-HAND_SIDE (𝒢_ap, σ1, σ2), β2)
22:         end if
23:     end for
24:     return rhs
25: end function
```

The output is a set of rules representing applicative tree grammar. As mentioned above, the applicative tree grammar manages ω types. The translation of such grammars into tree grammar leads to computation of rules in the form of $\alpha \mapsto *()$. To demonstrate that the application of Algorithm 2 on the result of Algorithm 1 can produce the original application tree grammar, we have to manage such rules. In other words, we have to show that $T_{ta}(T_{at}(\mathcal{G}_{ap})) = \mathcal{G}_{ap}$ is satisfied. The algorithm encodes each production rule in the tree grammar by an application rule using function COMPUTE_FUNCTION_TYPES (line 4). If we have rules in the form of $\alpha \mapsto c(\beta_1, \ldots \beta_n)$ and where c is a combinator, $\alpha, \beta_1, \ldots, \beta_n$ with $n \in \mathbb{N}$ are nonterminals, they are encoded as follows: $\alpha \mapsto @(\beta_n \to \alpha, \beta_n)$, $\beta_n \to \alpha \mapsto @(\beta_{n-1} \to \beta_n \to \alpha, \beta_{n-1})$, …, $\beta_2 \to \cdots \to \beta_n \to \alpha \mapsto @(\beta_1 \to \cdots \to \beta_n \to \alpha, \beta_1)$, $\beta_1 \to \cdots \to \beta_n \to \alpha \mapsto c$ in lines 12–19. Rules of the form $\alpha \mapsto c()$ are encoded by $\alpha \mapsto c$, shown in line 14. If the tree grammar contains any rules of the form $\alpha \mapsto *()$, the algorithm collects all combinators. Additionally, for each combinator c, it adds a rule of the form $\alpha \mapsto c$ (line 17). The output of the algorithm is a set of rules that represents the applicative tree grammar. Using the algorithm, each tree grammar can be translated into an applicative tree grammar, thus $T_{ta}(\mathcal{G}) = \mathcal{G}_{ap}$. The application of T_{ta} (Algorithm 2) to the result of algorithm T_{at} (Algorithm 1) returns an applicative tree grammar

Algorithm 2 Tree Grammar Translation into Applicative Tree Grammar

```
1: function ENCODE_TREE_GRAMMAR (𝒢)
2:     applicativeTreeGrammarSet ← ∅
3:     for all r in R do
4:         applicativeTreeGrammarSet ← applicativeTreeGrammarSet ∪
5: COMPUTE_FUNCTION_TYPES(r)
6:     end for
7:     return applicativeTreeGrammarSet
8: end function
9:
10: function COMPUTE_FUNCTION_TYPES(r)
11:     rulesSet ← ∅
12:     case r := α ↦ c(β_1, ..., β_n) then
13:         ruleSet ← ruleSet ∪ α ↦ @(β_n → α, β_n) ∪ ··· ∪ β_1 → ··· → α ↦ c
14:     case r := α ↦ c() then
15:         ruleSet ← ruleSet ∪ α ↦ c
16:     case r := α ↦ ∗() then
17:     for all c ∈ ℱ do
18:             ruleSet ← ruleSet ∪ α ↦ c
19:     end for
20:     return ruleSet
21: end function
```

equal to the input of algorithm T_{at}, such that $T_{ta}(T_{at}(\mathscr{G}_{ap})) = \mathscr{G}_{ap}$ is satisfied. In the reverse direction we have: $T_{at}(T_{ta}(\mathscr{G})) = \mathscr{G}$. Here, the input of the algorithm T_{ta} is a tree grammar $\mathscr{G}$ equal to the output of the algorithm T_{at}. From now on, we assume that all grammars are applicative, unless otherwise stated.

The algorithmic generation of the tree grammars by the inhabitation machine represented by the (CL)S Scala Framework is described in [29]. The algorithm grows the tree grammars stepwise. This structure allows an easy enumeration of inhabitants. In [50], decision problems such as emptiness, uniqueness, and finiteness and complexity are presented based on the corresponding machine, a non-deterministic finite tree automaton (NFTA) [50].

2.5 (CL)S Scala Framework

The (CL)S is a synthesis framework providing an implementation of a type inhabitation algorithm for combinatory logic with intersection types [103; 34]. The implementation of the inhabitation algorithm began several years ago. Among the first versions of the synthesis algorithm was presented in the PhD work of Boris Düdder [57]. This version represents an implementation in the programming language F#. The research in the field was continued by Andrej Dudenhefner [60]. His work investigates and improves the performance of the inhab-

itation approach implemented in F#. At the same time, Jan Bessai began to implement the theory presented in [31]. The implementation of the inhabitation algorithm is fully integrated into the programming language Scala. In his work [29], Jan Bessai has proved the soundness and completeness of the algorithm. The (CL)S Framework's formalisation is achieved with use of the theorem prover Coq [31]. The (CL)S Framework is publicly available [36].

The intended use of the (CL)S Framework is to automatically compose software, where the typed combinators represent software components. Programs are generated from a user-specified repository Γ of typed combinators. This approach is used for the automatic synthesis of software presented in [35; 32; 33; 59; 76; 13].

The integration of the (CL)S algorithm into the Scala program language allows for simple specification of the framework to make components usable for the current application. The framework is available as a Scala library that enables the use of any local IDE and from any program. It supports an extension of intersection types with constructors and products, as well as the adjusted BCD subtype relation system presented in Section 2.5.4 [29]. This type expression represents the specification of the applicative terms presented in Definition 2 and is defined as follows:

Definition 9 (Intersection Types with Products) Intersection types, denoted by $M, N \in \mathbb{T}$, are formed as follows:

$$M, N ::= \omega \mid c(M) \mid (M * N) \mid (M \to N) \mid (M \cap N)$$

Constructors are represented by c and ranged over a set of constructors denoted by $\mathbf{C}$. M and N denote intersection types. They can be constructed from products $(M * N)$, function types $(M \to N)$, and from intersection $(M \cap N)$. □

Constants are unary constructors. For the application of multiary constructors, products must be used required. For example, using products, we can encode the position in a two-dimensional Boolean array as $Pos(x * y)$. Here, products are left-associative – for example, $M * N * L = ((M * N) * L)$ – while intersections and arrows are right-associative – for instance, $M \cap N \cap L = (M \cap (N \cap L))$ and $M \to N \to L = (M \to (N \to L))$.

The Scala implementation of the inhabitation algorithm takes a finite substitution space, a subtype environment, a repository, and a sequence of target types [29]. These domain-specific requirements are explained in the following sections. The inhabitation machine results in a tree grammar, where nonterminals are represented by types (cf. Section 2.3).

2.5.1 Substitution Space

In contrast to FCL, the (CL)S Scala Framework supports the definition of substitution space. An input specification can include a definition of substitutions allowed for type variables.

Usually, the beginning of the specifications states the kinding declaration for variables.

```
val alpha      = Variable("alpha")
val kinding: Kinding = Kinding(alpha)
   .addOption('Pen)
   .addOption('Pencil)
```

Here, constructors are represented by `'Pen`, `'Pencil`, and `'Drawing`. If a combinator definition specifies a semantic type such as: `val semanticType ='Drawing(alpha)`, where the variable `alpha` is specified instead of a concrete type, the inhabitation algorithm considers the following well-typed extension: `'Drawing('Pen):&: 'Drawing('Pencil)`. Using `Kinding`, the algorithm iterates over the possible assignments of the variables and constructs a substitution map according to the defined substitution space to replace the variables.

The (CL)S Framework allows specifications represented by a notation similar to the mathematical one. In Scala syntax, constants and type constructors must be prefixed by a single quotation mark. This notation can be seen in the example above. Function types are represented by =>: – for example, $\sigma \rightarrow \tau$ becomes σ =>: τ. The binary intersection type operators are represented by :& – for example, $\sigma \cap \tau$ becomes σ :& : τ tau. Constructors with multiple arguments can be represented using the binary product operator $*$. In Scala syntax, the operator is specified by $<*>$. Therefore, $\sigma * \tau$ becomes $\sigma <*> \tau$.

2.5.2 Repository

Two options are accepted by (CL)S for the implementation of the domain-specific repository Γ. The first way is based upon mathematical notation. The repository includes typed combinators of the form $(c : \sigma)$. The combinator name is denoted by c and σ is an intersection type. The second option requires specifications based on the standard Scala components. The combinators are annotated by `@combinator`. The typical specification for combinators in the repository is defined as objects that have a function `apply` annotated by its native type and a semantic type. The specification of a semantic type is optional. Listing 2.1 encodes part of a repository Γ using the first mentioned option for specification of a repository, where `Player` represents the native type and the position description – the semantic type. For the sake of the readability, only the specification of the combinators $start$ and $right$ is presented.

```
val Gamma =
    Map("start " -> ('Player :&: 'Pos('0<*>'2)),
        "right" ->  ('Player =>: 'Player) :&:
                    ('Pos('0<*>'1) =>: 'Pos('1<*>'1)) :&:
                    ('Pos('1<*>'1) =>: 'Pos('2<*>'1)) :&:
                    ('Pos('0<*>'3) =>: 'Pos('1<*>'3)) :&:
                    ('Pos('1<*>'3) =>: 'Pos('2<*>'3)))
```

Listing 2.1: Mathematically similar representation of a repository in Scala

As mentioned, the notation of specifications in the repository is similar to the mathematical one. Listing 2.2 shows an example of Scala specification. Here, a part of the class `LabyrinthRepository` is presented that also implements the specification of combinators $start$ and $right$ with native types.

```
1  class LabyrinthRepository {
2      @combinator object start {
3          def apply(player: Player): Player = player.goRight()
4              val semanticType =
5                  ('Pos('0 <*>  '2))
6      }
7      @combinator object right {
8          def apply(player: Player): Player = player.goRight()
9              val semanticType =
10                 ('Pos('0 <*>  '1) =>: 'Pos('1 <*>  '1)) :&:
11                 ('Pos('1 <*>  '1) =>: 'Pos('2 <*> '1)) :&:
12                 ('Pos('0 <*>  '3) =>: 'Pos('1 <*> '3)):&:
13                 ('Pos('1 <*>  '3) =>: 'Pos('2 <*> '3)))
14     }
15 }
```

Listing 2.2: Scala representation of combinators with native and semantic types

We can extract type information from the specification of the combinators. For example, combinator $right$ with its semantic and native types is represented in mathematical notation by: $right : \{(Player \rightarrow Player) \cap (Pos(0,3) \rightarrow Pos(1,3)) \cap (Pos(1,3) \rightarrow Pos(2,3))\}$, as shown in Figure 4.6. The semantic type specification (cf. lines 4 and 9) provides additional conditions on the use of the combinators. They are user-defined and one-to-one adopted. The algorithm constructs such repositories to maps of the kind presented in Listing 2.1.

2.5.3 Inhabitation Request

Automatic synthesis is performed by answering the inhabitation question presented in Section 2.1. As mentioned, the inhabitation algorithm searches for terms that are formed from the typed combinators. The resulting terms have the given target type τ. Listing 2.3 presents a labyrinth example for an inhabitation request in Scala. Section 4.3.1 details the example presented here. The algorithm searches for solutions with native type `Player` and semantic type `'Pos('2, '3)` defined mathematically as follows: `Player` $\cap$ `Pos(2, 3)`.

```
1  val labyrinthRepository = new LabyrinthRepository
2  val reflectedRepository:ReflectedRepository[LabyrinthRepository] =
       ReflectedRepository(labyrinthRepository,
3                          substitutionSpace,
4                          semanticTaxonomy,
5                          classLoader)
```

```
6 val results:InhabitationResult[Player] =
7         reflectedRepository.inhabit[Player]('Pos('2, '3))
```

Listing 2.3: Definition of inhabitation request in (CL)S Framework

In line 1, the variable `labyrinthRepository` is an instance of the domain-specific repository `LabyrinthRepository`. Listing 2.2 shows a part of the repository. To define the inhabitation context, a reflection of the repository is needed. Line 2 shows the construction of `reflectedRepository` with the instance of the user-specific repository. Moreover, the reflected repository is constructed with substitution space, semantic taxonomy. The Java class loader for the Java Virtual Machine is also used to achieve reflection information.

2.5.4 Subtype Environment

A specification of the subtype environment is also allowed. As mentioned in Section 2.5, the subtype relation $\leq$ is the least relation closed under the rules presented in Definition 3. In (CL)S, the subtype relation is extended to include constructors and products using the following rules [37]:

- The (CAx) rule controls the order of constructors with an argument.

$$\frac{c \leq d \qquad M \leq N}{c(M) \leq d(N)} \text{ (CAx)}$$

- The (CDist) rule allows distribution of intersection over unary constructors.

$$\frac{}{c(M) \cap c(N) \leq c(M \cap N)} \text{ (CDist)}$$

- The (ProdSub) rule allows, such as the (Sub) rule, co- and contra-variant subtyping of functions. In this case for products.

$$\frac{M_1 \leq N_1 \qquad M_2 \leq N_2}{M_1 \star M_2 \leq N_1 \star N_2} \text{ (ProdSub)}$$

- The (ProdDist) rule allows distribution of intersections for products.

$$\frac{}{(M_1 \star M_2) \cap (N_1 \star N_2) \leq M_1 \cap N_1 \star M_2 \cap N_2} \text{ (ProdDist)}$$

The subtype relation of native types can be used to reflect inheritance. A combination of subtyping and semantic types allows for a representation of taxonomic hierarchies. In the following part of this section, we consider an example that deals with the possible directions of movement in a labyrinth. As shown in Listing 4.5, taxonomy is used to represent four possible

directions. Here, taxonomies are relations between constructor names. Hence, for a taxonomy t, we have $'M <=' N$ if $'N$ is in $t('M)$. In line 2, the definition of the supertype `Direction` is shown. The usage of the function `addSubtype` in the following lines represents the subtype relation. We define a map with all subtypes belonging to a supertype as follows:

```
1     lazy val taxonomy =
2         Taxonomy (" Direction ")
3         . addSubtype (" Left ")
4         . addSubtype (" Right ")
5         . addSubtype (" Up ")
6         . addSubtype (" Down ")
```

Listing 2.4: Representation of taxonomy information

As can be seen, the types `Left`, `Right`, `Up`, and `Down` are subtypes of the supertype `Direction`. In other words, for a taxonomy t we have `Left` $\leq$ `Direction`, `Right` $\leq$ `Direction`, `Up` $\leq$ `Direction`, and `Down` $\leq$ `Direction` if `Direction` is in $t(\{$`Left, Right, Up, Down`$\})$. The algorithm arranges the transitive reflexive closure of the taxonomies according to the BCD subtype relation presented in Section 2.2 so that it results in:

```
{(Left, Direction), (Right, Direction), (Up, Direction), (Down,
  Direction), (Left, Left), (Right, Right), (Up, Up), (Down, Down)
  , (Direction, Direction)}
```

2.6 Visualisation of Tree Grammars

Just as the writing of clean code [93] makes the code maintainable and leads developers to better understand the implementing program, clear and proper visualisation of data is crucial for the analysis and interpretation of information. A visualisation using graphs is a common way to obtain a representation of much information. A directed graph is a pair of $\mathcal{G} = (V, E)$ with a set of nodes V and a set of edges $E \subseteq V \times V$, where $e \in \{(u, v) | (u, v) \in V^2$ and where $u \neq v\}\}$ [28; 27]. This visualisation approach has proven necessary for the analysis, presentation, documentation, and comparison of data. Graphical visualisation allows structural analysis to detect interesting relations and effects, as well as defects and problems. Outside of the functional advantages, effective visualisation results in a salient representation of vast data information so that this representation can also benefit nonexperts. The motivation to visualise the tree grammars generated by the (CL)S Scala Framework centres on being able to analyse results, detect defects, and better understand the inhabitation algorithm. In this way, a user gains an idea of the functionality of the inhabitation technique. Accordingly, the structure of the results using the graphical visualisation and the usage of the defined combinators can be discovered. Often, the user who creates the repository with typed combinators expects specific solutions. However, the repository can contain combinators defined with faulty or incomplete type specifications or include typos. In such case, it can be difficult for the user to understand

why the inhabitation algorithm yields unexpected solutions or why not all combinators are used. Chapter 4 provides detailed information about discovering specification problems with the IDE.

This section considers the visualisation of tree grammars resulting from the (CL)S Scala Framework using graphical structures. We start by presenting the first version of our visualisation approach. In the context of this approach, we applied *directed compound graphs* in terms of user-friendly representation of the tree grammars (see Section 2.6.1). Afterwards, we introduce the *hypergraphs* as a new graphical representation approach of tree grammars computed by the inhabitation algorithm. In Section 2.6.3, we discuss the advantages and disadvantages of these approaches and establish decisions regarding the approach used for visualisation in the IDE. In the following sections, we consider the tree grammar presented in Definition 5 and not the applicative tree grammars because of the clear representation of the results. The reason is that applicative tree grammars are encoded by binary hypergraphs. The visualisation of extensive inhabitation results leads to an unclear and bulky representation.

2.6.1 Directed Compound Graphs

At the beginning of the development of the IDE for the (CL)S Framework, we applied *directed compound graphs* to visualise the resulting tree grammars. Such graphs are typically used to represent expansive information as networks, for instance in the field of chemistry, biochemistry, or bioinformatics [115; 67]. The definition of these graphs is based on [121].

Definition 10 (Directed Compound Graphs) A compound graph G is a quadruple $G = (C, V, E, F)$ with

- a finite set of compound nodes C,
- a finite set of nodes V,
- a finite set of adjacency edges $E \subseteq V \times C$ where an edge $e \in \{(u, v) | u \in V \text{ and } v \in C\}$,
- a finite set of inclusion edges F, where an adjacency edge between vertexes v and u does not exist so that

$$\{(v, u) \in F \mid u \in Anc(v) \cup Desc(v)\} = \emptyset,$$

  where vertex $u \in V$, $v \in C$, $Anc(v)$ is a finite set of ancestors of v, and $Desc(v)$ is a finite set of descendants of v. □

If we have an edge $f = (v, u) \in F$, then u includes v. Using directed compound graphs, tree grammars can be represented with terminals as nodes ($V = \mathscr{F}$) and non-terminals as compound nodes ($C = \mathscr{N}$). For each production in the tree grammar $\alpha \rightarrowtail c(\beta_1, \beta_2, \ldots \beta_n)$,

where $c \in \mathscr{F}$ and $\alpha, \beta_1, \beta_2, \ldots, \beta_n \in \mathscr{N}$ with $n \geq 0$, we add edges $e_1, e_2, \ldots, e_n$ that extend from vertex c to compound nodes $\beta_1, \beta_2, \ldots \beta_n$, such that we have $e_1 = (c, \beta_1), e_2 = (c, \beta_2), \ldots, e_n = (c, \beta_n)$. Furthermore, we add inclusion edge f with compound node α including c, yielding $f = (\alpha, c)$.

We consider the following example to illustrate the representation of tree grammars through compound graphs:

$$\begin{aligned} \mathscr{G} = \{ & A \mapsto\ c(D, E),\ d(D), \\ & D \mapsto d(), \\ & E \mapsto e(A) \} \end{aligned}$$

Figure 2.1: Tree grammar example

The presented tree grammar $\mathscr{G}$ has a set of nonterminals $\mathscr{N} = \{A, D, E\}$ and a set of terminals $\mathscr{F} = \{c, d, e\}$. Figure 2.2 illustrates the directed compound graph visualising the tree grammar $\mathscr{G}$.

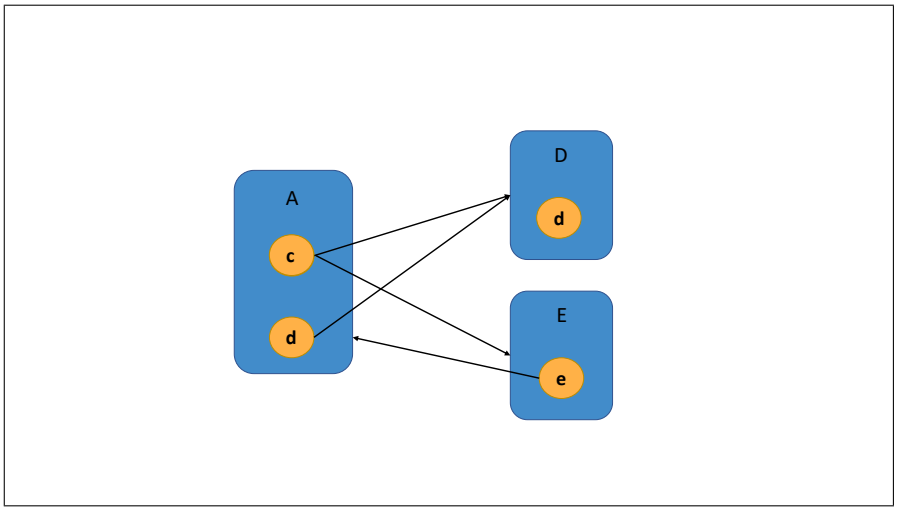

Figure 2.2: Tree grammar visualisation as a compound graph

The presented graph includes compound nodes $C = \mathscr{N}$ (marked in blue) and nodes $V = \mathscr{F}$ (marked in yellow). With respect to the above example, we have a graph $G_{\mathscr{G}}$ with

$$\begin{aligned} C_{\mathscr{G}} &= \{A, D, E\} \\ V_{\mathscr{G}} &= \{c, d, e\} \\ E_{\mathscr{G}} &= \{(c, D), (c, E), (d, D), (e, A)\} \\ F_{\mathscr{G}} &= \{(A, c), (A, d), (D, d), (E, e)\} \end{aligned}$$

Compound graphs are helpful in representing information about groups of data and their relationships. Here, we can easily investigate the types of combinators. The size of the compound nodes depends on the number of terminals with the same type specification.

2.6.2 Hypergraphs

Hypergraphs are used to visualise information in the fields of artificial intelligence, database theory, fuzzy logic, propositional logic, and others [20; 21]. They comprise an alternative representation to the compound graphs described above (see Section 2.6.3 for a comparison). Hypergraphs are like regular graphs, but their edges can connect more than two nodes (i.e., they have a cardinality). If we consider a hypergraph $\mathscr{H}$ with cardinalities of all hyperedges is $|E_i| = 2$ with $i = 1,\ldots,n$, then we have a 2-uniform hypergraph representing an ordinary graph [70]. Figures 2.3a and 2.3b exemplify an undirected graph and undirected hypergraph. In an ordinary graph, an incidence function maps every edge E to two nodes so that we have 1:1 connections, $E \rightarrow \{\{x,y\}|(x,y) \in V^2 \wedge x \neq y\}$ [27; 108]. The outgoing connections of the hyperedges represent finite vectors of nodes, n:n connections. In this case, each hyperedge has three nodes, in contrast to the regular graph, where the edges have exactly two nodes (cf. Figure 2.3a).

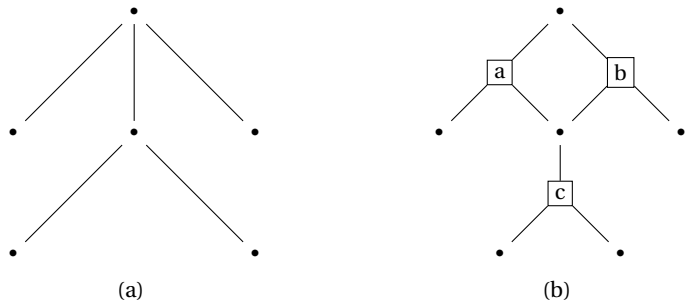

Figure 2.3: Example of graph (a) and hypergraph (b)

The following definition of hypergraphs derives from [62] and [108].

Definition 11 (Hypergraphs) A directed labelled hypergraph $\mathscr{H}$ over an alphabet of terminals $\mathscr{T}$ and an alphabet of nonterminals $\mathscr{N}$ is a quadruple $\mathscr{H} = (V, E, \text{nod}, \text{lab})$ with

- a finite set of nodes V,
- a finite set of edges (or hyperedges) E,
- a family of incidence functions nod, with $nod_E : E \rightarrow V^*$ and $nod_V : V \rightarrow E$, and
- a family of labelling function lab, with $lab_E : E \rightarrow \mathscr{T}$ and $lab_V : V \rightarrow \mathscr{N}$. □

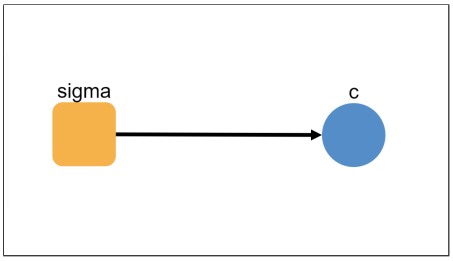

Figure 2.4: Visualisation of var rule as a hypergraph

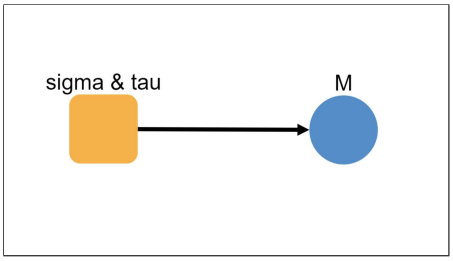

Figure 2.5: Visualisation of intersection rule (∩I) as a hypergraph

Every directed hyperedge has incoming and outgoing connections. The latter are denoted by nod_E, while the incoming connections are described by nod_V [68; 70]. To represent the tree grammar using a hypergraph, we consider the labelling alphabets of terminals and nonterminals. Each nonterminal $\mathcal{N}$ in a tree grammar is represented by nodes V, such that $V = \mathcal{N}$. Every node is labelled with types. For a production $\alpha \rightarrowtail c(\beta_1, \beta_2, \ldots \beta_n)$ with $n \in \mathbb{N}$, we add an edge $e \in E$ with $\text{nod}_V(\alpha) = e$ and $\text{nod}_E(e) = (\beta_1, \beta_2, \ldots \beta_n)$ where $\text{lab}(e) = c$ [30]. The range of $\text{nod}_E(e)$ for edge e is the set of nodes of e denoted by $rang(nod_E(e)) = n$. The numbering of the outgoing edges denotes the argument's positions.

The tree grammar encoding by hypergraphs is straightforward according to the inhabitation rules presenter in Section 2.2. For example, Figure 2.4 represents the var rule, Figure 2.5 illustrates the intersection rule, and Figure 2.6 displays the arrow elimination rule (→ E).

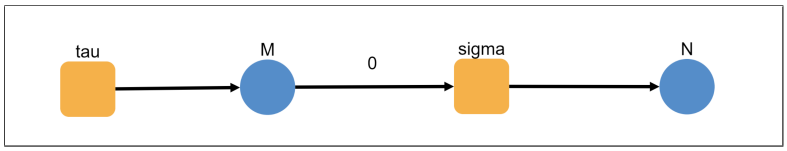

Figure 2.6: Visualisation of arrow elimination rule (→ E) as a hypergraph

The tree grammar $\mathcal{G}$, presented as compound graph in Figure 2.2 is illustrated using hypergraph in Figure 2.7. The target type is also A, which can be easily recognised in this layout. The represented elements are as follows: $V = \{A, D, E\}, E = \{c, d, e\}, nod_E = \{\{c, D\}, \{c, E\}, \{d, D\}, \{e, A\}\}$, and $nod_V = \{\{A, c\}, \{A, d\}, \{D, d\}, \{E, e\}\}$.

Subhypergraphs

The developed IDE for the (CL)S Scala Framework provides a perspective on which each inhabitant can be visualised separately. Detailed information about these perspectives is presented in Section 4.3.3. In this way, the user can investigate which combinators and types were used for the inhabitation. To visualise the inhabitants separately, we apply *subhypergraphs*, the

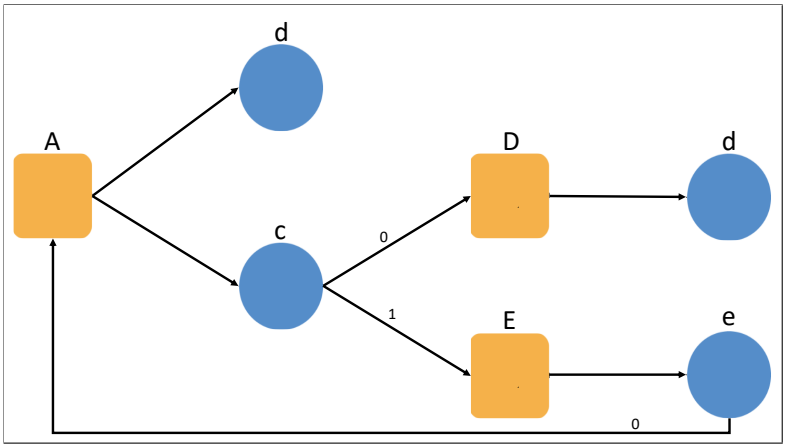

Figure 2.7: Visualisation of tree grammar $\mathscr{G}$ as a hypergraph

definition of which is based on [53].

Definition 12 (Subhypergraphs) A hypergraph $\mathscr{H}' = (V', E', \text{nod}_{E'}, \text{nod}_{V'}, \text{lab}')$ is called a subhypergraph of hypergraph $\mathscr{H} = (V, E, \text{nod}_E, \text{nod}_V, \text{lab})$, where $\mathscr{H}' \subseteq \mathscr{H}$, if $V' \subseteq V$, $E' \subseteq E$, $\text{nod}_{E'}(e) = \text{nod}_E(e)$, $\text{nod}_{V'}(e) = \text{nod}_V(e)$, and $\text{lab}'(e) = \text{lab}(e)$. □

As mentioned in Section 2.3, an inhabitant resulting from the (CL)S Framework is word w, where $w \in \mathscr{L}_\tau(\mathscr{G})$ where τ is a target type [29]. Thus, considering the example above, when we have $w = c(d, e(d))$, we construct a subhypergraph in terms of investigation of which combinators and rules were used for the construction of this term. Figure 2.8 illustrates a subhypergraph $\mathscr{H}'$ of the hypergraph $\mathscr{H}_\mathscr{G}$ shown in Figure 2.7. This graph represents the construction of term $d(d)$ derived for the start symbol A that is, again, at the beginning of the graph because of the tree-like structure.

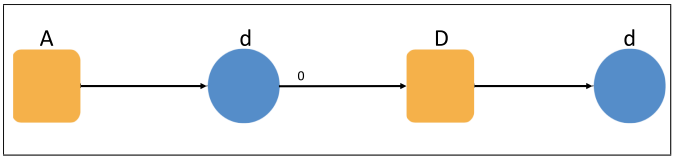

Figure 2.8: Visualisation of term $d(d)$ as a subhypergraph

2.6.3 Comparison

To verify the usability of the presented approaches for graphical visualisation, we investigate the readability and the flexibility of the constructed graphs. A poor layout representation can overstrain and confuse the user. According to Dogrusoz et al. [56], the criteria used to investigate the quality of a graph's visualisation are subjective. For this reason, we measured

the quality according to some generally accepted criteria such as minimal crossing number of edges, uniform distribution of nodes, and lack of overlapping of nodes [54; 56; 111; 124].

Experiments with different use cases have shown that for small tree grammars, both approaches can be applied. Nevertheless, the (CL)S Scala Framework aims to synthesise programs that can cause the generation of large tree grammars. For example, the use cases presented in Chapter 6 request repositories including between 60 and 294 combinators. The tree grammars generated by the algorithm grow exponentially [29]. In such a case, we investigated the readability and the clarity of the resultant graphs.

As mentioned in Section 2.6.1, the directed compound graphs are often applied in the field of chemistry and biologics. They suit to cases in which much information must be visualised to investigate the relations between elements, pairs of nodes, or groups. The clustering of the nodes represents the interactions between elements. Further information about the usage of compound graphs can be found in [68]. A problem with this approach in combinatory logic is that it is not practical in all situations. We aim to appropriately visualise how the inhabitation algorithm (cf. Section 2.5) computes solutions to increase the understandability of the process. Clustering of tree grammar information does not support this aim. The experiments with cumbersome solutions mentioned above have shown the representation of inhabitation results to be easier to understand and more precise without the aggregation of combinators by types (without compound nodes). That is, where too many nonterminals are represented using compound nodes (cf. Figure 2.2, blue rectangle), the visualisation is imbricated and disorganized. The resulting compound graphs become too big; nodes, hidden; and edges, difficult to discover. In this case, it takes time to order and understand the components. Considering the small examples presented in Figures 2.2 and 2.7, we can also see that the compound graph representation is unclear because of the edge–edge crossing, despite the small number of nodes.

Moreover, the arity of the combinators can be visualised through the labelling of $node_E(e)$ (cf. Figure 2.7). This representation is also possible with compound graphs using edge labels. For large solutions, however, the visualisations are overloaded and unclear because of the additional information.

The layouts of the graphs are also investigated. Section 4.3.2 discusses the importance and necessity of different graphical designs. A representation using hypergraphs allows different automatically generated layouts to be applied. Again, this approach allows better and more flexible navigation through a representation of large tree grammars. In this way, a user can select a layout with, for example, fewer overlapping edges and nodes or a tree-like representation where the target is at the beginning (cf. Figure 2.7).

In the search process, we applied a step-wise representation of the computation of terms. Each step visualises the target, the combinator(s), and the next target(s). The application of hypergraphs is more effective in representing large datasets, not only because of the range of different layouts but also because of the straightforward representation of the tree's growth. In a

more extensive graph, this feature is very significant for user-friendly visualisation. Section 4.3 outlines more details about the advantages of this step-wise visualisation.

For the above-discussed reasons, the (CL)S IDE supports the visualisation of tree grammars by hypergraphs. Chapter 6 introduces detailed information about and representations of these graphs, based on real application cases.

2.7 Satisfiability Modulo Theories

The application of Satisfiability Modulo Theories (SMT) to synthesis is diverse. For example, Solar-Lazama et al.[101] introduce an approach that completes partial implementations using synthesis. Another technique to combine SMT and synthesis is programming by examples presented by Gulwani et al. [83]. Further, Reynolds et al. [107] present a program synthesis approach completely realised with the use of an SMT solver. The synthesis problem in the case presented in Section 3.1 is solved using combinatory logic. SMT is applied as a filtering approach to restrict the complete enumeration of inhabitants computed by the (CL)S Framework. This restriction approach is detailed in Section 3.1.

SMT deals with the problem of the satisfiability of logical formulas regarding a background theory. Around 1980, the use of decision procedures began. In the 1009s, based on this knowledge, research in the field of SMT and SMT solvers began [40]. Nowadays, SMT solvers are vital to computer programs verification [10; 52]. SMT solvers prove the satisfiability of the SMT procedures. Section 2.7.2 briefly reviews SMT solvers. Within the filtering approach, we define the input formulas according to the SMT-LIB standard. In this way, we can use a wide range of solvers and benchmarks.

2.7.1 Satisfiability-Modulo-Theory Library

The Satisfiability-Modulo-Theory Library (SMT-LIB) is a standard first introduced within the scope of the PDRAP 2003 Workshop [102]. Clark Barrett, Pascal Fontaine, and Cesare Tinelli manage this initiative. The main goal is to provide a standard description of the background theories and a library of benchmarks for SMT solvers. More than 30 SMT solvers are publicly available. Using SMT-LIB, they can be compared and evaluated.

In this work, we use the current version 2.6 of the SMT-LIB standard, which consists of three main components: theory declarations, logic declarations, and scripts. The `Core` theory represents the basic and is included in every available SMT-LIB theory declaration [24]. Accordingly, the theory of the Booleans is always additionally allowed. This theory defines the basic Boolean operators, alongside the following functions:

- `(= x y Bool)` returns true if and only if the arguments `x` and `y` are identical.

- `(distinct x y Bool)` returns true if and only if the arguments `x` and `y` are not identical.

- `(ite Bool x y)` returns the second argument `x` if and only if the first argument is `true`. Otherwise the function returns the third argument `y`.

This work applies the theory of integers (the `Ints` theory), meaning we have a two-sorted theory that allows the standard theory of integers in addition to the basic Boolean operators [24].

In addition to the underlying first-order logic, the SMT-LIB format allows the declaration of a sublogic to restrict the set of allowed formulas. We use the linear fragment of the theory of integers (`Ints`) and the Booleans (`Core`) with uninterpreted constant symbols — linear integer arithmetic (LIA) [102], allowing existential and universal quantifiers over integers. Moreover, this logic allows addition and the functional symbol `*`, although only in the form `(* m n)` or `(* n m)`, where `n` is a free constant and where `m` is a term that can also be given a negative integer coefficient `(- m)`. To define the LIA as a logic, we also apply in the solver environment the following command: `( set-logic LIA )`.

The third component represents sequences of commands defined in a syntax close to this of the programming language LISP. The order of the commands is essential for communication with SMT solvers. More information about the scripts used in this work is presented and discussed later in this section. After the usage of command `set-logic`, commands for declaring new sort or function symbols can be used. To declare a function f that receives $\sigma_1 \ldots \sigma_n$ and returns σ, commands of the form `(declare-fun f` $(\sigma_1 \ldots \sigma_n)$ σ`)` with $n \geq 0$ must be used. The function name should be unique or an error will be reported. The command `(define-fun f ((`x_1 σ_1`) ... (`$x_n \sigma_n$`)) `σ `t)` can be used to define a function f with arguments that are not only sorts but also variables. Here, the name must also be unique. This command is equivalent to the following combination of commands:

```
1 (declare-fun f (σ1...σn) σ)
2 (asserts (forall ((x1 σ1)...(xn σn)) (= (f x1...xn) t)).
```

To instruct a solver to assume the satisfiability of a formula, we use a command of the following form:

```
(assert <expression>).
```

We thereby define constraints that can be conditions or restrictions. In this example, we use a binder is used that represents the universal quantifiers of first-order logic `forall` and introduces n variables $(x_1, \ldots, x_n)$, where each variable is associated with a sort $(\sigma_1, \ldots, \sigma_n)$. The well-sorted term t is of the sort σ. Otherwise, the solver reports an error.

2.7.2 SMT Solver

As mentioned, SMT solvers deal with the satisfiability of formulas regarding background theories [52]. They are automated and the problems to be checked are defined according to the SMT-LIB standard.

To start checking the satisfiability using the SMT solver, the command `(check-sat)` must be used at the end of the script. Three answers are possible:

- *sat:* This is a standard output if a formula ϕ is satisfiable. The solver can find a model under which the set of assertions is satisfiable by using the command `get-model`.
- *unsat:* A solver returns this output if a formula ϕ is unsatisfiable. In this case, the solver cannot find a model under which all assertions in the script are true. Using the command `(get-unsat-core)`, the solver returns the notifications of the assertions that are unsatisfiable.
- *unknown:* In this case, the solver cannot determine the satisfiability of a problem. In such a case, quantified expressions can be the problem. Since they are challenging, they can force an SMT solver to return this answer. The complexity of the formulas when the available memory is not enough or the time is expired can also cause this answer to be returned. This behaviour limits the usefulness of the approach in use cases of quantified verification conditions [106].

As mentioned, more than 30 SMT systems exist. Among them, some currently used systems are Alt-Ergo, AProVE, Boolector, CVC4, Minkeyrink, Q3B, SMT-RAT, STP, veriT, Yices 2,and Z3 [102]. In this work, we use the SMT Solver Z3 from Microsoft Research [52]. Within our research, solvers can be replaced with reasonable effort, due to the SMT-LIB standard. In this work, we use an abstraction over the SMT-LIB language in Scala that allows Z3 to be easily replaced, for example, by CVC4 [4]. Investigations based on other SMT solver were not considered in this work because of the filtering results discussed in Chapter 5.

Chapter 3

Filtering of Terms

Each application has specific requirements. A significant advantage of the (CL)S Scala Framework is its ability to manage variability based on a user-specified repository including typed combinators. However, the underlying type system can limit the expression of domain-specific knowledge. For example, the consideration of the logical connections such as conjunction, disjunction, and negation by intersection types is not explicitly provided. Using combinatory logic, a user can locally specify type information on a combinator, but the expressiveness of the results' global structure is not straightforward using type variables. For instance, it is complex to consider additional expressions of requirements referring to the resulting programs' and solutions' behaviour more specifically and adequately by the type system. Furthermore, a disadvantage of this synthesis approach is that the resulting terms can be both trivial and irrelevant [29]. For example, the term $id(x)$ is the same as $id(\ldots(id(x)))$. In addition to providing a user-friendly IDE, a filtering approach was developed. Based on domain-specific constraints, the filtering approach restricts the set of type correct inhabitants. In this way, the user can define restrictions to forbid the direct application of combinator id to some term x or other domain specifications required from the current application.

The filtering approach supports users getting only those solutions that comply with specific requirements and avoiding these solutions with identical execution and runtime behaviour by restricting certain terms produced by the (CL)S Scala Framework. Thus, we can define restrictions for specific use cases where the order of combinators is essential. For example, order definition is crucial for use cases, where fabric plans must be synthesised or in cyber-physical systems, where hardware resources are limited. In the scope of program synthesis, a runtime exception can be avoided using constraints on the order.

In his work, Jan Winkels [132] deals with the automatic composition of factory planning projects using combinatory logic. This application illustrates a use case where the synthesis provides many variants of possible planning workflows that comply with the specified target type. However, an additional examination is necessary. This way, the consideration of project-

specific numerical and module-specific constraints should be ensured. Numerical constraints can comprise budget limits, time limits, or resource limits. Incompatibility between modules is an example of a module-specific constraint. To ensure the computation of a domain-specific solution, Jan Winkels uses the library *Graph for Scala* [12] for the verification. It provides functionality for the graph's generation. Based on these graphs, the user can apply constraints to ensure that domain-specific restrictions are considered. The graph's structure resembles such plans and schedules for factory planning. This approach is also suitable for such applications because of the JSON interface [132]. However, *Graph for Scala* cannot be used for the general filtering approach because the solutions can be checked only one by one. Accordingly, the approach cannot ensure there is a solution until it finds one. Very often, the number of terms computed by the (CL)S Scala Framework is infinite. In such cases, this filtering method is inefficient.

Moreover, the filtering approach can support users during the development of the input specification. In some cases, it remains possible to use the type system to restrict the solution space, although this procedure can be complicated when a repository is large. A user can investigate the current repository using a specific pattern for the restriction of the terms. Based on the filtered results, the developer can still decide whether the repository should be changed. An example of such a development is the application case presented in Section 6.2 where the authors synthesise milling processes using (CL)S.

In this section, we present and discuss approaches for filtering out trivial and irrelevant terms, as well as those not complying with the application's requirements. The emptiness problem presented above is decidable by these approaches. The earliest approach is based on the combination of the (CL)S Scala Framework and the SMT. The resulting tree grammars are translated into SMT formulas, and the restrictions are formulated as constraints so that SMT solver can decide whether a solution exists among the constraints. This approach is detailed in Section 3.1. The second approach is based on a restriction modifying the tree grammar produced by the inhabitation (s. Section 3.2). It works recursively based on regular expressions that must be restricted. The disadvantages of this approach lead to the development of the third filtering algorithm. The result is also a tree grammar so that the emptiness problem is also decidable [50]. In this work, we call such user-defined expressions *patterns*. A detailed discussion of the algorithm and the formal correctness are presented in Section 3.3. Moreover, this section presents certain essential cases in which term filtering is required (see Section 3.3.2) as well as a discussion of the advantages and disadvantages of this approach (see Section 3.3.3). Section 3.4 presents a parser technique that supports the filtering approach and that is integrated into the IDE.

3.1 Filtering Based on Satisfiability Modulo Theories

This section introduces a filtering approach called CLS-SMT. It combines the strengths of the (CL)S Framework and SMT. By restricting user-defined constraints, the approach increases the usability of the IDE and compensates for the limitations imposed by the type system discussed in the following. This technique was first presented in [85]. Background information about SMT is briefly presented in Section 2.7.

3.1.1 Filtering Approach

Using the (CL)S Framework, a user defines a repository including combinators with type specifications. The synthesis allows the construction of all possible combinations that cannot be easily restricted according to user-specified constraints. Furthermore, as mentioned, this approach does not consider the following logical connectives: conjunction, disjunction, and negation. These drawbacks can be compensated by SMT, which allows the formulation of such connectives.

The approach translates the tree grammars computed by the (CL)S Framework into a set of SMT formulas. Afterwards, using domain-specific constraints and an SMT solver, we restrict the resultant set of inhabitants. The filtering approach solves the following problem: Find a word M that is both grammatically correct (which implies well-typedness according to the repository and typing rules) and fulfils all user-defined constraints. Figure 3.1 illustrates the implemented CLS-SMT filtering approach.

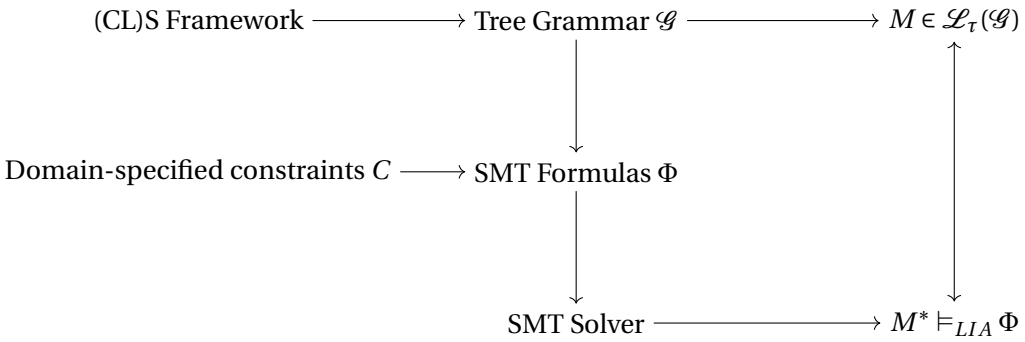

Figure 3.1: Overview of SMT filtering approach

In any tree grammar $\mathscr{G}$ generated by (CL)S, a term M is a word of the grammar $\mathscr{G}$, such that $M \in \mathscr{L}_\tau(\mathscr{G})$ where $\mathscr{L}_\tau(\mathscr{G})$ is the language of the grammar with target symbol τ if and only if $\Gamma \vDash M : \tau$. The framework then automatically translates the tree grammar $\mathscr{G}$ into a set of appropriate SMT formulas ϕ. A detailed explanation of the translation process is presented in Section 3.1.2. Domain-specified patterns have to be formulated as constraints according

to the SMT-LIB standard. The approach collects the formulas into a script. Let Φ denote the conjunction of the formulas, meaning we have $(\psi \wedge \phi) \in \Phi$, where ϕ is already translated into a tree grammar. The framework passes the generated script to an SMT solver so that satisfiability can be considered. A formula is satisfiable if there is a model M^* such that $M^* \vDash_{LIA} \Phi$. If the solver cannot find a model, then the formula is unsatisfiable. Any SMT tree model satisfying the constraints represents a term M of the grammar $\mathscr{G}$, and every word $M \in \mathscr{L}(\mathscr{G})$ can be translated to an SMT model M^*. Therefore, we have $M^* \vDash_{LIA} \neg\psi \wedge \phi$. In other words, the word $M \notin \mathscr{L}(C)$ and $M \in \mathscr{L}(\mathscr{G})$, where $\mathscr{L}(C)$ is the language of all terms fulfilling the domain-specific constraints; therefore, M is the complement language of $\mathscr{L}(C)$, $M \in \mathscr{L}(\overline{C})$. In this case, a satisfiable formula is simultaneously type correct and specification correct.

3.1.2 SMT Script Generation

This section presents the generation of the third component required by the SMT-LIB standard, namely – the scripts. To perform this filtering approach, we must define formulas according to the standard. The formulas collected into a script can be pushed to an SMT solver. SMT scripts contain a sequence of commands. If the definition and the order of the commands are correct, the satisfiability of the formulas can be checked.

Tree Grammar to SMT Formulas

Based upon the completeness of the inhabitation algorithm, we know that if there is a non-empty applicative tree grammar $\mathscr{G}$, there is at least one word M that can be derived from the grammar for the target type τ. In other words, we have $M \in \mathscr{L}_\tau(\mathscr{G})$. As mentioned, the inhabitation algorithm supports subtyping. As such subtyping rules have been fully applied when the tree grammar is constructed and the encoding into SMT formulas no longer needs to consider them. We define a data structure for applicative terms as follows:

Definition 13 (Inhabitant Tree) An inhabitant tree is a binary tree over integers. It is defined as follows:

$$leftChild,\ rightChild \in inhabTree ::= 0\ \ leftChild\ rightChild \mid c,$$

with $c \in C = \{1, \ldots, n\}$ and n representing the finite number of combinators used in a tree grammar $\mathscr{G}$. Each vertex has exactly two children ($leftChild$ and $rightChild$). A vertex with children is labelled at 0. Such nodes are called application nodes. Along with the nonterminals, functions $leftChild$ and $rightChild$ are declared, which partially map the elements of $inhabTree$ to their respective left and right children, if they exist. □

An inhabitation tree with a combinator with k arguments includes k application nodes. The combinator symbol denotes the leftmost leaf. The rightmost leaf of the parent element of the

combinator represents the nth argument of this combinator c. We represent an application node by @. According to the definition, an application node has exactly two children. Let us consider the example presented in Figure 3.2.

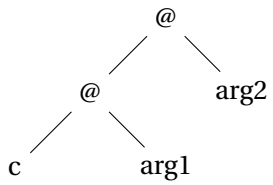

Figure 3.2: Visual representation of inhabitant tree

It represents a visualisation of the term $((c(arg1))arg2)$, where combinator c is applied to the arguments $arg1$ and $arg2$. The corresponding inhabitation tree is as follows:

$$0\ (leftChild\ (0\ (leftChild\ 1)\ (rightChild\ 2))\ (rightChild\ 3)$$

Here, combinator c is encoded as 1, $arg1$ as 2, and $arg2$ as 3.

A tree grammar can also be represented as an SMT formula. Let I be an inhabitation tree with $v \in V$, where V is a finite set of vertices. We introduce the following functions additionally:

- labelling function $inhabitant\colon\ V \mapsto \Sigma_V$, where Σ_V represents the alphabet of vertex labels of a tree $\Sigma_V \in \{0\} \cup C$.

- mapping function $ty : V \mapsto \mathcal{N}$, where $\mathcal{N}$ is the finite set of nonterminals in the grammar generated by the inhabitation algorithm. The function maps vertices of a tree to nonterminals from the tree grammar.

Suppose $\mathcal{G} = (\tau, \mathcal{N}, \mathcal{F}, R)$ is an applicative tree grammar (cf. Definition 7) constructed by the (CL)S Scala Framework with a production rule $\alpha \mapsto c(\beta_1, \beta_2)$, where are nonterminals and $c \in \mathcal{F}$ terminal symbol. Each production rule from the tree grammar is translated into theory-specific structural constraints on the tree by the application of Algorithm 3. Each translated rule represents a possible set of subtrees. Here, each combinator symbol of the rules is also represented by the leftmost leaf in the subtrees. Figure 3.3 illustrates the rule considered above.

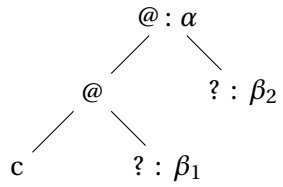

Figure 3.3: Representation of a production rule

Let i be the node $@ : \alpha$ that represents nonterminal α. Then, we have the combinator c on position `(leftChild (leftChild i))`. The arguments are typed according to β_1 and β_2. They are at positions `(rightChild (leftChild i))` and `(rightChild i)`, respectively. According to these rules, we can also construct subtrees for β_1 and β_2.

Applying Algorithm 3, we get Boolean expressions that encode LIA constraints for the rules from the tree grammar. These expressions are included in an assertion with a $forall$ expression and evaluated to verify whether the inhabitation tree is valid.

Algorithm 3 Rule Translation

```
1:  function TRANSLATE_RULE (typeId, values)
2:      xorSet ← ∅
3:      for all (combinator, parameters) in values do
4:          cTransl ← TRANSLATE_COMBINATOR(combinator, parameters)
5:          xorSet ← xorSet ∪ cTransl
6:      end for
7:      return (ite (= (ty i) typeId) (xor xorSet) true)
8:  end function
9:
10: function TRANSLATE_COMBINATOR(combinator, args)
11:     constrSet ← ∅
12:     currentAddress ← i
13:     pList ← args.reverse
14:     for all p in pList do
15:         constrSet ← constrSet ∪ (= (ty (rightChild currentAddress)) p)
16:         constrSet ← constrSet ∪ (= (inhabitant currentAddress) 0)
17:         currentAddress ← (leftChild currentAddress)
18:     end for
19:     combinatorConstraint ← (= (inhabitant currentAddress) combinator)
20:     combinedSet ← combinatorConstraint ∪ constrSet
21:     return (and (combinedSet))
22: end function
```

The function `Translate_Rule` considers each rule sorted in the given set of values (lines 3–5). The function `Translate_Combinator` returns a translation of a combinator and its argument list. The subtrees representing the translation of the combinators in a current production rule are combined using the function `xor` (line 7). This function takes two or more arguments, but here, in order to increase readability, the pseudo-code represents an application of the function `xor` and `and` to sets. The function `xor` is used because for each production rule, only one combinator can be used in the current subtree. The labelling of the nodes can be expressed using constraints on the functions $inhabitant$ and ty. In line 7, a root node constraint is included to ensure the mapping of node 1 of the tree to the target nonterminal using the function ty. The variable i represents vertices $v \in V$. Using this universal quantified variable i and its translated children, the function `Translate_Combinator` translates all arguments in a rule beginning with the last one in the $args$ list because it is the rightmost child in the inhabitation tree (lines 14–18). In line 19, the $currentAddress$ is the leftmost child and corresponds to the combinator in the current right-hand side.

Using the commands presented in Section 2.7.1, we declare the functions that ensure that tree grammar is represented. The functions `inhabitant` and `hasType` (see Listing 3.1) receive an integer and return an integer that represents the encoding terminals and nonterminals, respectively. The functions `lChild` and `rChild`, which receive arguments `i` of sort integer and return a sum of sort integer, correspond to the right and left child in an inhabitation tree, while `isAppNode` corresponds to an application nodes denoted by @ (cf. Figure 3.3).

```
1    (declare-fun inhabitant (Int) Int)
2    (declare-fun hasType (Int) Int)
3    (define-fun lChild ((i Int)) Int (+ i i))
4    (define-fun rChild ((i Int)) Int (+ 1 (+ i i)))
5    (define-fun isAppNode ((i Int)) Bool (= (inhabitant i) 0))
```

Listing 3.1: Declaration of identifiers

After these commands, we insert the declaration of the current tree grammar translated by Algorithm 3. Listing 3.2 shows the commands that encode the following tree grammar:

$$\begin{aligned} \mathcal{G}_s = \{ & S \mapsto @(A,B), \\ & S \mapsto c, \\ & A \mapsto c_1, \\ & B \mapsto c_2 \} \end{aligned}$$

```
1 (define-fun tConstr0() Bool (forall ((i Int)) (ite (=(hasType i)
   0)(xor (and (isAppNode i) (=(hasType (rChild i)) 2) (=(hasType (
   lChild i)) 1))(=(inhabitant i) 1)) true)))
2 (define-fun tConstr1() Bool (forall ((i Int)) (ite (=(hasType i)
   1)(=(inhabitant i) 2) true)))
```

```
3 (define-fun tConstr2() Bool (forall ((i Int))(ite (=(hasType i) 2)
    (=(inhabitant i) 3) true)))
```

Listing 3.2: Declaration of tree structure

The first declaration represents the first two rules from $\mathcal{G}_S$, where the left-hand side, the nonterminal S, is encoded by 0 using the function `hasType` (cf. Listing 3.1, line 2): (= ($hasType\ i$) 0), and where i is a universal quantified variable. As can be seen, the function `xor` is used for the representation to ensure the usage of only one right-hand side. The translation is straightforward, thus nonterminals A and B are encoded by 1 and 2 and combinators c, c_1 and c_2, by 1,2 and 3. In the first rule, A and B are the left and the right child of the application node, so they are encoded by (= ($hasType(lChild\ i)$) 1) and (= ($hasType(rChild\ i)$) 2), respectively. The translation of the combinators is represented using the function `inhabitant` (cf. Listing 3.1, line 1).

The encoded tree grammar and domain-specific restrictions represented by constraints complying with the SMT-LIB standard must be proved by an SMT solver. We define the following assertions that check whether the stated translations of the rules presented in Listing 3.2 are true:

```
(assert tConstr0)
(assert tConstr1)
(assert tConstr2)
```

If the formulas are satisfiable, the defined functions are true, and the encoding is correct. Using this command, the solver also proves the satisfiability of the defined domain-specific constraints.

Let us consider the example presented in [85] that represents a repository for the synthesis of sort programs. For the synthesis, the small repository Γ_S is used (as shown in Figure 3.4).

$$
\begin{aligned}
\Gamma_s = \{\ & default : double, \\
& id : \alpha \rightarrow \alpha, \\
& min : double \rightarrow SortedList(double) \rightarrow minimal \cap double, \\
& values : List(double), \\
& inv : double \rightarrow double, \\
& sortmap : (\alpha \rightarrow \alpha) \rightarrow List(\alpha) \rightarrow SortedList(\alpha)\ \}
\end{aligned}
$$

Figure 3.4: Repository for sorting of lists

name	id
default	1
id	2
min	3
values	4
inv	5
sortmap	6

Table 3.1: Encoding of the combinators

The first combinator $sortmap$ applies a function to each element, and thereafter, a sorting of the list is performed. The combinator id returns its argument unchanged. The inv combinator represents the inverse function $inv(x) = 1/x$, which can be applied to double values.

The result type of the min combinator is an intersection of $minimal$ and $double$. This combinator will be applied if we want to sort a list with elements of type $double$ and find the minimal values of the elements. The $default$ combinator typed by $double$ returns the default value if a list is empty. To sort a $double$ list and to find its minimal value, the following request is required:

$$\Gamma_s \vdash ? : minimal \cap double.$$

In this case, we get the tree grammar $\mathcal{G}_s$ that computes an infinite set $\mathbb{A}$ of terms. The tree grammar is presented in Figure 3.5. As can be seen in the $double$ rule, the reason for the infinite number of solutions is that combinators id and inv can be applied on arguments of type $double$.

The application of combinators inv and id is not restricted, so these combinators lead to the computation of an infinite set of terms. To avoid the production of trivial terms, we define a constraint (see Listing 3.3) allowing the usage of these combinators only as arguments (lines 1 and 2), and the combinator min must have a terminal as the first argument (lines 3–8).

$$\begin{aligned}\mathcal{G}_s = \{&SortedList(double) \mapsto \{sortmap(double \to double, List(double))\},\\ &minimal \cap double \mapsto \{id(minimal \cap double), min(double, SortedList(double))\},\\ &double \mapsto \{id(double), default(), inv(double), min(double, SortedList(double))\},\\ &double \to double \mapsto \{id(), inv()\}\\ &List(double) \mapsto \{id(List(double)), values()\}\}\end{aligned}$$

Figure 3.5: Tree grammar for sorting of lists

```
1 (assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 2))))
2 (assert (forall ((i Int)) (not (= (inhabitant (leftChild i)) 5))))
3 (assert (forall ((i Int))
4             (ite (= (inhabitant (leftChild i)) 3)
5                  (not (= (inhabitant (rightChild i)) 0))
6                  true)
7         )
8 )
```

Listing 3.3: Constraint example for restriction of trivial solutions

Here, the combinator id is encoded as 2, min as 3, and inv as 5. These constraints lead to the generation of the following terms that are relevant to the use case:

$$((min\ default)\ ((sortmap\ inv)\ values)) \text{ and}$$
$$((min\ default)\ ((sortmap\ id)\ values)).$$

The encoding of the combinators as constraints corresponds to that in the tree grammar translation (see Table 3.1). Figures 3.6 and 3.7 present the terms as trees. The used labelling pattern is $combinatorname : (vertex\ id, combinator\ id)$, where the values of $combinator\ id$ appear in Table 3.1.

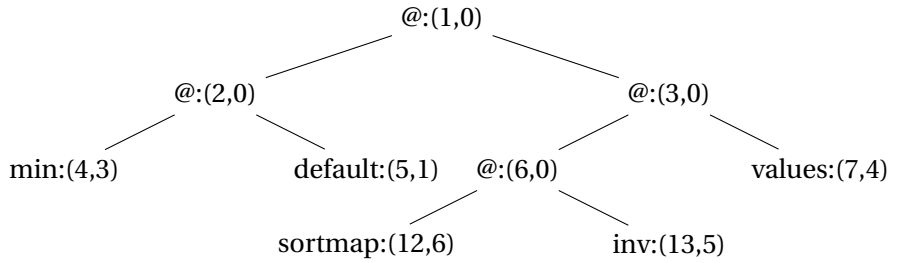

Figure 3.6: Visual representation of term $((min\ default)\ ((sortmap\ inv)\ values))$

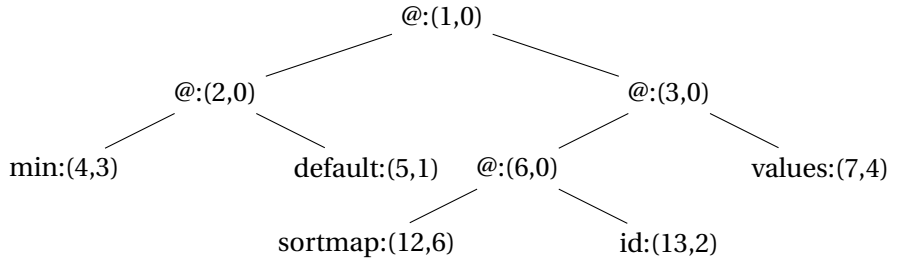

Figure 3.7: Visual representation of term $((min\ default)\ ((sortmap\ id)\ values))$

Considering a small labyrinth example similar to that presented in Section 4.3, The (CL)S Framework synthesise all possible paths, so the following terms are also possible solutions:

$$up(right(up(down(up(down(up(start)))))))$$
$$down(up(up(right(up(start)))))$$
$$up(down(up(down(up(right(up(start)))))))$$
$$...$$

These terms represent trivial and inefficient solutions. With user specification, we can restrict inhabitants that include cycles, for instance. One example of an inefficient term can be seen in the unitary movements to the right, then to the left and back to the right. This case can be restricted by the definition of constraints that forbid the following order of combinators *right(left(...))*. When the combinator *down* is used in the next step, the usage of combinator *up* must be forbidden. Listing 3.4 displays an example of the SMT formula for filtering inhabitants by restriction of certain order. Here, the combinator *right* is encoded by 1, and *left* by 2. The (CL)S-SMT approach seek solutions that exclude this order of combinators.

```
(assert
    (forall ((i Int))
        (not (and (= (inhabitant (leftChild i)) 1)
                  (= (inhabitant (leftChild(rightChild i))) 2)
             )
        )
    )
)
```

Listing 3.4: Filtering by the order of combinators

3.1.3 Limitations

Throughout the representation of inhabitation results and constraints as SMT formulas, a restriction of solutions can be achieved. The synthesis based on combinatory logic reduces the search space of the SMT solvers. Moreover, this approach makes it unnecessary to change the defined inhabitation specifications. Nevertheless, the approach also carries certain disadvantages. The SMT-encoding process rests on constraints expressed as universal quantified assertions. Thereby, the SMT encoding requires the instantiation of quantifiers. Dealing with high numbers of quantifiers and complicated instantiation terms remains difficult for SMT solvers, even though the community has been continuously improving solutions to this problem. Experiments show that the formal setup may be so powerful that the SMT solver is overstrained because there are assertions that cannot practically be decided (while the underlying logic is theoretically decidable). In such a case, the solver produces the value *unknown* as an answer so that we can say that SMT filtering approach works unreliably.

We applied a labyrinth example to investigate this approach. The results show this approach to perform well with small use-cases resulting in tree grammars with a manageable number of rules [85]. For the synthesis of paths in a 3×4 labyrinth, the framework quickly computes a result. As the investigation shows, a significant disadvantage of the filtering approach is that it is unreliable and not well suited for movement plans larger than a size of 10×10. The more complex the use case, the more time necessary to check satisfiability. Moreover, very often the answer is *unknown*. As mentioned, the application of the synthesis to real problems must handle repositories with hundreds of combinators. Consequently, the filtering takes much time, and the probability of finding no solution is relatively high. Bessai et al. [39] also investigated the (CL)S-SMT approach using the construction of sorting functions and the filtering of redundant paths in a labyrinth considered in this section. The authors showed that the presented Haskell implementation can check emptiness and finiteness in the linear case, and this represents an improvement of the filtering approach based on SMT. In summary, these disadvantages indicate that the (CL)S-SMT approach is not suitable for fail-safe and fast filtering user support that can be provided by the (CL)S IDE. For this reason, we consider two other approaches, as presented and discussed in the following.

3.2 Filtering with Recursion Based on Tree Grammar Modification

Because of the drawbacks of the CLS-SMT approach, we developed another filtering algorithm based on modifying the tree grammars generated by the (CL)S Framework. The filtering approach presented in this section is achieved through a recursive modification of the production rules so that after the application, no words containing a given pattern can be derived for the given start symbol.

A domain-specific pattern can also be considered as a constraint restricting the synthesised program's specific output behaviour. Formally, we define patterns as follows:

Definition 14 (Pattern) For set Pat and $p, p_1, p_2 \in Pat$, we define

$$p, p_1, p_2 ::= * \mid c \mid @(p_1, p_2),$$

where p_1 and p_2 are subpatterns of the original pattern p. Any application of term or combinator is denoted by $*$. A named component or combinator is represented by c. The expression $@(p_1, p_2)$ represents an application of p_1 to p_2. □

The size of the pattern p, denoted by $size(p)$, is defined as follows:

$$Size(p) = \begin{cases} 1, & \text{if} p \in \mathbf{B} \vee p = *, \\ 1 + Size(p_i), & i \in \{1, \ldots, n\} \end{cases}$$

3.2.1 Filtering Approach

One of the input values of the filtering algorithm is an unranked tree grammar (cf. Definition 5). Based on a given pattern, the implementation results in an unranked tree grammar that cannot derive terms that match this pattern. Applying the algorithm to each production rule makes it search recursively for matches. If there is a match to the pattern, a modification of the rule follows. Beyond modifying the original production rule, the algorithm computes new nonterminals and inserts new rules depending on the match. Each new nonterminal becomes a fresh name. Therefore, additional functionality is required to generate the new names and to ensure their uniqueness. Here, production rules that do not match the pattern occur unchanged in the new tree grammar. The possible behaviour steps of the algorithm are presented as follows:

- Modification of a production rule with a combinator with $arity = 0$ on the right-hand side that matches the pattern. The nonterminal receives a fresh name, and the matched combinator does not occur in the new tree grammar.

- Modification of a production rule with a right-hand side that includes more than one combinator, where at least one matches the pattern. The new rule contains combinators that do not match. The notation of these combinators remains unchanged. The matching combinator does not occur in the new rule.

- Generation of a new rule with a fresh nonterminal and a modified right-hand side. If the matched combinator has an $arity > 0$, its arguments get fresh names. The algorithm searches for matches in the rules of the arguments recursively. For each argument with a fresh name, the algorithm inserts a new rule according to the cases above.

- Production rules that do not match the given pattern are assumed unchanged in the new tree grammar.

To explain the approach, we consider the following example:

Example 1 Given a tree grammar $\mathcal{G}_r$ with the following rules:

$$\begin{aligned}\mathcal{G}_r = \{S &\mapsto \{c(A,B), c_2()\},\\ A &\mapsto \{c(A,B), c_1()\},\\ B &\mapsto \{c_2()\}\}\end{aligned}$$

and $p := c(c_1, c_2)$. The algorithm considers the right-hand sides. The new grammar after the first iteration is shown in the following:

$$\begin{aligned}\mathcal{G}'_r = \{S &\mapsto \{c(A',B), c(A,B'), c_2()\},\\ A &\mapsto \{c(A'',B), c(A,B''), c_1()\},\\ B &\mapsto \{c_2()\},\\ A' &\mapsto \{c(A,B)\},\\ B' &\mapsto \{\},\\ A'' &\mapsto \{c(A,B)\},\\ B'' &\mapsto \{\}\\ \}\end{aligned}$$

As can be seen by application of rules A' and A'', the pattern can be constructed. If the algorithm generates new rules in some iteration, a next iteration follows. In this case, the second iteration results in the following tree grammar:

$$
\begin{aligned}
\mathscr{G}_r'' = \{ & S \mapsto \{c(A', B), c(A, B'), c_2()\}, \\
& A \mapsto \{c(A'', B), c(A, B''), c_1()\}, \\
& B \mapsto \{c_2()\} \\
& A' \mapsto \{c(A''', B), c(A, B''')\}, \\
& B' \mapsto \{\}, \\
& A'' \mapsto \{c(A'''', B), c(A, B'''')\}, \\
& B'' \mapsto \{\}, \\
& A''' \mapsto \{c(A'', B), c(A, B'')\}, \\
& B''' \mapsto \{\}, \\
& A'''' \mapsto \{c(A'', B), c(A, B'')\}, \\
& B'''' \mapsto \{\} \\
\}
\end{aligned}
$$

The modified tree grammar contains rules (for example A'' and A'''') that cause cycles. Nevertheless, the inhabitation ceases because new rules without further recursive targets exist. The filtering algorithm stops when the new grammar does not receive new entries in the final iteration. Demonstrably, each rule leads to another rule that does not match the pattern (i.e. a rewritten rule or an empty rule). □

One drawback of the algorithm is that its termination cannot be easily shown because of the recursion. Furthermore, the algorithm cannot handle the ω case because of the application of the tree grammar presented in Definition 5. Moreover, the dependence on the function that creates new and unique names is necessary. Otherwise, the algorithm leads to illegal solutions.

3.3 Filtering without Recursion Based on Tree Grammar Modification

To compensate for the disadvantages of the first version of the filtering approach presented in Section 3.2, we developed another algorithm. It performs similarly to that with recursion, but it allows one to reason about its correctness more easily. A detailed investigation of the performance of the filtering methods is presented in Section 5.1. Moreover, it is not required to control the uniqueness of the names of the modified rules.

3.3.1 Filtering Approach

The algorithm presented in this section modifies the applicative tree grammar (cf. Definition 7) so that it also behaves well with ω. To specify this filtering approach, a definition of the pattern language is necessary.

Definition 15 (Pattern Language) Let M and N be given terms, such that $M, N \in \mathbb{A}$ and c is a combinator. We define the language of pattern $\mathscr{L}'(p)$ as the least set closed under the following rules:

$$\frac{}{c \in \mathscr{L}'(*)} \qquad \frac{}{c \in \mathscr{L}'(c)}$$

$$\frac{}{@(M,N) \in \mathscr{L}'(*)} \qquad \frac{M \in \mathscr{L}'(p_1) \qquad N \in \mathscr{L}'(p_2)}{@(M,N) \in \mathscr{L}'(@(p_1,p_2))}$$

We define a language $\mathscr{L}(p)$ as a set closed under the following rules:

$$\frac{M \in \mathscr{L}'(p)}{M \in \mathscr{L}(p)} \qquad \frac{M \in \mathscr{L}(p)}{@(M,N) \in \mathscr{L}(p)} \qquad \frac{N \in \mathscr{L}(p)}{@(M,N) \in \mathscr{L}(p)}$$

These rules allow for the occurrence of a given pattern p not only at the beginning of a word but also deeper in the tree. From now on, to increase readability, we consider the language $\mathscr{L}(p)$ as a set of all words, where pattern p occurs anywhere in the word and not only at the beginning. □

The language complements are essential to the filtering approach. We define the complement language of $\mathscr{L}(p)$ according to [79] as follows:

Definition 16 (Complement Languages) If $\mathscr{L}(p)$ is a regular language over alphabet Σ, then $\overline{\mathscr{L}}(p)$, the complement of $\mathscr{L}(p)$, is the set of all words in Σ^* that are not in $\mathscr{L}(p)$; accordingly, we have $\overline{\mathscr{L}}(p) = \Sigma^* - \mathscr{L}(p)$ that is also a regular language. The complement language $\overline{\mathscr{L}}(p)$ of the language $\mathscr{L}(p)$ is the least set closed under the following rules:

$$\frac{M \in \Sigma^* \qquad M \notin \mathscr{L}(p)}{M \in \overline{\mathscr{L}}(p)}$$

According to the closure properties of the regular languages, if languages $\mathscr{L}(\mathscr{G})$ and $\mathscr{L}(p)$ are regular languages, then the difference $\mathscr{L}(\mathscr{G}) - \mathscr{L}(p)$ is also a regular language [79]. We define the difference of the language of a regular tree grammar and the language of a pattern using complement languages as follows:

Definition 17 (Difference of Languages) Let $\mathscr{L}(p!\mathscr{G})$ be a set of terms that is in language $\mathscr{L}(\mathscr{G})$, where $\mathscr{G}$ is a tree grammar, but not in the language that constructs a pattern p: $\mathscr{L}(p)$. Language $\mathscr{L}(p!\mathscr{G})$ is then the difference of $\mathscr{L}(\mathscr{G})$ and $\mathscr{L}(p)$ denoted by $\mathscr{L}(\mathscr{G}) - \mathscr{L}(p)$ or $\mathscr{L}(\mathscr{G}) \cap \overline{\mathscr{L}}(p)$, where $\overline{\mathscr{L}}(p)$ is the complement language of $\mathscr{L}(p)$. The intersection of these languages is the least set closed under the following rules:

$$\frac{c \in \mathscr{L}(\mathscr{G}) \qquad c \in \overline{\mathscr{L}}(p)}{c \in \mathscr{L}(p!\mathscr{G})} \qquad\qquad \frac{@(M,N) \in \mathscr{L}(\mathscr{G}) \qquad @(M,N) \in \overline{\mathscr{L}}(p)}{@(M,N) \in \mathscr{L}(p!\mathscr{G})}$$

Therefore, a word w is in $\mathscr{L}(p!\mathscr{G})$ and does not include p if $w \in \mathscr{L}(\mathscr{G})$ and $w \notin \mathscr{L}(p)$. □

The union of the languages of all trees rooted in nonterminal $A \in \mathscr{N}$ where the pattern p is forbidden is represented by $\bigcup_{A \in \mathscr{N}} \mathscr{L}_A(p!\mathscr{G})$. This union corresponds to the intersection of the language of the tree grammar $\mathscr{G}$ with the complement language of the given pattern $\overline{\mathscr{L}}(p)$. That is, it corresponds to: $\bigcup_{A \in \mathscr{N}} \mathscr{L}_A(p!\mathscr{G}) = \mathscr{L}(\mathscr{G}) \cap \overline{\mathscr{L}}(p)$.

Definition 18 (Filter) For applicative tree grammar $\mathscr{G}$ and pattern $p \in Pat$ define

- a rule in an applicative tree grammar:

  $$R_T \ni r ::= A \mapsto c \mid A \mapsto @(B,C)$$

  for $c \in \mathbf{B}$, $A, B, C \in T$, where T is a set and $\mathbf{B}$ is a finite set of combinators.

- the set of all subpatterns $p_1, \ldots p_k$ with $k \geq 0$ of pattern p:

  $$p_1, \ldots p_k \in S(p) \text{ where} p_i \neq p \text{ with } 0 \leq i \leq k$$

- a power set of a set of all subpatterns $S(p)$ as the set of all subsets of $S(p)$:

$$\mathscr{P}(S(p)) = \{S \mid S \subseteq S(p)\}$$

- a set P as an element of a set U where:

$$U = \{sp \cup \{p\} \mid sp \in \mathscr{P}(S(p))\}$$

- function FORBID: $(\mathbb{R}^*_T \times P) \rightarrow \mathbb{R}^*_{T \cup (U \times T)}$

$$\text{forbid}(\mathscr{G},\ p) = \begin{cases} [\,r \mid sp \in \mathscr{P}(S(p)), & \\ \quad r \leftarrow \text{forbidIn}(\mathscr{G}, p, sp)\,] & \text{for } \mathscr{G} \neq \emptyset \\ [::] & \text{otherwise} \end{cases}.$$

- function FORBID_IN: $(\mathbb{R}^*_T \times Pat \times S) \rightarrow \mathbb{R}^*_{T \cup (S \times T)}$

$$\text{forbidIn}(\mathscr{G}, \text{p, sp}) =$$
$$\begin{cases} \textcircled{1} \begin{cases} \text{forbidIn}(\mathscr{G}_{rest}, p, sp) & c \in P \\ [::] & \text{for } * \in P \\ [:: P!A \mapsto c \,\&\, \text{forbidIn}(\mathscr{G}_{rest}, p, sp)] & \text{otherwise} \end{cases} \\ \quad \text{for } P = \{p\} \cup sp \text{ and } \mathscr{G} = [:: A \mapsto c \,\&\, \mathscr{G}_{rest}] \\ \textcircled{2} \begin{cases} ([:: P!A \mapsto @(\{P \cup & \\ \quad \{p_{11}, p_{12}, \ldots, p_{1n}\}\}!B,\ P!C)\ \& & \\ \quad P!A \mapsto @(P!B,\ \{P \cup \{p_{21}, p_{22}, \ldots, p_{2n}\}\}!C)\ \& & \\ \quad \text{forbidIn}(\mathscr{G}_{rest}, p, sp))] & \text{for } P = \{c_0, \ldots, c_k, @(p_{11}, p_{21}), \\ & \quad @(p_{12}, p_{22}), \ldots, @(p_{1n}, p_{2n})\} \\ & \quad \text{with } k \in \mathbb{N}_0 \text{ and } n \in \mathbb{N} \\ [::] & \text{for } * \in P \\ [:: P!A \mapsto @(P!B,\ P!C)\ \& & \\ \quad \text{forbidIn}(\mathscr{G}_{rest}, p, sp)] & \text{otherwise} \end{cases} \\ \quad \text{for } P = \{p\} \cup sp \text{ and } \mathscr{G} = [:: A \mapsto @(B, C)\ \&\ \mathscr{G}_{rest}] \\ \textcircled{3} \quad [::] \\ \quad \text{for } \mathscr{G} = [::] \end{cases}$$

- possible behaviour steps of the filtering algorithm that are closed under the rules:

$$\frac{c \in P \quad * \notin P}{P!(\mathscr{G}_{checked}, [:: A \mapsto c \,\&\, \mathscr{G}_{rest}]) \rightsquigarrow (\mathscr{G}_{checked}, \mathscr{G}_{rest})}\ (\text{FORBIDCOMB})$$

$$\frac{\text{not exist } \{@(p_1,p_2)\} \in P \qquad * \notin P}{P!(\mathscr{G}_{checked}, [:: A \mapsto @(B,C) \,\&\, \mathscr{G}_{rest}]) \rightsquigarrow ([:: P!A \mapsto @(P!B, P!C) \,\&\, \mathscr{G}_{checked}\,], \mathscr{G}_{rest})} \text{ (PATAPP)}$$

$$\frac{* \in P}{P!(\mathscr{G}_{checked}, [:: r \,\&\, \mathscr{G}_{rest}]) \rightsquigarrow (\mathscr{G}_{checked}, \mathscr{G}_{rest})} \text{ (FORBIDSTAR)}$$

$$\frac{c \notin P \quad * \notin P}{P!(\mathscr{G}_{checked}, [:: A \mapsto c \,\&\, \mathscr{G}_{rest}]) \rightsquigarrow ([:: P!A \mapsto c \,\&\, \mathscr{G}_{checked}\,], \mathscr{G}_{rest})} \text{ (PATCOMB)}$$

$$\frac{\begin{array}{l} P = \{c_0, \ldots, c_k, @(p_{11}, p_{21}),\ @(p_{12}, p_{22}), \ldots, @(p_{1n}, p_{2n})\} \text{ with } k \in \mathbb{N}_0 \text{ and } n \in \mathbb{N} \\ \qquad P!(\mathscr{G}_{checked}, [:: A \mapsto @(B,C) \,\&\, \mathscr{G}_{rest}]) \rightsquigarrow \\ \qquad ([:: P!A \mapsto @(\{P \cup \{p_{11}, p_{12}, \ldots, p_{1n}\}\}!B,\ P!C) \,\& \\ \qquad P!A \mapsto @(P!B,\ \{P \cup \{p_{21}, p_{22}, \ldots, p_{2n}\}\}!C) \,\&\, \mathscr{G}_{checked}\,], \mathscr{G}_{rest}) \end{array}}{} \text{ (FORBIDAPP)}$$

The closure of $\rightsquigarrow$ with n steps is the least relation closed under the following rules:

$$\frac{}{P!(\mathscr{G}_{checked}, \mathscr{G}_{rest}) \rightsquigarrow_0 (\mathscr{G}_{checked}, \mathscr{G}_{rest})}$$

$$\frac{P!(\mathscr{G}'_{checked}, \mathscr{G}'_{rest}) \rightsquigarrow (\mathscr{G}''_{checked}, \mathscr{G}''_{rest}) \qquad P!(\mathscr{G}''_{checked}, \mathscr{G}''_{rest}) \rightsquigarrow_n (\mathscr{G}'''_{checked}, \mathscr{G}'''_{rest})}{P!(\mathscr{G}'_{checked}, \mathscr{G}'_{rest}) \rightsquigarrow_{n+1} (\mathscr{G}'''_{checked}, \mathscr{G}'''_{rest})}$$

Accordingly, the n-step closure of $\mathscr{G}'$ from $\mathscr{G}$ corresponds to the reflexive, transitive closure of $\rightsquigarrow$ denoted by $\rightsquigarrow_*$:

$P!(\mathscr{G}_{checked}, \mathscr{G}_{rest}) \rightsquigarrow_* (\mathscr{G}'_{checked}, \mathscr{G}'_{rest})$ iff there exists a positive integer n, s.t. $P!(\mathscr{G}_{checked}, \mathscr{G}_{rest}) \rightsquigarrow_n (\mathscr{G}'_{checked}, \mathscr{G}'_{rest})$. □

The filtering algorithm computes from a list of production rules representing an applicative tree grammar a new list of rules. In addition to the applicative tree grammar $\mathscr{G}$, the function FORBID gets a user-specified pattern p. The function computes a finite set of subpatterns $S(p)$ based on the given pattern p. Then, the computation of a power set of $S(p)$ follows. It is denoted as $\mathscr{P}(S(p))$ and of a size 2^k, where k is the size of $S(p)$. For example, for a pattern $p = @(@(c, p_1), p_2)$, we get a set of subpatterns $S(p) = \{c, p_1, p_2, @(c, p_1)\}$ and a power set

$$\mathscr{P}(S(p)) = \{\{\}, \{c\}, \{p_1\}, \{p_2\}, \{@(c,p_1)\}, \{c,p_1\}, \{c,p_2\}, \{c,@(c,p_1)\}, \{p_1,p_2\}, \{p_1,@(c,p_1)\}, \{p_2,$$

$$@(c,p_1)\}, \{c,p_1,p_2\}, \{c,p_1,p_2\}, \{c,p_1,@(c,p_1)\}, \{p_1,p_2,@(c,p_1)\}, \{c,p_1,p_2\}, \{c,p_1,p_2,@(c,p_1)\}\}.$$

To forbid the pattern and the subpatterns, the function FORBID_IN executes for each element sp_i of the power set where $1 \leq i \leq |\mathscr{P}(S(p))|$. Furthermore, FORBID_IN is applied to each rule of the applicative tree grammar $\mathscr{G}$. For each element P that represents the union of the original pattern p and an element sp_i, the function returns a new modified applicative tree grammar containing rules that cannot construct the elements of P. The number of calls m of the function is equal to the size of $\mathscr{P}(S(p))$ that also corresponds to the size of U. The function results in

$$p!\mathscr{G} := \bigcup_{P \in U} P!\mathscr{G},$$

where $p!\mathscr{G}$ denotes that the original pattern p is forbidden in $\mathscr{G}$ and $P!\mathscr{G}$ denotes that all elements on the left-hand side of the exclamation mark are forbidden in $\mathscr{G}$. The function FORBID returns a union of all applicative tree grammars modified according to the current P. For each rule in the grammar, it is necessary to forbid not only the set of subpatterns sp but also the given original pattern p, for the following reason. Suppose the algorithm forbade the pattern and the subpattern separately. In this case, the union of the generated applicative tree grammars can synthesise terms containing the pattern p deeper in the tree. Moreover, this approach ensures that all terms that do not contain the given pattern remain contained in the language.

The function FORBID_IN iterates over the rules in $\mathscr{G}$ and searches for matches. We consider the cases under ①, where the current rule has a form of $A \mapsto c$ and where there is a match. This case occurs, on the one hand, when $p = c$. Then, $S(p)$ is empty and P contains only the element $\{c\}$. On the other hand, $c \in P$, $* \notin P$, but $p \neq c$. In that case, the algorithm filters this rule out and the algorithm continues with $\mathscr{G}_{rest}$. The result grows only with the addition of newly modified rules. The list $\mathscr{G}_{rest}$ contains the rules that must be considered, and in each step, the algorithm drops the considered rule.

When input grammar becomes empty, meaning no further rules must be considered (case ③), the algorithm stops and returns the completed list of modified rules. If $p = *$ or $* \in P$ – for example, if $p = @(c, *)$ – the function returns an empty list because all kinds of rules match with $*$ and must be forbidden. In the case that we have $p \neq c$ and $* \notin P$, or $c \notin P$ and $* \notin P$, the algorithm rewrites the left-hand side of the rule by $P!A$. The right-hand side remains unchanged. The function then adds the new rule $P!A \mapsto c$ and continues with $\mathscr{G}_{rest}$.

If the rule is an application rule of form $A \mapsto @(B, C)$ and the set P contains a subpattern of the forms $@(p_1, p_2)$ and $* \notin P$, the algorithm computes two new rules (see the cases under ②). In this case, it does not matter whether a subpattern of the form c is in the set P because the rule does not match such patterns anyway. The new apply rules are modified according to the rules presented above. On the one hand, we must forbid the left side of the application rule. The name of the function type in the first rule centres on P and p_1 because it can match only left sides in an apply pattern, and in this way, the algorithm recursively forbids the pattern. The argument type name is composed of its name and the set P. On the

other hand, in the second rule, the algorithm modifies the argument type according to the set P and the subpattern p_2 to recursively forbid the pattern. As in the first rule, the original name is also considered. The modified target name is based on the original left-hand side and the current set P. The added rules are of the form $P!A \mapsto @(\{P, p_1\}!B,\ P!C)$ and $P!A \mapsto @(P!B,\ \{P, p_2\}!C)$. Even if P includes more than one apply pattern or combinators, such that $P = \{c_1, \ldots, c_k, @(p_{11}, p_{21}),\ @(p_{12}, p_{22}), \ldots, @(p_{1n}, p_{2n})\}$ with $k \in \mathbb{N}_0$ and $n \in \mathbb{N}$, the filtering algorithm considers separately the left- and right-hand sides of the patterns. It modifies the rule as follows: $P!A \mapsto @(\{P \cup \{p_{11}, p_{12}, \ldots, p_{1n}\}\}!B,\ P!C)$ & $P!A \mapsto @(P!B,\ \{P \cup \{p_{21}, p_{22}, \ldots, p_{2n}\}\}!C)$. In this manner, the algorithm forbids only the pattern p and allows the production of all terms resulting from the original grammar, which do not include the pattern. A modification does not occur if $*$-pattern is in the set P. In this case, the algorithm returns an empty list regardless of the current rule. If there is no pattern of the form $@(p_1, p_2)$ in P, the algorithm rewrites the apply rule by changing the names, similar to the first case; s.t. the new rule is of the form $P!A \mapsto @(P!B,\ P!C)$. From now on, for the sake of readability, we denote pattern set with extensions by $P \cup ext$.

Lemma 1 (Functionality of the Filtering Algorithm) *For all inputs $P, \mathcal{G}_{checked}$, and $\mathcal{G}_{rest}$, there is at most one $\mathcal{G}'_{checked}$ and $\mathcal{G}'_{rest}$ that results from $P!(\mathcal{G}_{checked}, \mathcal{G}_{rest}) \rightsquigarrow (\mathcal{G}'_{checked}, \mathcal{G}'_{rest})$.* □

PROOF We show that if the relation with input $P, \mathcal{G}_{checked}$, and $\mathcal{G}_{rest}$ steps to $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$ and $(\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$, then $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$ and $(\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ are equal, by case distinction on a derivation of step $P!(\mathcal{G}_{checked}, \mathcal{G}_{rest}) \rightsquigarrow (\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$. We assume that the induction hypothesis holds for any subderivation, s.t. the cases that must be considered depend on the last rule used in $P!(\mathcal{G}_{checked}, \mathcal{G}_{rest}) \rightsquigarrow (\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$ and the last rule in $P!(\mathcal{G}_{checked}, \mathcal{G}_{rest}) \rightsquigarrow (\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$. We consider the following cases:

- If $\mathcal{G}$ is empty, then $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$ and $(\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ are also empty, and the induction hypothesis holds.
- If $* \in P$, then $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1})$ and $(\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ are empty because the derivations end with FORBIDSTAR, so that the result holds.
- If $c \in P$, $* \notin P$, there does not exist p_1 and p_2 s.t. $@(p_1, p_2) \in P$, then if the current rule is of the form $A \mapsto @(B, C)$, the only possible step is PATAPP. If the rule is of the form $A \mapsto c$, then it does not matter whether $@(p_1, p_2)$ is in P, and the derivation ends with the rule FORBIDCOMB. In this case, the statement $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1}) = (\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ is true if and only if the derivations end with PATAPP or FORBIDCOMB.
- If $c \notin P$ and $* \notin P$ and if the current rule is of the form $A \mapsto c$, then it does not matter whether there exist p_1 and p_2 s.t. $@(p_1, p_2) \in P$, and the derivation ends with PATCOMB, such that the result holds.
- If there exist p_1 and p_2 so that $@(p_1, p_2) \in P$, $c \notin P$, and $* \notin P$, then for a rule of the form $A \mapsto c$, the only possible step is PATCOMB. If the rule is of the form $A \mapsto @(B, C)$, then

FORBIDAPP is the only step applicable for this input. It follows that $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1}) = (\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ is satisfied, if the rule PATCOMB or FORBIDAPP is used.

- Analogously, if P contains also a pattern of the form c, such that there exist p_1 and thus p_2 so that $@(p_1, p_2) \in P$, $c \in P$, and $* \notin P$, then only the rules FORBIDAPP and FORBIDCOMB can be applied, according to the input rules. If the rule is of the form $A \mapsto c$, then FORBIDCOMB is the only possible step, and if the rule is of the form $A \mapsto @(B, C)$ then only the application of FORBIDAPP can occur. Thus, $(\mathcal{G}'_{checked1}, \mathcal{G}'_{rest1}) = (\mathcal{G}'_{checked2}, \mathcal{G}'_{rest2})$ is satisfied if and only if the derivations end with FORBIDCOMB or FORBIDAPP.

As the cases above show, the rules presented in Definition 18 are mutually exclusive. By induction, the one-step relation only allows unique results. Therefore, it is true that transitive closure yields only unique results. ■

Lemma 2 (Termination of the Filtering Algorithm) *For all P, $\mathcal{G}_{rest}$ exist $\mathcal{G}_{checked}$ and n so that*

$$P!([::], \mathcal{G}_{rest}) \rightsquigarrow_n (\mathcal{G}_{checked}, [::]).$$ □

PROOF We prove the termination of the filtering algorithm by analysing all the possible inputs and steps. The growth of $\mathcal{G}_{checked}$ is limited by the size of $\mathcal{G}_{rest}$. We know that $\mathcal{G}_{rest}$ is finite. In every possible relation step, a rule is removed from $\mathcal{G}_{rest}$ so that $n = |\mathcal{G}_{rest}|$. ■

Definition 19 (Open References) Nonterminals on the right-hand side of production rules in a given tree grammar $\mathcal{G}$ are called open references if these nonterminals do not occur on the left-hand side of a rule in $\mathcal{G}$. □

Definition 20 (Compatibility of Nonterminals) Let P, $P \cup ext_1$, $P \cup ext_2$, ... $P \cup ext_n$ be elements of the set U computed by the filtering algorithm then all nonterminals $P!A$, $P \cup ext_1!A, P \cup ext_2!A, \ldots P \cup ext_n!A$ in $\mathcal{G}_{checked}$ and $\mathcal{G}'$ where $P!(\mathcal{G}_{checked}, \mathcal{G}) \rightsquigarrow_1 (\mathcal{G}_{checked} + +\mathcal{G}', \mathcal{G}_{rest})$ are compatible if:

- The pattern set P is either equal to pattern sets used on the right-hand side in $\mathcal{G}'$ or they are true supersets.
- Open references in $\mathcal{G}_{checked}$ are either with extensions so that they cannot be closed and applied with the new rules in $\mathcal{G}'$ or there is a left-hand side $\mathcal{G}'$ that cannot be used for the construction of forbidden words. □

Definition 21 (Stable Grammars) Let $P!A$, $P \cup ext_1!A, P \cup ext_2!A$, ..., $P \cup ext_n!A$ be nonterminals that occur on the right-hand side of the rules in the grammar $\mathcal{G}_{checked} + +\mathcal{G}'$ then the grammar $\mathcal{G}_{checked} + +\mathcal{G}'$ where $P!(\mathcal{G}_{checked}, \mathcal{G}) \rightsquigarrow_1 (\mathcal{G}_{checked} + +\mathcal{G}', \mathcal{G}_{rest})$ is stable if the following conditions are true:

- The production rules in $\mathcal{G}_{checked}$ cannot construct forbidden words, which are in the set P.
- Open references in apply rules in $\mathcal{G}_{checked}$ are compatible with the nonterminals in $\mathcal{G}'$.

□

Definition 22 (Addable Grammars) Let $P!A, P \cup ext_1!A, P \cup ext_2!A, \ldots, P \cup ext_n!A$ be nonterminals that occur on the right-hand side in $\mathcal{G}'$ then the grammar $\mathcal{G}'$ can be added to the grammar $\mathcal{G}_{checked}$, $P!(\mathcal{G}_{checked}, \mathcal{G}) \rightsquigarrow_1 (\mathcal{G}_{checked} + + \mathcal{G}', \mathcal{G}_{rest})$, where $\mathcal{G}_{checked}$ is a stable grammar (cf. Definition 21) if the following conditions are true:

- The nonterminals $P \cup ext_1!A, P \cup ext_2!A, \ldots, P \cup ext_n!A$ in $\mathcal{G}'$ cannot be used for the construction of words using rules from $\mathcal{G}_{checked}$. In other words, the new nonterminals with extensions are unproductive because they do not have reverse references in $\mathcal{G}_{checked}$.
- Nonterminals on the right-hand side in $\mathcal{G}'$ are compatible with the nonterminals in $\mathcal{G}_{checked}$ and they cannot be used for the construction of forbidden words. □

Definition 23 (Union Condition) The union condition of grammar $\mathcal{G}_{checked}$, $C_u(\mathcal{G}_{checked})$ is defined as follows: for all $\mathcal{G}'$ if $\mathcal{G}'$ is an addable grammar then the union $\mathcal{G}_{checked} + + \mathcal{G}'$ is stable. □

Lemma 3 *For all $\mathcal{G}_{checked}$, $\mathcal{G}'$, P*

$$\textit{if } P!(\mathcal{G}_{checked}, \mathcal{G}) \rightsquigarrow_1 (\mathcal{G}_{checked} + + \mathcal{G}', \mathcal{G}_{rest}) \textit{ then } \mathcal{G}' \textit{ is an addable grammar.}$$ □

PROOF According to the filtering rules, $\mathcal{G}'$ can only contain rules of the form $P!A \mapsto c$, $P!A \mapsto @(P!B, P!C)$, or $P!A \mapsto @(P!B, P \cup ext_2!C)$, $P!A \mapsto @(P \cup ext_1!B, P!C)$, where ext_1 and ext_2 are elements of the power set $\mathcal{P}(S(p))$ and are subsets of P. The rules are computed according to the filtering rules PATCOMB, PATAPP, and FORBIDAPP so that we have the following three cases:

1. The rules in $\mathcal{G}'$ are computed according to the filtering rule FORBIDAPP: the new production rule is of the form $P!A \mapsto @(P \cup ext_1!B, P \cup ext_2!C)$. The pattern sets are $P \cup ext_1 \neq P$ and $P \cup ext_2 \neq P$ (if $P \cup ext_1 = P$ or $P \cup ext_2 = P$, then we have the second case, presented in the following) so that they cannot refer to rules in $\mathcal{G}_{checked}$ because the pattern set P is a subset of pattern sets with extensions $P \cup ext_1, P \cup ext_2 \ldots P \cup ext_n$. In other words, there is no nonterminal on the left-hand side in $\mathcal{G}_{checked}$ with the aforementioned pattern sets that can be used for the construction of words, and the new rules are unproductive. Open references in $\mathcal{G}_{checked}$ either cannot be closed, or they cannot construct forbidden words so that the nonterminals are compatible.

2. The rule in $\mathcal{G}'$ is computed according to the filtering rule PATAPP: the new production rule is of the form $P!A \mapsto @(P!B, P!C)$. If PATAPP was used, there is no apply construction in P; therefore, the new rule cannot be used to construct a word that is in P.

3. The rule in $\mathcal{G}'$ is computed according to the filtering rule PATCOMB: the new production rule is of the form $P!A \mapsto c$. There are no nonterminals on the right-hand side of the rule, so there are no references to rules in $\mathcal{G}_{checked}$. The open references in $\mathcal{G}_{checked}$ can use the production rule for the construction of words because c is not in the pattern set P.

The filtering rules FORBIDCOMB and FORBIDSTAR do not add any new rules, so $\mathcal{G}'$ is empty. In this case, there are no nonterminals with references to $\mathcal{G}_{checked}$ and the nonterminals in $\mathcal{G}_{checked}$ still cannot construct words that are forbidden and are still compatible so that the statement is true. ■

Lemma 4 *For all* $\mathcal{G}_{checked}$, $\mathcal{G}'$, P

$$\textit{if } P!(\mathcal{G}_{checked}, \mathcal{G}_{rest}) \rightsquigarrow_1 (\mathcal{G}_{checked} + + \mathcal{G}', \mathcal{G}'_{rest}) \textit{ and the union condition } C_u \textit{ applies for } \mathcal{G}_{checked}, C_u(\mathcal{G}_{checked}) \textit{ then } C_u(\mathcal{G}_{checked} + + \mathcal{G}') \textit{ is true.}$$ □

PROOF The grammar $\mathcal{G}'$ is an addable grammar (direct consequence of Lemma 3) so that the new nonterminals are compatible with these in $\mathcal{G}_{checked}$, and the grammar $\mathcal{G}_{checked} + + \mathcal{G}'$ cannot construct forbidden words that are in the set P. Open references with extensions on the right-hand side in $\mathcal{G}_{checked}$ cannot be closed using the rules in $\mathcal{G}'$. Open references without extensions that can be closed cannot construct forbidden words (cf. Lemma 3). It follows that the union grammar $\mathcal{G}_{checked} + + \mathcal{G}'$ is a stable grammar. Let $\mathcal{G}'_{checked}$ be equal to the union $\mathcal{G}_{checked} + + \mathcal{G}'$ and $\mathcal{G}''$ be an addable grammar where $P!(\mathcal{G}'_{checked}, \mathcal{G}_{rest}) \rightsquigarrow_1 (\mathcal{G}'_{checked} + + \mathcal{G}'', \mathcal{G}'_{rest})$. Open references in $\mathcal{G}'_{checked}$ can be closed using rules from $\mathcal{G}''$. According to Lemma 3, the closed rules cannot construct forbidden words because the rules in $\mathcal{G}''$ are either rules with open references or rules that cannot be used for the construction of words from the set P so that the grammar $\mathcal{G}'_{checked} + + \mathcal{G}''$ is stable. It follows that the grammar $\mathcal{G}_{checked} + + \mathcal{G}' + + \mathcal{G}''$ is a stable grammar so that the union $C_u(\mathcal{G}_{checked} + + \mathcal{G}')$ is true. ■

Lemma 5 *If* $P!([::], \mathcal{G}) \rightsquigarrow_n (\mathcal{G}_{checked} + + \mathcal{G}', [::])$ *then* $C_u(\mathcal{G}_{checked} + + \mathcal{G}')$ *with* $\mathcal{G}' = [::]$ *holds.* □

PROOF Induction of the number of n.

- BASIS STEP: Direct consequence of Lemma 3 and Lemma 4.

- INDUCTION STEP: According to Lemma 3, $\mathcal{G}'$ is an addable grammar. In Lemma 4, we have shown that $C_u(\mathcal{G}_{checked})$ is satisfied; therefore, the induction hypothesis is applicable. It follows that the resulted grammar is stable so that the union condition $C_u(\mathcal{G}_{checked} + + \mathcal{G}')$ holds. ■

Lemma 6 (Filtering Algorithm Soundness) *For all $\mathcal{G}_{checked}$,*

if $P!([::], \mathcal{G}) \rightsquigarrow_n (\mathcal{G}_{checked}, [::])$, then there does not exist a word w so that w is in the language of the grammar $\mathcal{G}_{checked}$, $w \in \mathcal{L}(\mathcal{G}_{checked})$ and in the language of the original pattern p, $w \in \mathcal{L}(p)$. □

PROOF According to Lemma 5, the union condition $C_u(\mathcal{G}_{checked})$ is satisfied; therefore, the resulted grammar $\mathcal{G}_{checked}$ is stable. A stable grammar cannot construct words that are in the set P (cf. Definition 21). It follows that if a word w is in the language $\mathcal{L}(p)$, then it cannot be in the language $\mathcal{L}(\mathcal{G}_{checked})$. ■

Lemma 7 (Filtering Algorithm Completeness) *For all $\mathcal{G}$, $\mathcal{G}_{checked}$, p, and P,*

if there exists a word w so that w is in the language of the original grammar $\mathcal{L}(\mathcal{G})$ but not in the language of the pattern $\mathcal{L}(p)$, then the word w is in the language of the resulting grammar $w \in \mathcal{L}(\mathcal{G}_{checked})$, where $P!([::], \mathcal{G}) \rightsquigarrow_n (\mathcal{G}_{checked}, [::])$. □

PROOF We proceed by induction on the number of steps in a derivation of a word w.

- BASIS STEP: By one-step derivation, only applying rules of the form $A \mapsto c$ can be used, where A is a target symbol and $c \in \mathbb{A}$. In this case, we have $c \in \mathcal{L}(\mathcal{G})$ and $c \notin \mathcal{L}(p)$ so that only the filter rule PATCOMB can be used. It follows that the new rule in $\mathcal{G}_{checked}$ is of the form $P!A \mapsto c$ and the new target $P!A$ so that the word c is in the language $\mathcal{L}(\mathcal{G}_{checked})$.

- INDUCTION HYPOTHESIS: Assume that for every derivation with $n \geq 1$ steps, w is in $\mathcal{L}(\mathcal{G}) - \mathcal{L}(p)$

- INDUCTION STEP: If a word w of the form $@(w_1, w_2)$ is in the language $\mathcal{L}_A(\mathcal{G})$ and not in $\mathcal{L}(p)$, then there are rules of the form $A \mapsto @(B, C)$, $B \mapsto w_1$ and $C \mapsto w_2$ in $\mathcal{G}$. The word w_1 is in the language $\mathcal{L}_B(\mathcal{G})$ and w_2 in $\mathcal{L}_C(\mathcal{G})$. By the induction hypothesis, if a word w_1 is in the language $\mathcal{L}_B(\mathcal{G})$, then w_1 is also in the language $\mathcal{L}_{P_1!B}(\mathcal{G}_{checked})$. Accordingly, $w_2 \in \mathcal{L}_{P_2!C}(\mathcal{G}_{checked})$ is also true. It follows that there are nonterminals of the form $P_1!B$ and $P_2!C$ on the left-hand side of the rules in $\mathcal{G}_{checked}$, where P_1 and P_2 are in the set U, and they can be equal. These nonterminals can be used for the construction of w_1 and w_2. To derive the word w from the rules in $\mathcal{G}_{checked}$, a rule of the form $P!A \mapsto @(P_1!B, P_2!C)$ is required. According to the filtering rule PATAPP, there is such a rule in the grammar $\mathcal{G}_{checked}$. In this case $P = P_1 = P_2$. The rules resulting from the filtering rule FORBIDAPP, can also construct the word w, but in this case $P = P_1$ or $P = P_2$. It follows that the word w is in the language $\mathcal{L}_{P!A}\mathcal{G}_{checked}$.

Considering the cases above, we have shown that $w \in \mathcal{L}(\mathcal{G}) - \mathcal{L}(p)$ if and only if $w \in \mathcal{L}(\mathcal{G}_{checked})$ so that $\forall p : \mathcal{L}(\mathcal{G}) - \mathcal{L}(p) \subseteq \mathcal{L}(\mathcal{G}_{checked})$ is true. ■

The deciding emptiness and finiteness of the solution of the filtering algorithm presented in Definition 18 is equivalent to the linear case introduced by Bessai et al. [39]. The authors have shown that the intersection problem is $EXPTIME$-complete. The filtering algorithm computes a new tree grammar for each element from the power set and the given pattern. An exponential blow-up that occurs depends on the size of the power set $\mathcal{P}(S(p))$. Thus, we have an upper bound equal to $2^{|S(p)|}$. The maximal size of the new grammar $\mathcal{G}'$ is

$$g \cdot |\mathcal{P}(S(p))|$$

where g is the number of rules in the tree grammar $\mathscr{G}$. Usually, the size of the new grammar $\mathscr{G}'$ does not correspond to the maximal size. The reason is that after applying the filtering algorithm, two auxiliary functions are applied to $\mathscr{G}'$ to remove all unproductive and unreachable rules that cannot be used for the construction of words.

The simultaneous restriction of more than one pattern is also possible. According to Comon at al. [50], the class of regular tree languages is closed under union so that we have $\mathscr{L}(p') \cup \mathscr{L}(p'') = \mathscr{L}(p' \cup p'')$, where p' and p'' are regular expressions. Consequentially, a pattern such as $p = c(p_1|p_2)$ is equal to $p = c(p_1)|c(p_2)$. In this case, the filtering algorithm runs once with the original tree grammar $\mathscr{G}$ and pattern $p' = c(p_1)$ and after that with the modified grammar $\mathscr{G}'$ and pattern $p'' = c(p_2)$. As mentioned, the new target symbol is derivable from the pattern. In this case, we have $c(p_2)!c(p_1)!A$, where A is the original target.

3.3.2 Application of the Filtering Approach

As mentioned, depending on the construction of the user-specified repository Γ with typed combinators, the tree grammar resulting from the (CL)S Framework can be very large or infinite. In this section, we discuss certain classic ambiguity problems that the filtering algorithm can restrict to avoid redundant solutions.

Identity

The language of the constructed tree grammar can contain equivalent terms. In such a case, the terms derived from a grammar computed by (CL)S differ, yet define the same functions and are thereby redundant. For instance, we may consider the following example including the identity function:

$$\begin{aligned}\Gamma_{id} = \{&id : \sigma \to \sigma,\\ &x : \sigma\}.\end{aligned}$$

For the target type σ the following tree grammar gets computed by (CL)S:

$$\begin{aligned}\mathscr{G}_{id} = \{\ &\sigma \mapsto\ @(\sigma \to \sigma, \sigma),\\ &\sigma \mapsto\ x,\\ &\sigma \to \sigma \mapsto id\ \}.\end{aligned}$$

This grammar contains a cyclic rule that can result in terms such as:

$$id(id(id(x))), id(id(\ldots(id(x)))).$$

All terms of this form are equal to x. The computed grammar yields productive cycles because the combinator x can be used to derive words (terms) for the start symbol σ. Using the filtering algorithm, we reduce the number of terms in the language $\mathscr{L}(\mathscr{G}_{id})$ by restricting such trivial productions. We define a pattern $p = id(id(*))$ (in apply form: $@(id,@(id,*))$) that rejects the redundant solution presented above. In this case, we have subpatterns $p_1 = @(id,*)$, $p_2 = id$ and $p_3 = *$. Due to the given pattern, a constructor cannot have a child with the same constructor name. The filtering algorithm results in the following tree grammar, which forbids the pattern. After applying an algorithm to prune unreachable and unproductive rules, we get the following tree grammar:

$$\begin{aligned}\mathscr{G}'_{id} = \{&\{@(id,@(id,*))\}!\sigma \mapsto @(\{@(id,@(id,*))\}!\sigma \to \sigma, \{@(id,@(id,*));@(id,*)\}!\sigma),\\ &\{@(id,@(id,*))\}!\sigma \to \sigma \mapsto id,\\ &\{@(id,@(id,*));@(id,*)\}!\sigma \mapsto x\,\}.\end{aligned}$$

The new target type is $\{@(id,@(id,*))\}!\sigma$. As can be seen, the production rules in $\mathscr{G}'$ cannot construct words with the pattern p; for example, $id(id(\ldots(id(x))))$. The only solution possible is $id(x)$.

Associativity

Associativity constraints are significant if the trivial results of the inhabitation algorithm must be avoided. A tree grammar can produce both right- and left-associative trees. For example, consider the following repository Γ_a:

$$\begin{aligned}\Gamma_a = \{&f : X \to X \to X,\\ &x : X,\\ &y : X,\\ &z : X\}.\end{aligned}$$

Now assume that f satisfies the associative law $f(M,\ f(N,\ O)) = f(f(M,\ N),\ O)$ for all terms $M,\ N,\ O$. Tree grammar $\mathscr{G}_a$ (cf. Figure 3.8) computed for Γ_a and target type X produces left- and right-associated trees involving f. Figure 3.9 illustrates parts of possible parse trees that can be synthesised. As shown, the names of the constructors match the name of the children. The left associative constructor f can have a left child with the same name and for the right associativity, f has as a right child constructor f.

$$\mathscr{G}_a = \{X \mapsto\ f(X,X) \mid x() \mid y() \mid z()\}$$

Figure 3.8: Tree grammar that constructs left and right associativity

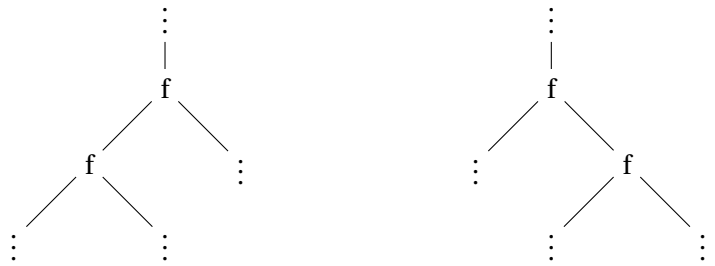

Figure 3.9: Patterns for left and right associativity

We define a pattern forbidding right associative terms by $p = f(*, f(*,*))$. The apply form of the pattern is $@(@(f,*),@(@(f,*),*))$. Here, the subpatterns are $p_1 = @(f,*)$, $p_2 = @(@(f,*),*)$, $p_3 = f$, and $p_4 = *$. The following tree grammar derives from the filtering and pruning algorithms:

$$
\begin{aligned}
\mathscr{G}' = \{&\{@(@(f,*),@(@(f,*),*))\}!X \mapsto x|y|z| \\
&\quad @(\{@(@(f,*),@(@(f,*),*))\}!X \to X, \{@(@(f,*),@(@(f,*),*));@(@(f,*),*)\}!X), \\
&\{@(@(f,*),@(@(f,*),*))\}!X \to X \mapsto \\
&\quad @(\{@(@(f,*),@(@(f,*),*));@(f,*)\}!X \to X \to X, \{@(@(f,*),@(@(f,*),*))\}!X)| \\
&\quad @(\{@(@(f,*),@(@(f,*),*))\}!X \to X \to X, \{@(@(f,*),@(@(f,*),*));@(@(f,*),*)\}!X), \\
&\{@(@(f,*),@(@(f,*),*))\}!X \to X \to X \mapsto f, \\
&\{@(@(f,*),@(@(f,*),*));@(f,*)\}!X \to X \to X \mapsto f, \\
&\{@(@(f,*),@(@(f,*),*));@(@(f,*),*)\}!X \mapsto x|y|z, \\
&\{@(@(f,*),@(@(f,*),*));@(@(f,*),*);@(f,*)\}!X \to X \to X \mapsto f\ \}
\end{aligned}
$$

According to the pattern, the original target X is modified to $\{@(@(f,*),@(@(f,*),*))\}!X$. Starting with the new target, the inhabitation algorithm cannot construct terms with right associativity. The left association restriction works analogously to the pattern $p = f(f(*,*),*)$.

Precedence

Similar to the associative constraints, with the use of precedence constraints, a specific usage order of combinators can be restricted. For example, after using the combinator f, the combinator g cannot be used until some other combinator is used. Figure 3.10 shows two parse trees that are allowed without restriction.

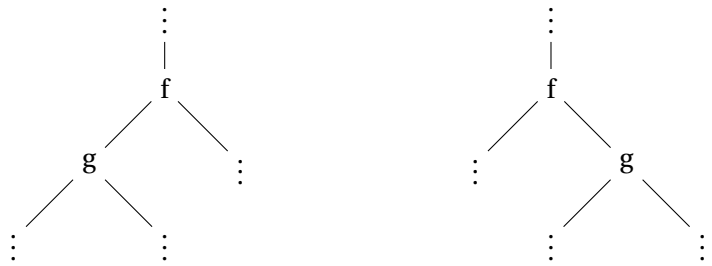

Figure 3.10: Patterns for precedence

Applying using precedence constraints, the filtering approach can resolve ambiguity problems. We consider an example presented by Adams et al. [14]. Figure 3.11 illustrates two possible parse trees for the string $1+2+3*4$.

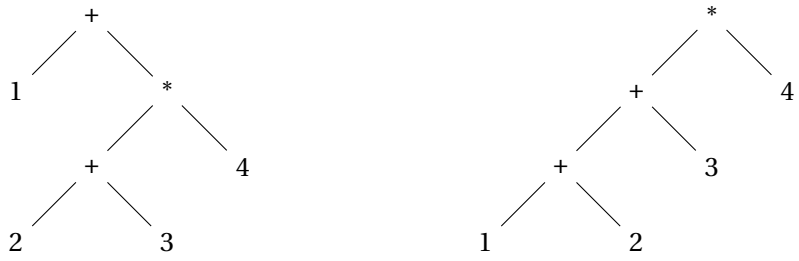

Figure 3.11: Precedence example for $1+2+3*4$

This example illustrates that the definition of a pattern must be planned well. If we define a pattern $p = mult(add(*,*),*)$ where $mult$ and add denote the mathematical symbols from the example, the filtering algorithm restricts the construction of the trees presented in Figure 3.11. According to the use case, this pattern leads to potentially solution relevant terms being filtered out. To filter out only the left tree, the pattern can be of the form $p = add(*,mult(*,*))$. Examples of the restriction of the right tree are patterns $p = add(add(*,*),*)$ and $p = mult(add(add(*,*),*),*)$.

Distributive Property

Distributivity restrictions are essential for the synthesis of program code. Better data locality can thus be achieved [95; 96; 119; 130]. Using distributivity constraints, we can handle the fusion problem presented by Meijer et al. in [97]. For example, we consider $Map(f(Map(g((x)))$ that can be represented by $Map(f \circ g(x))$.

Let us consider the loop fusion. It is a transformation of a program code aiming to merge

multiple loops into one. This merging is possible if and only if the dependencies of the statements are not reversed. Otherwise, the results can represent incorrect values [130]. For example, we consider the following for loops:

```
for (i=0; i<n; i++){
    a = i + 1;
}
for (i=0; i<n; i++){
    b = i + 2;
}
```

In this case, the statements are disconnected. Accordingly, the presented for loops can be merged to optimise the program code as follows:

```
for (i=0; i<n; i++){
    a = i + 1;
    b = i + 2;
}
```

The approach opposite to loop fusion is loop distribution, where loops can be transformed into different loops (e.g., to allow their parallel execution) if their dependencies do not impact the results.

In the above example, we can restrict solutions allowing a sequence of for loops by defining a pattern and applying the filtering approach. An example of such a pattern can be presented as follows: $p = seq(for(*), for(*))$. Here, combinator seq represents a synthesised program section sequentially concatenating two program statement blocks.

3.3.3 Limitations

The advantage of the filtering algorithm presented in Section 3.3.1 is that the number of rules in the grammar and the size of the power set are finite. As mentioned, these properties are central to the formal verification of the algorithm. Nevertheless, this approach faces certain limitations. One is the specification of the user-defined pattern p, which must be filtered out. It is a regular expression. The filtering algorithm therefore cannot be used with a pattern that cannot be derived from a tree grammar so that a pattern p is $p \in \mathscr{L}$ where $\mathscr{L}$ is a regular language [50]. For example, using the filtering algorithm presented in this work, we cannot represent the equality or inequality of subtrees with the same ancestors. Comon et al. [50] discuss automata with constraints between brothers and reduction automata. Using these approaches, equality or disequality between subtrees can be ensured. The first approach uses constraints that check positions, which should have the same ancestors. This kind of automata can thereby recognise terms of the form $f(M,M)$ with term $M \in \mathbb{A}$. The reduction automata also recognises equalities (i.e., the term $f(M,M)$), but the number of constraints on each run of the automata is restricted. The number of disequality constraints can be arbitrary,

but equality constraints must be fixed. The example of limitation also includes cases such as filtering the given pattern on certain positions in the tree.

The filtering approach restricts all terms, including the pattern p. This restriction is also a kind of limitation because there is no possibility to forbid a given pattern at the beginning, at the end, or somewhere in the tree. For example, when the application case depends on a certain number of resources that must be used, the filtering approach cannot be used to ensure the correct number. For example, Figure 3.12 displays terms that can be filtered using the following pattern: $p = f(x(*), *)$.

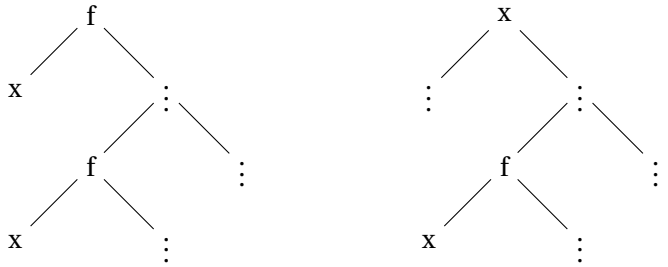

Figure 3.12: Example for a limitation

If the application case requires only one usage of resources f and x in this order, the right term would be correct in contrast to the left one. A pattern such as $p \rightarrow p$ cannot be defined to restrict only terms with patterns that occur more than once. In our case, with the application of pattern p, both terms are filtered out (cf. Figure 3.12).

3.4 Parser

This section discusses the parser technique applied in this work. Using parser combinators, higher-order functions can be modelled to convert several passed parsers into a new parser. A combinator represents a higher-order function. We used the Scala library *Scala Parser Combinators* [6] to translate the user's text entries into a valid inhabitation request for the (CL)S Framework. To achieve this translation, we developed classes including parsers for the user-specified inhabitation request and filtering pattern.

3.4.1 Translation of Inhabitation Requests

To translate the input into inhabitation requests, the implemented parser takes strings and returns `Type` objects. Extending from the trait `RegexParsers`, we can use such a function based on regular expressions for the parsing. For instance, Listing 3.5 presents functions that translate the text entries into the inhabitation rules presented in Section 2.5. The value `word`

with type `Regex` specifies which characters can be recognised as a valid word. The function `ctor` recognises words as a constructor (cf. Definition 9). The combinator `opt` stands for optional.

```
val word: Regex =
    """[a-zA-Z0-9=>\. \[\]]*[a-zA-Z0-9=>\.\[\]]""".r

def ctor: Parser[Type] = word ~ opt("("~ tyPro ~ ")") ^^ {
    case name ~ None => Constructor(name)
    case name ~ Some(_ ~ tys ~ _) => Constructor(name, tys)
}

def tyArrow: Parser[Type] = tyPro ~ opt("->" ~ tyArrow) ^^ {
    case lhs ~ Some(_ ~ rhs) => Arrow(lhs, rhs)
    case lhs ~ None => lhs
}

def tyPro: Parser[Type] = tyProduct ~ opt("*" ~ tyPro) ^^ {
    case lhs ~ Some(_ ~ rhs) => Product(lhs, rhs)
    case lhs ~ None => lhs
}

def tyProduct: Parser[Type] = tyInter ~ opt("*" ~ tyInter) ^^ {
    case lhs ~ Some(_ ~ rhs) => Product(lhs, rhs)
    case lhs ~ None => lhs
}

def tyInter: Parser[Type] = tyS ~ opt("&" ~ tyInter) ^^ {
    case lhs ~ Some(_ ~ rhs) => Intersection(lhs, rhs)
    case lhs ~ None => lhs
}

def tyS: Parser[Type] = ctor | "(" ~ tyArrow ~ ")" ^^ {
    case _ ~ ty ~ _ =>  ty
}
```

Listing 3.5: Representation of taxonomy information

Functions `tyProduct` and `tyInter` convert, for example, inputs such as $('a <*> 'b : \& : 'c)$ and $('a : \& : 'b <*> 'c)$ into a valid inhabitation requests, where the precedence over products is parsed to the left or to the right, respectively.

The `ctor` function represents a parser for constructors. The parser for a type is defined by `tyS`. It combines `ctor` and other types in parentheses using the |-combinator (`or`-combinator). Hence, it is allowed to use a constructor or other type given in parentheses. If it matches a constructor, the function returns an object of the case class `Constructor(name: String,`

`argument: Type =Omega)`. The next function `tyPro` is a parser for products of arguments (cf. Section 2.5). We consider the example presented in Section 4.3.1. Here, we have a constructor with two arguments represented as a product such as $'Pos('3 <*> '4)$. The parsing functions also consider right and left association. For example, products associate to the left so that we have the following:

$$'a <*> 'b <*> 'c = (('a <*> 'b) <*> 'c).$$

Right association is represented by intersection and arrow:

$$('a :&: 'b :&: 'c) = (('a :&: 'b) :&: 'c)$$
$$('a =>: 'b =>: 'c) = (('a =>: 'b) =>: 'c)$$

The function that recognises intersection is `tyInter`. This function translates the entries into an object of the case class `Intersection(sigma: Type, tau: Type)`. Similar to this function, the parser matching arrows is represented by `tyArrow`. It returns an object of the case class `Arrow(source: Type, target: Type)`.

3.4.2 Translation of Filtering Patterns

In this approach, we translate patterns to be restricted from the computed inhabitation result into a valid input for the filtering algorithm presented in Section 3.3. In addition to the help functions presented in Listing 3.5, the functions `pattern`, `combinator`, and `star` are used for the translation into a proper pattern (see Listing 3.6).

```
def pattern: Parser[Pattern] = combinator | star

def combinator: Parser[Pattern] = word ~ opt("("~ pattern ~ ")")
  ^^ {
    case name ~ None => Term(name, Seq.empty)
    case name ~ Some(_ ~ tys ~ _) => Term(name, tys)
}

def star: Parser[Pattern] = "*" ^^ {
    case _ => Star()
}
```

Listing 3.6: Representation of taxonomy information

According to Definition 14, a pattern can be a combinator name or application of a term to another term. Moreover, using the characters $*$, we restrict any term or application. The function `star` maps these characters to the case class `Star()` and `combinator` translates the combinators with or without arguments.

Chapter 4

Integrated Development Environment for (CL)S Framework

The Combinatory Logic Synthesizer (CL)S Framework is intended to be used for practical applications. As a result of this aim, the concept of an IDE has emerged to support nonexperts in the field of combinatory logic with intersection types. Within the scope of this dissertation, an IDE for the (CL)S Framework was developed, implemented, and evaluated. Chapter 2 outlined the theoretical foundation of the (CL)S IDE. Based on those background theories, this chapter provides detailed information about the development of the web-based implementation. Moreover, the functionalities of the IDE are presented and discussed using real application cases. Section 4.1 provides an overview of the architecture behind the developed concept for user-friendly support for the (CL)S Framework. Subsequently, a detailed description of the technical background is provided in Section 4.2. Section 4.3 presents each perspective supported by the implementation in detail. The chapter ends with a critical discussion about the developed IDE for the (CL)S Framework.

4.1 Architecture Overview

According to the theoretical background presented in Chapter 2, a web-based IDE for the (CL)S Scala Framework was developed. During the realisation, features such as usability, maintainability, and interoperability were considered. The architecture of the implementation consists of several components that ensure a straightforward representation of inhabitation information. A critical discussion of the selected components is provided in Section 4.4.

The (CL)S IDE architecture is modular and based on several projects. The IDE is publicly available online [16]. The modularity ensures easy maintainability, flexibility, and expandability of the framework according to the specification of use cases. Figure 4.1 presents an overview of the data flow between the components when using and debugging specifications.

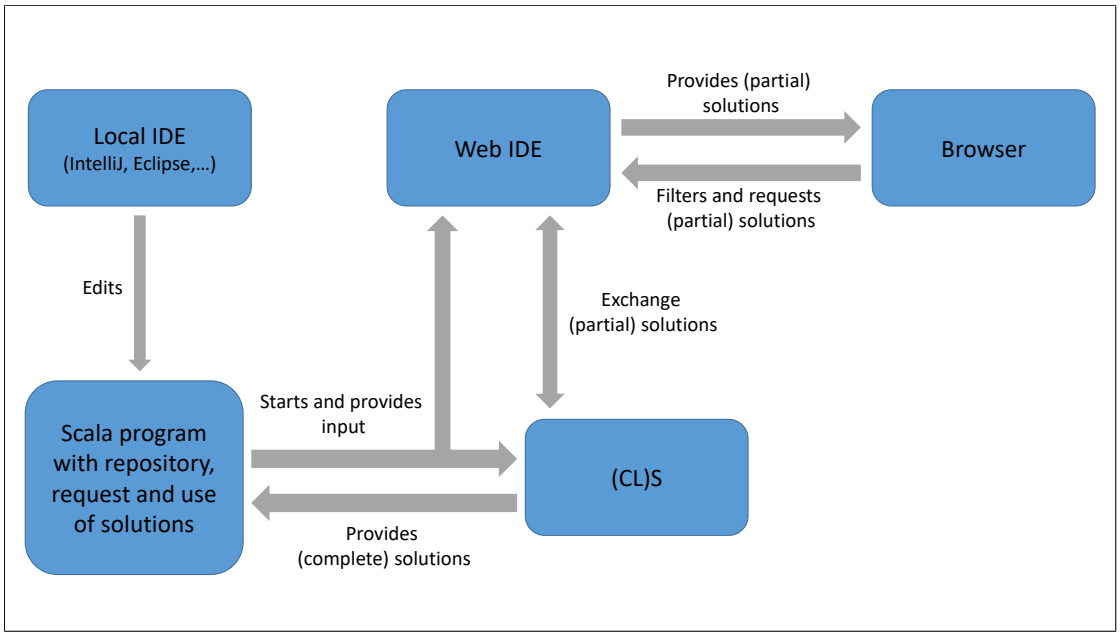

Figure 4.1: Data flow when using the (CL)S-IDE

In the figure, the blue boxes indicate the components, and the grey arrows represent the data flow that occurs when the IDE for the (CL)S Scala Framework is used. The architecture comprises the following components:

- **Local IDE**: For the specification of the inhabitation inputs, a local IDE is required. In this work, for the development and evaluation of the applications, the most commonly used Scala IDE *IntelliJ* [82] was employed. This component does not have an impact on the results, which is why any local IDE with the build tool sbt [7] for Scala can be used. The project (CL)S IDE is available as the library `cls-scala-ide` and can be used by listing it in the `build.sbt` file as follows:

```
libraryDependencies += "org.combinators" %% "cls-
  scala-ide" % "<VERSION>"
```

The string `<VERSION>` must be replaced by a released version [16] or a development version. Using the development version, the (CL)S IDE can be extended according to the current application and thus the expenditure is manageable.

- **Specifications**: Using the local IDE, the domain-specific repository with the typed combinators and the synthesis targets (cf. Section 2.5) must be defined. Moreover, finite substitution space and a subtype environment can be specified and maintained using the local IDE. The inhabitation results can also be used and investigated in the local IDE regardless of the web-based IDE [29]. Furthermore, the local IDE is essential for defining the setups of the web-based application. A detailed discussion of the specification is presented in Section 4.2.2.

- **(CL)S Scala Framework**: The (CL)S Framework receives the provided input, which is implemented using the local IDE, and then computes all possible solutions using the inhabitation algorithm presented in Section 2.1. According to the manual above, the library of the framework (`cls-scala`) must also be listed in the `build.sbt` file. This library allows for interaction with the inhabitation results using the local IDE and the maintenance and editing of the abovementioned input specifications.

- **Web IDE**: The web-based IDE supports debugging capabilities and other features. As illustrated in Figure 4.1, the (CL)S Framework communicates with the IDE by providing the inhabitation results and input specification required for understanding the search process. The framework translates the tree grammar into a hypergraph. Moreover, the filtering algorithm presented in Section 3.3 is included in the Web IDE implementation. The IDE uses the web application framework Play [91], which requires an implementation of a controller for the Play framework that manages requests and responses. The controller must be instantiated, and the specification must be passed onto it. The application instantiates a web server which hosts the website. As previously mentioned, these setups must be specified in the local IDE. More details about the implementation of the controller that hosts the (CL)S IDE are presented in Sections 4.2.2 and 4.2.3.

- **Browser**: The web-based implementation of the IDE for the (CL)S Framework provides browser independence. This means that any modern browser such as Firefox, Chrome, or Internet Explorer 10+ can be used for investigating the inhabitation results. Furthermore, warnings, information about unused specifications, step-by-step generation of the solutions, and much more can be discovered in the corresponding perspective. More details about the different perspectives are presented in Section 4.3. The target type can also be changed, or a pattern for the filtering function can be defined using the web realisation.

4.2 Technical Implementation

This section provides details about the technical implementation of the web-based IDE for the (CL)S Framework. Figure 4.2 presents the architectural pattern of Model-View-Controller applied to the architecture of the IDE. It illustrates the request/response-oriented structure of the framework. The web client is represented above the dotted line. Using any browser, users can access debugging information, define new inhabitation requests, and filter of terms among other actions. The data must be translated into a suitable format to be used by subsequent components, which is why the inputs of the users are passed to a parser (cf. Section 3.4).

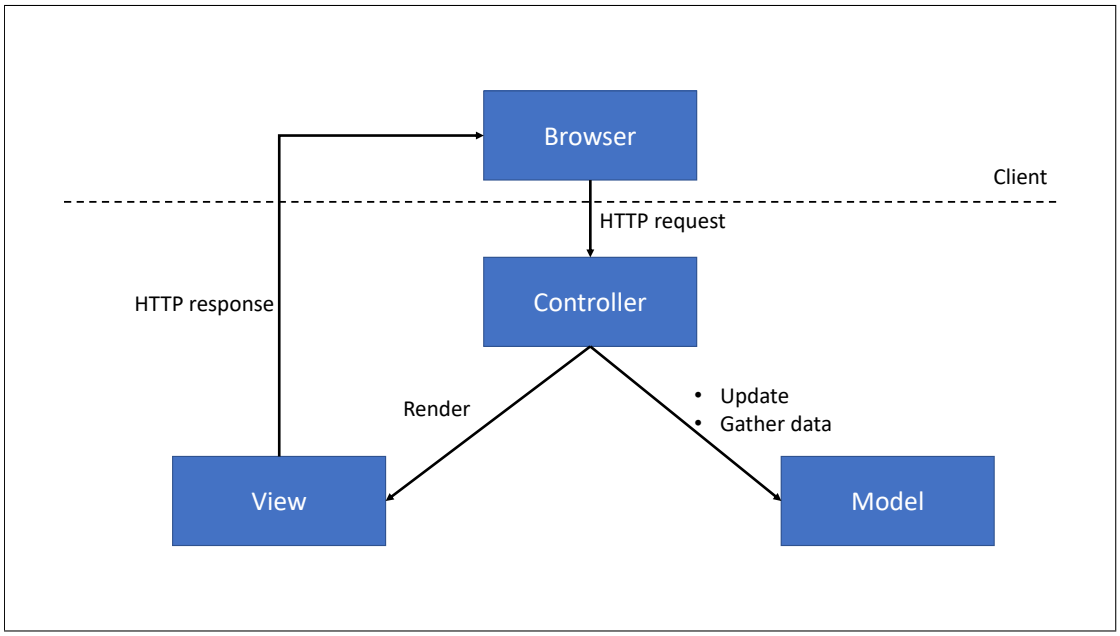

Figure 4.2: Overview of the web realisation

The `Controller` responds to the actions of the user. It receives and handles HTTP requests. The first part of the request represents the HTTP method, which in this case is the GET-method. The second part of the request is jQuery URI information. Thus, action operations from the `DebuggerController` class can be found and executed. The selected action function computes results that are then sent back in the form of an HTTP response. The implementation of these layers is performed using the local IDE. Details of the implementation of the `DebuggerController` are presented in Section 4.2.3.

In general, the model layer encapsulates the domain-specific information, which in this case is the computation of solutions by the inhabitation algorithm behind the (CL)S Framework. Based on these results, the web-based IDE provides debugging information. The controller receives the results from the model and renders a template file.

The view layer reproduces the information provided by the model. Here, the rendered form is

easily accessible and intelligible, and it aims to support nonexperts in the field of type theory. Figure 4.3 depicts the dependencies within the implementation of the IDE. As mentioned earlier, the IDE supports the user-friendly visualisation of the computed results by the (CL)S Framework. Therefore, the project `cls-scala-ide` cannot be used independently of the project `cls-scala`, which implements the inhabitation algorithm. The `cls-scala-smt` project provides the filtering approach based on SMT, which was presented in Section 3.1. The dotted arrow represents the optional connection between the projects that enables this project to be used separately without user support. This means that the usage is necessary only when SMT filtering must be applied within the IDE. A dependency on the *Play* framework is required for the web representation of the inhabitation results in the IDE.

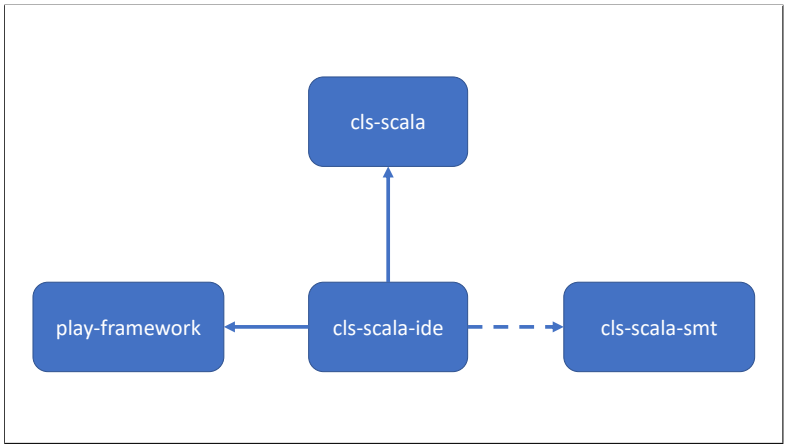

Figure 4.3: Dependencies of the projects

4.2.1 Tree Grammar Visualisation

The main aim of developing an IDE for the (CL)S Framework is to represent the relationship between grammar, combinators, and terms in a form which is easy for users to understand. As mentioned in Section 2.6.2, the tree grammars that are computed by the inhabitation algorithm are visualised using hypergraphs. To develop a user-friendly web representation, we used the JavaScript library *Cytoscape.js* [67]. Notably, Cytoscape does not provide a complete web application. An advantage of this library is that it allows interactive visualisations of networks. Moreover, Cytoscape has no platform-specific dependencies, which enables a fast and interactive representation.

To apply the library, it must be added to the `build.sbt` file of the IDE project as follows:

```
libraryDependencies += "org.webjars.bower" % "cytoscape" % "<
  VERSION>"
```

The exchange of the hypergraph elements occurs in JSON format. Listing 4.1 presents the supported format:

```
{
    "nodes": [
            {
            "data": {"label":" ", "style":" ", "id":" "}
            }
            ],
            "edges":[
            {
            "data":{"source":" ","target":" ","label":" ","id":" "
              }
            },
            "data":{"source":" ","target":" ","label":null,"id":"
              "}
            }
            ]
}
```

Listing 4.1: Representation of nodes in JSON format

In a regular hypergraph, the style of a node can be `type node` or `combinator-node`. The non-terminals and the combinators from the resulting tree grammar are represented by `type-node` and `combinator-node`, respectively. In the representation of applicative tree grammars, there are also nodes of the style `application-node`. In a debugging mode, `target nodes`, `unusable combinator nodes`, and `uninhabited type nodes` also exist, which allow the representation of the current targets in each step as well as each possible problem. Usually, unusable combinator nodes and uninhabited type nodes, which are collected during the inhabitation, cause unexpected results. An individual style represents these nodes to warn the user that something could be wrong. Furthermore, each node has a label and an ID. The labels are based on the domain specifications, while the IDs are required for the unique allocation of the elements. In addition to an ID, the edges have a specification of `source` and `target`. The position of the argument labels the edges if the source is a combinator node and the target is a type node. If the edge connects a type node with a combinator node, the label is `null`. The function `toGraph` implemented in the `DebuggerController` class translates the inhabitation results into JSON format and assigns IDs to the graph's components.

In addition to Cytoscape, the IDE is implemented using the *Bootstrap* framework [3]. In this work, we used the HTML components. The library is specified as follows: `"org.webjars"% "bootstrap"% "<VERSION>"`, where in this work the version 3.3.7-1 is used. The framework allows faster and easier web development, which also ensures support for browser and platform independence [1].

4.2.2 Web-Based Realisation

As previously mentioned, the standard Scala web application framework Play [77] is used for the web-based realisation. Using the `build.sbt` file, the Play libraries that are necessary for building the project are added to the project as follows: `lazy val root =(project in file(".")).enablePlugins(PlayScala)`. The `sbt` plugin in the file `plugins.sbt` is added as follows: `addSbtPlugin("com.typesafe.play"% "sbt-plugin"% "<VERSION>")`. At the time of writing, version 2.6.9. is used. An advantage of the *Play* framework is that change detection occurs automatically, which helps optimise the time of the development process. The framework checks the files in the source code, and if there are any changes, they can be updated by refreshing the browser. Play recompiles and restarts the Akka Http server.

Figure 4.4 illustrates the architecture of the IDE as well as the life cycle of a request in detail.

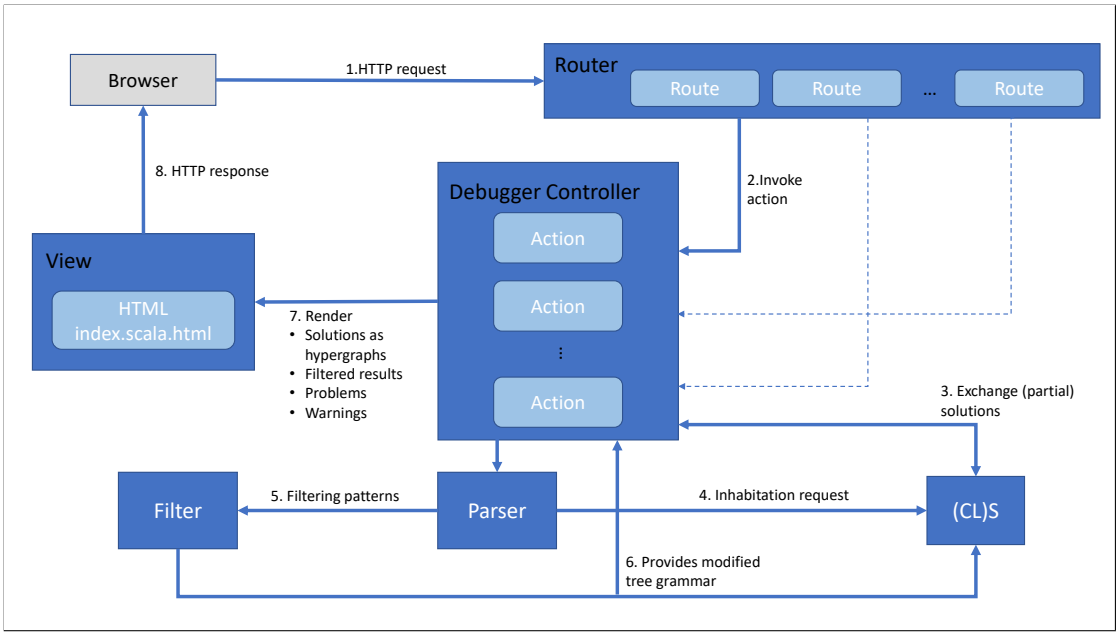

Figure 4.4: Overview of request life cycle

The path that occurs is described as follows:

1. An *HTTP Server* receives the user request. In this work, the integrated web server is Akka HTTP [2]. Akka works with actors that allow the realisation of concurrent, parallel, and distributed systems [2]. A Play web server configuration includes a definition of the routes that are available in the `resources/routes` file.

2. The router is essential for the translation of a `HTTP` request to a `play.api.mvc.Action`. The defined routes in the *router* file lead to the configuration class (`DebuggerController`), as depicted in Figure 4.5, which handles the request. For example, the action method

`showGrap()` obtains the inhabitation results and translates the tree grammar. It returns a graph in JSON format, as presented in Section 4.2.1. According to the HTTP request, the action `showStep(step: Int)` returns a graph that is also in JSON format, but only for the first step of the search process and the function `showResult(inhabitant: Int)` returns the first inhabitant. If there are any router errors, they become visible in the browser. Detailed information about this configuration of the controller is presented in Section 4.2.3.

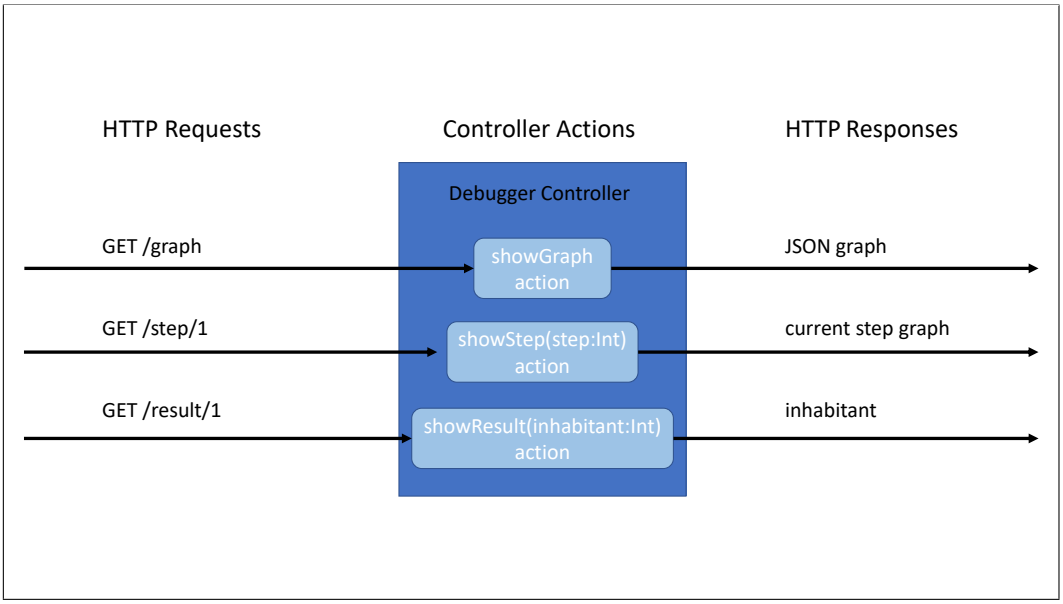

Figure 4.5: Workflow of requests

3. Using the web-based user interface, users access inhabitation information provided by the model layer represented by the (CL)S Framework. According to the request, the model layer returns the computed results. The response can be information such as the complete tree grammar, a specific inhabitant or step of the searching process, input specifications, or filtering results.

4. A new inhabitation request can be made using the web-based IDE. Using a text field, the users can define new inhabitation requests. The parser must translate the text entries into an acceptable format for the (CL)S Framework.

5. As mentioned in Section 3, the filtering algorithm requires user specification of the pattern. Users can define a domain-specific pattern if some inhabitation results should be restricted. A parser translates the input into the pattern format.

6. The filtering algorithm provides the new modified tree grammar, which cannot construct the user-specific pattern. The IDE visualises the new tree grammar, which can be used for the computation of a new inhabitation.

7. The view layer renders the results of the called action function. Depending on the response, the web format is either JSON (cf. Section 4.2.1) or HTML.

8. The computed information is sent back to the client in the form of an HTTP response.

The implementation of the web-based IDE and the examples used for its verification are discussed in Chapter 5.

4.2.3 Definition of the Debugger Controller

To start the debugger for the (CL)S Framework, the *Play* application must be set up. The router uses a `resources/routes` file, which is compiled. The `resources/routes` file includes a representation of all routes that are required by the current application. Moreover, it provides information about the location of a different router that has to be used [77]. In the labyrinth example, to use the domain-specific router, we add the following prefix to the file:

```
->      /       org.combinators.cls.ide.labyrinth.LabyrinthController
```

The presented route leads to the configuration class presented in Listing 4.2. Such configuration classes must extend the class `org.combinators.cls.ide.DebuggerController` and the trait `org.combinators.cls.ide.DebuggerEnabled`, which are part of the `cls-scala-ide` library. Thus, an action operation can be found. The `DebuggerController` class includes functions that manage the generation of `Action` values. The trait `org.combinators.cls.ide.DebuggerEnabled` defines the configuration of the *Play* web server routes. Listing 4.2 presents the implementation of the controller that hosts the IDE for the (CL)S Framework:

```
class LabyrinthController @Inject()(val webJarsUtil: WebJarsUtil,
    val lifeCycle:ApplicationLifecycle, assets: Assets)
        extends DebuggerController(webJarsUtil, assets)
            with DebuggerEnabled {

        lazy val target: Type = 'Pos('0, '0)
        lazy val repository = new Repository
        override val controllerAddress: String = "labyrinth"
        override val projectName = controllerAddress
        override val tgts: Seq[Type] = Seq(target)
        override val refRepo: Option[ReflectedRepository[_]] =
          Some(Gamma)
        override val result: Option[InhabitationResult[Unit]] =
          Some(Gamma.inhabit[Unit](target))
}
```

Listing 4.2: Definition of debugger controller

The actor model of the Akka HTTP web server can be combined with the lightweight Google Guice [11] framework to support the realisation of the *dependency injection* design pattern. Using the annotation `@Inject` on a constructor, we can declare components as dependencies. Furthermore, the configuration class includes fields that must be overridden. The variable `controllerAddress` is a user-defined specification of the address where the IDE is accessible. This means that the complete address is a combination of the default settings as well as the user-defined specifications. In this example, the address where the results can be viewed is as follows: `http://localhost:9000/labyrinth/ide/`, where `labyrinth` is the `controllerAddress` defined by the user. The user can define the name of the project, which will be displayed on the website. By default, the project name is the same as the `controllerAddress`. In addition to configuring the address, the inhabitation information must be overridden. The field `tgts` represents the domain-specific initial inhabitation targets that must be shown in the web-based IDE. For the visual representation, the reflected repository and the results are also required. In the aforementioned example, `Gamma` represents the reflected repository and `result` represents all possible inhabitation results.

A request path can be defined using static or dynamic paths. An example of a static path is `http://localhost:9000/labyrinth/ide/graph`. The function `showGraph` in Listing 4.3 represents the corresponding action operation (see Figure 4.5). Such operations are defined in the `DebuggerController` class.

```
  def showGraph = Action {
    treeGrammar.nonEmpty match {
        case true =>
            graphObj = Json.toJson[Graph](toGraph(treeGrammar))
            Ok(graphObj.toString)
        case false =>
            Ok("Inhabitant not found!")
    }
}
```

Listing 4.3: Example of `Action` operation

As can be seen, the `showGraph` operation does not require parameters and returns an `Action` value. If the inhabitation algorithm computes a solution, the method `showGraph` returns a graph object as a JSON value. If the tree grammar is empty, then there is no solution, so the function returns the message *'Inhabitant not found!'* to inform users. To develop a user-friendly IDE, we also used action generator methods with parameters for a more precise specification of requests. Such action operations are called from dynamic request paths. Examples of such requests are as follows: `GET / result/1` and `GET / step/1`, as shown in Figure 4.5. An action method that corresponds to the first request is presented as follows:

```
def showResult(inhabitant: Int) = Action {
    try {
        Ok(result.get.terms.index(inhabitant).toString)
    }
    catch {
        case _: IndexOutOfBoundsException => play.api.mvc.Results.
          NotFound(s"404,
             Inhabitant not found: $inhabitant")
    }
}
```

Listing 4.4: Definition of the `Action` function that returns a certain inhabitant

Such functions are defined in the `DebuggerController`. Here, the URI pattern that calls the presented action is `GET / result/1` (see Figure 4.5). The method extracts the data that are relevant from the request path. In this case, the user selects an inhabitant with number `1`. Inhabitants, which must be visualised separately, can be chosen from the list of solutions (see Figure 4.18). More details about the front end are presented in the next section. The number, which can be extracted from the URI request, is also the value of the parameter for the action function `showResult(inhabitant:Int)` that is set. According to the index, this function returns a specific inhabitant in the form of a hypergraph in JSON format and in the form of tree as a string. If there is no inhabitant, then the function returns a `NotFound` from the trait `play.api.mvc.Results`, which supports the generation of results.

In addition, other (helpers) action operations are implemented in the `DebuggerController` class that support the user-friendly IDE for the (CL)S Framework. They collect and parse the data delivered from the application specifications or inhabitation algorithm. All action methods return `play.api.mvc.Results` values. For example, the function presented in Listing 4.3 results in an HTTP response that contains the status code `200 OK` and a response body.

4.3 IDE Perspectives

This section provides an overview of the developed perspectives of the web-based IDE for the (CL)S Framework. Eight perspectives aim to support the usage of combinatory logic synthesis with intersection types. The presentation of the IDE is based on labyrinth examples; these examples are extensions of each other (s. Section 4.3.1). They are also used as use cases in [30; 85; 39]. These examples were inspired from [46]. Here, the authors present a synthesis of robot motion plans from a given workspace and a given set of obstacles that can move. This labyrinth example cannot be measured with existing path-finding algorithms, but such use cases are of interest for the examination of scaling properties. Moreover, they help improve the performance of the (CL)S Framework by removing the generation of redundant recursive inhabitants targets. Furthermore, the sizes of the use cases are manageable, such that it is possible to present the features of the different IDE perspectives using the examples. In the following, Section 4.3.1 provides an overview of the application cases and Sections 4.3.2–4.3.9 present each perspective.

4.3.1 Application Cases

We consider two variations of the labyrinth example. Each aims to find all possible paths from the start to the goal position using the inhabitation framework. It is possible to go *up, down, right*, or *left*, if an obstacle does not occupy the new position. The inhabitation algorithm computes all possible paths and each word derived for the start symbol represents a path from the start to the goal position. If there is no word, there is no path. Figure 4.6 shows the first example with a size of 4×4. The start position is on $(0 * 2)$ (denoted by •) and the goal position is on $Pos(2 * 0)$ (denoted by ★).

	0	1	2	3
0			★	■
1	■			
2	•		■	
3	■			

Figure 4.6: Example of labyrinth 4×4

The repository is shown in Figure 4.7. As can be seen, the combinators in the repository consider all possible moves. Figure 4.8 presents the tree grammar produced by the inhabitation algorithm, where the target type is the goal position – $Pos(2 * 0)$.

$$
\begin{aligned}
\Gamma = \{ & start : Pos(0*2) \\
& up : (Pos(1*3) \to Pos(1*2)) \cap (Pos(1*2) \to Pos(1*1)) \cap \\
& \quad (Pos(1*1) \to Pos(1*0)) \cap (Pos(3*3) \to Pos(3*2)) \cap \\
& \quad (Pos(3*2) \to Pos(3*1)) \cap (Pos(2*1) \to Pos(2*0)) \\
& down : (Pos(1*0) \to Pos(1*1)) \cap (Pos(1*1) \to Pos(1*2)) \cap \\
& \quad (Pos(1*2) \to Pos(1*3)) \cap (Pos(2*0) \to Pos(2*1)) \cap \\
& \quad (Pos(3*1) \to Pos(3*2)) \cap (Pos(3*2) \to Pos(3*3)) \\
& left : (Pos(2*0) \to Pos(1*0)) \cap Pos(1*0) \to Pos(0*0)) \cap \\
& \quad (Pos(3*1) \to Pos(2*1)) \cap (Pos(2*1) \to Pos(1*1)) \cap \\
& \quad (Pos(3*3) \to Pos(2*3)) \cap (Pos(2*3) \to Pos(1*3)) \cap \\
& \quad (Pos(1*2) \to Pos(0*2)) \\
& right : (Pos(0*0) \to Pos(1*0)) \cap (Pos(1*0) \to Pos(2*0)) \cap \\
& \quad (Pos(1*1) \to Pos(2*1)) \cap (Pos(2*1) \to Pos(3*1)) \cap \\
& \quad (Pos(0*2) \to Pos(1*2)) \cap (Pos(1*3) \to Pos(2*3)) \cap \\
& \quad (Pos(2*3) \to Pos(3*3))\}
\end{aligned}
$$

Figure 4.7: Repository for the labyrinth example in Figure 4.6

$$
\begin{aligned}
\mathscr{G} = \{ & Pos(0*2) \mapsto \{\, start()\}, \\
& Pos(0*0) \mapsto \{\, left(Pos(1*0))\}, \\
& Pos(1*0) \mapsto \{up(Pos(1*1)), left(Pos(2*0)), right(Pos(0*0))\}, \\
& Pos(1*1) \mapsto \{down(Pos(0*1)), up(Pos(1*2)),\ left(Pos(2*1))\}, \\
& Pos(1*2) \mapsto \{down(Pos(1*1)), up(Pos(1*3)),\ right(Pos(0*2))\}, \\
& Pos(1*3) \mapsto \{down(Pos(1*2)),\ left(Pos(2*3))\}, \\
& Pos(2*0) \mapsto \{up(Pos(2*1)),\ right(Pos(1*0))\}, \\
& Pos(2*1) \mapsto \{down(Pos(2*0)), right(Pos(1*1)),\ left(Pos(3*1))\}, \\
& Pos(2*3) \mapsto \{left(Pos(3*3)),\ right(Pos(1*3))\}, \\
& Pos(3*1) \mapsto \{right(Pos(2*1)),\ up(Pos(3*2))\}, \\
& Pos(3*2) \mapsto \{up(Pos(3*3)),\ down(Pos(3*3))\}, \\
& Pos(3*3) \mapsto \{down(Pos(3*2)),\ right(Pos(2*3))\}\}
\end{aligned}
$$

Figure 4.8: Resulting tree grammar for target type $Pos(2*0)$

Figure 4.9 represents the second labyrinth example. The example is smaller, but the construction of the labyrinth generates special cases that can be detected by the (CL)S IDE. The stars represent optional goal positions, and the bullet at position $Pos(0*0)$ represents the

start position. According to these conditions, there is a no path from the start to the goal positions $Pos(2*0)$ (denoted by $\bigstar_2$) and $Pos(4*1)$ (denoted by $\bigstar_3$). These goals were chosen deliberately to demonstrate the usability of the IDE for the (CL)S Framework in special cases.

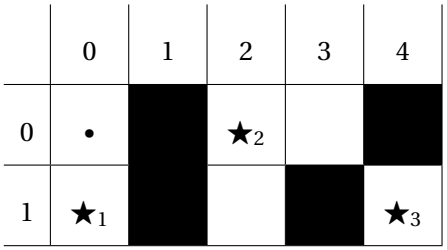

Figure 4.9: Example of labyrinth 5 × 2

The specification of the combinators is shown in Figure 4.10. According to the inhabitation request, the results can be unexpected. For example, for goal position $Pos(0*1)$, the inhabitation algorithm computes the tree grammar shown in Figure 4.11. It does not contain the combinators $right$ and $left$ because the obstacles do not allow such paths. The reason is obvious in this use case, but considering larger repositories, the clear overview can be lost.

$$\begin{aligned}\Gamma = \{ \quad & start : Pos(0*0), \\ & up : (Pos(0*1) \to Pos(0*0)) \cap (Pos(2*1) \to Pos(2*0)), \\ & down : (Pos(0*0) \to Pos(0*1)) \cap (Pos(2*0) \to Pos(2*1)), \\ & left : Pos(3*0) \to Pos(2*0), \\ & right : Pos(2*0) \to Pos(3*0) \quad \}\end{aligned}$$

Figure 4.10: Repository for the labyrinth example in Figure 4.9

$$\begin{aligned}\mathscr{G} = \{ \quad & Pos(0*0) \mapsto \{start(), up(Pos(0*1))\}, \\ & Pos(0*1) \mapsto down(Pos(0*0)) \quad \}\end{aligned}$$

Figure 4.11: Tree grammar for target $Pos(0*1)$

Due to the simplicity of the labyrinth examples, the presentation of the (CL)S IDE is possible using small graphs and a manageable number of solutions. A disadvantage is that these application cases are not sufficiently complex, such that all features cannot be considered. For this reason, we used larger repositories from real synthesis projects in some cases. More details about these projects are presented in Chapter 6. These repositories include hundreds of combinators, so they are logically more susceptible to errors. In the following, we detail

how the perspectives provided by the web-based IDE handle the labyrinth example and these additional application cases.

4.3.2 Result Overview

The first perspective that can be seen after starting the web-based (CL)S IDE is the *Result Overview* perspective. Figure 4.12 displays the results of the synthesis of labyrinth paths. Here, the small variation of the labyrinth example with the first goal position is considered (cf. Figure 4.9). Thus, the inhabitation question is: $\Gamma \vdash ? : Pos(0 * 1)$. The left side lists all available perspectives, and users can easily switch between them. On the right side, the inhabitation request is displayed. The representation of the resultant tree grammar (cf. Figure 4.11) is centrally displayed below the inhabitation request. As mentioned in Section 2.6.2, the graphical visualisation of the tree grammars is supported by hypergraphs, where the nonterminals are represented by yellow boxes and the combinators, by blue circles.

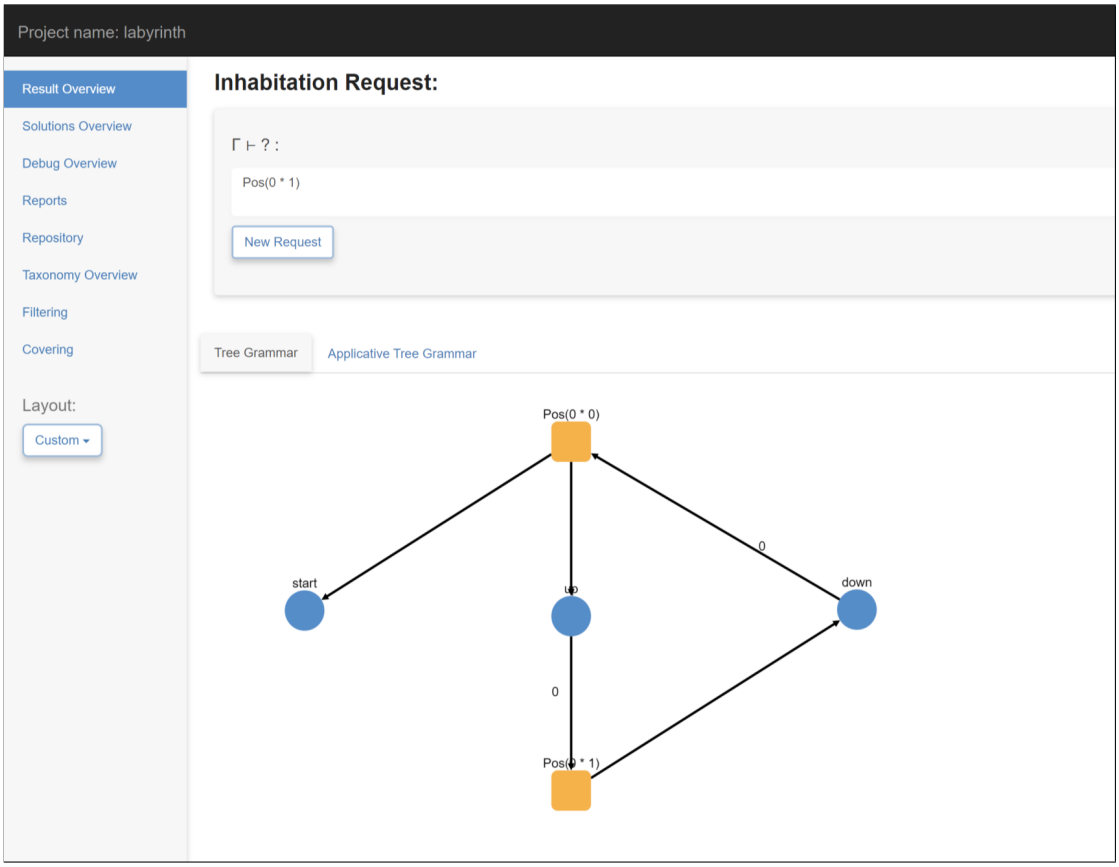

Figure 4.12: Result overview

Moreover, Figure 4.12 demonstrates that users can choose between a representation of the tree grammars or the applicative tree grammars (cf. Section 2.3). The algorithm computes

the applicative tree grammar presented in Figure 4.13 and Figure 4.14 illustrates its graphical representation in the IDE.

$$
\begin{aligned}
\mathcal{G}_@ = \{ \quad & Pos(0*0) \mapsto start, \\
& Pos(0*0) \mapsto @(Pos(0*1) \to Pos(0*0), Pos(0*1)), \\
& Pos(0*1) \mapsto @(Pos(0*0) \to Pos(0*1), Pos(0*0)), \\
Pos(0*0) \to Pos(0*1) & \mapsto down, \\
Pos(0*1) \to Pos(0*0) & \mapsto up \quad \}
\end{aligned}
$$

Figure 4.13: Applicative tree grammar for target $Pos(0*1)$

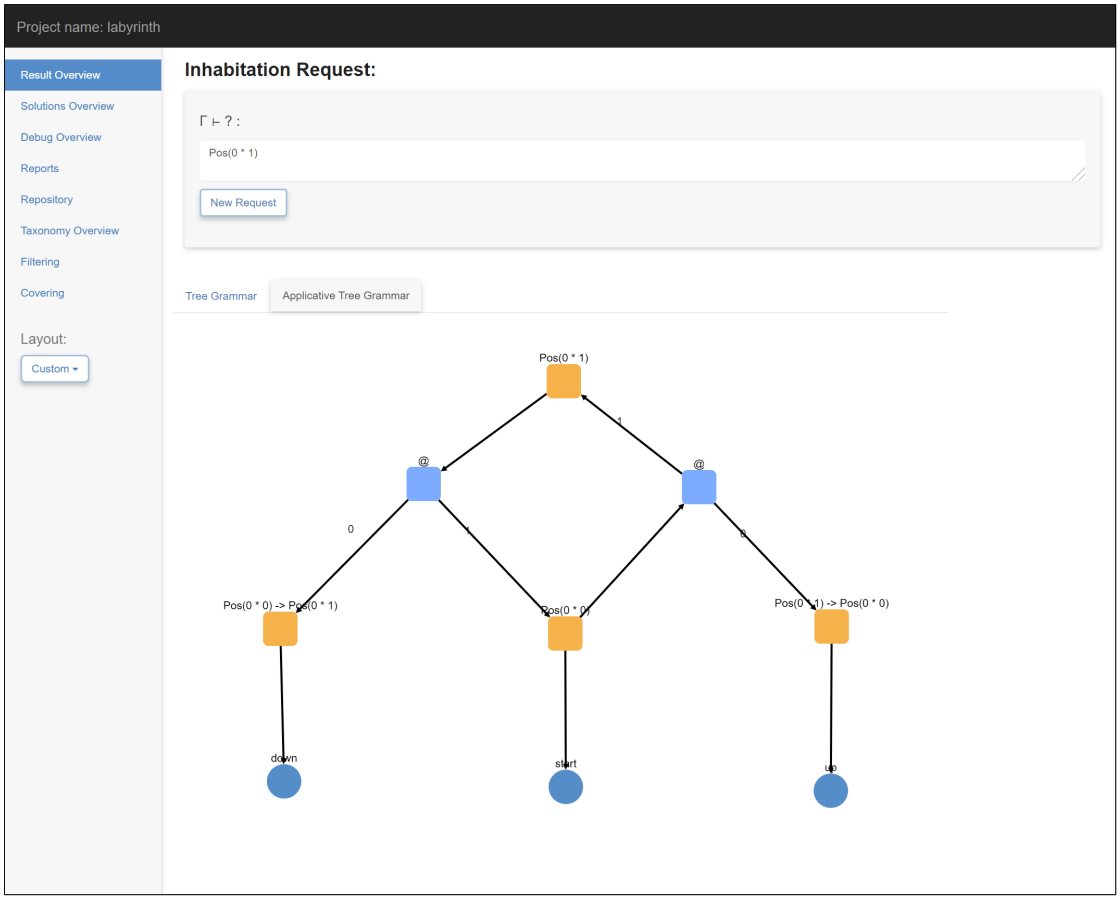

Figure 4.14: Result overview with applicative tree grammar

As can be seen in Figure 4.14, the apply nodes are represented by light blue boxes and denoted by @. Notably, this kind of graphs always includes more nodes than the other visualisation techniques, resulting from the binary structure of the applicative tree grammars. The size of the tree grammar in the example presented here is manageable, ensuring the elements are

still clear. Considering an example with more complex tree grammar, we can see that the larger the number of nodes, the more obscure the graph can become (see Figure 4.15). For reasons of usability, the representation of the results focuses on the graphs without apply nodes. Importantly, the IDE provides visualisation of both the tree grammars (cf. Figure 4.14).

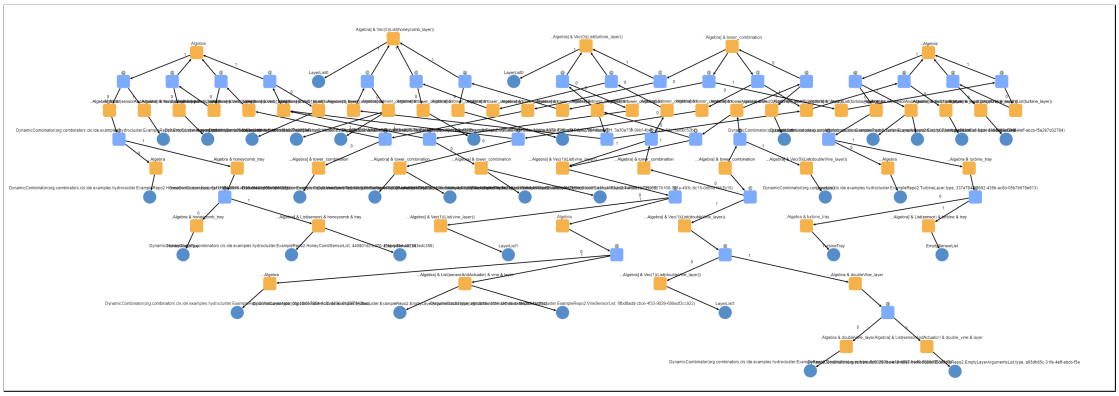

Figure 4.15: Example of applicative tree grammar visualisation

If the inhabitation was unsuccessful, the type inhabitation algorithm returns an empty solution set. We use textual information to inform the users that there is no result. In this case, the action function presented in Listing 4.3 returns the message that an inhabitant was not found. Detailed information about the reasons that lead to non-inhabitability can be found in the *Reports* perspective (see Section 4.3.5), where possible scenarios are also discussed.

According to the domain specification, the represented graph structure can be enormous. Figure 4.16 shows a hypergraph resulting from an application developed into the scope of a student project [13], as detailed in Section 6.3. The repository was used for the synthesis of different configurations of intelligent plant management systems. The hypergraph represents an average solutions' size. According to the target type and the repository used, the sizes of the tree grammars can be completely unique. The number of combinators in the used repositories is between 60 and 294. The visualisation of data using graphs is very challenging

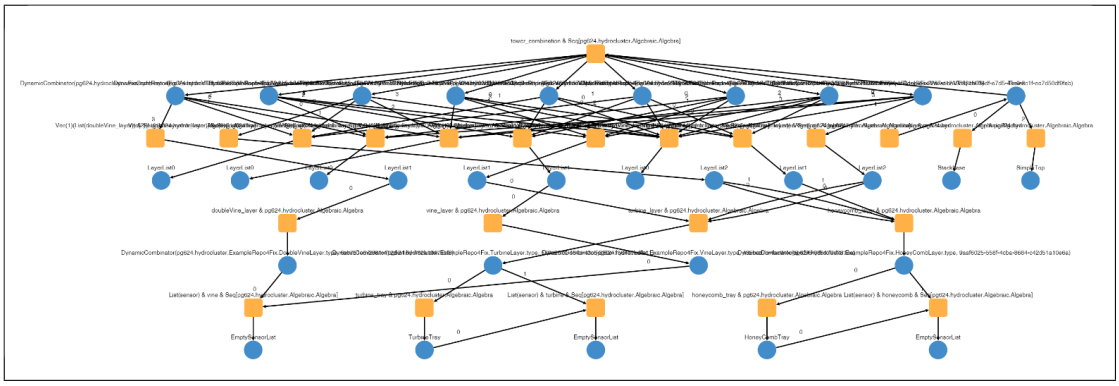

Figure 4.16: Example of a complex hypergraph

even beyond the field of computer science [92]. To increase the comprehensibility of the visualised information, we developed the functions for zooming in and zooming out. Users can also move the edges and the nodes of a visualised hypergraph. This feature is crucial in cases where the number of nodes is very large, as shown in Figure 4.16. Evidently, big hypergraphs can lead to labels of nodes or edges that visually overlap because there is insufficient space for the visualised tree grammar. Depending on the complexity of the graph, different layouts are sometimes more user-friendly and useful for the investigation of the represented information.

In certain cases, the overlapping problem can also be avoided using a different layout. The layout's algorithms allow an automatic reordering of nodes in the hypergraph. The algorithms included in the IDE are extensions of the Cytoscape.js library [67]. The dropdown list manages them on the left side (cf. Figure 4.12). The framework supports the following layouts:

- *breadthfirst*: The nodes are ordered according to a hierarchy.
- *random*: The nodes are ordered in random positions.
- *grid*: The nodes are ordered into a grid.
- *circle*: The nodes are ordered in a circle.
- *concentric*: The nodes are ordered into a concentric circle.
- *dagre*: This layout is proper for the representation of directed graphs and trees because it orders the nodes also according to a hierarchy.
- *cose*: This layout emphasises groups of nodes through analysis of the connections.
- *custom*: The nodes can be ordered by a user.

The default layout in the web-based IDE is *breadthfirst*. According to Franz et al. [67], this type of layout is well-suited to the representation of trees because of the structural similarity. This layout makes the content legible for the (CL)S IDE users because of the intuitive and straightforward structure, similar to tree grammars. Figure 4.17 exemplifies another representation of the labyrinth example results with target type $Pos(0 * 1)$ using the circle layout. Here, nodes are positioned according to the number of edge crossings and node types. In this way, nodes can be easily discovered because they are grouped around the circle.

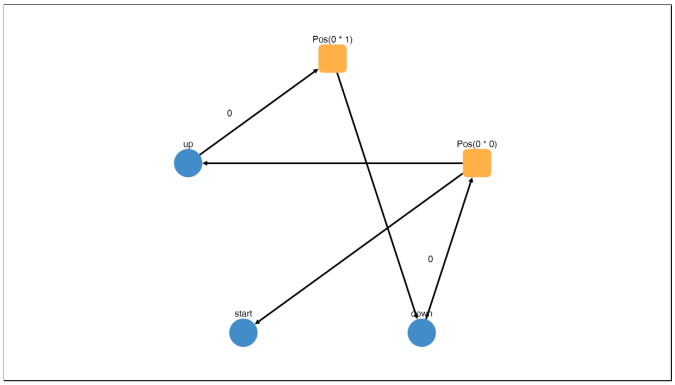

Figure 4.17: Representation using circle layout

As can be seen in Figure 4.12, a new request can also be made in the web-based IDE. Users can easily define a new request that is more specific or generic to investigate the inhabitation and to obtain better synthesis results. They then do not have to switch to the programmatic definition of the request, and they can experiment faster and more comfortably with the inhabitation. The parser technique presented in Section 3.4 analyses the input in the text field in IDE. According to the rules of the context-free grammar, the parser translates the user entry into target type. As mentioned, the inhabitation request can also be stated using the local IDE. In this case, after refreshing the browser, the representation of the new results follows.

4.3.3 Solutions Overview

To make the investigation of the results more accessible, we developed the *Solutions Overview* perspective. In this perspective, a list of all terms can be seen if the tree grammar can compute a finite number of results. If the tree grammar is empty, a text message like in the *Result Overview* informs the users about the non-inhabitation. If the tree grammar can compute an infinite number of results, a window with the message: *The result is infinite! How many solutions should be shown?* pops up on the screen. The user must enter the number of inhabitants into a text field.

Through the selection of variations from the list with the results, the associated inhabitant, as well as the corresponding hypergraph, can be visualised. Figure 4.18 displays the behaviour after the selection of `Variation 0`. Considering the labyrinth example in Figure 4.9 and target type $Pos(0 * 1)$, we get an infinite number of results because of the cycle that can be seen in the hypergraph (cf. Figure 4.12). The inhabitant is presented in a text form. Each hypergraph presented in this perspective represents a subhypergraph of the hypergraph presented in the *Result Overview* perspective. After selecting another variation, the old representation of the graph is replaced by a new one, and the selected inhabitant is displayed.

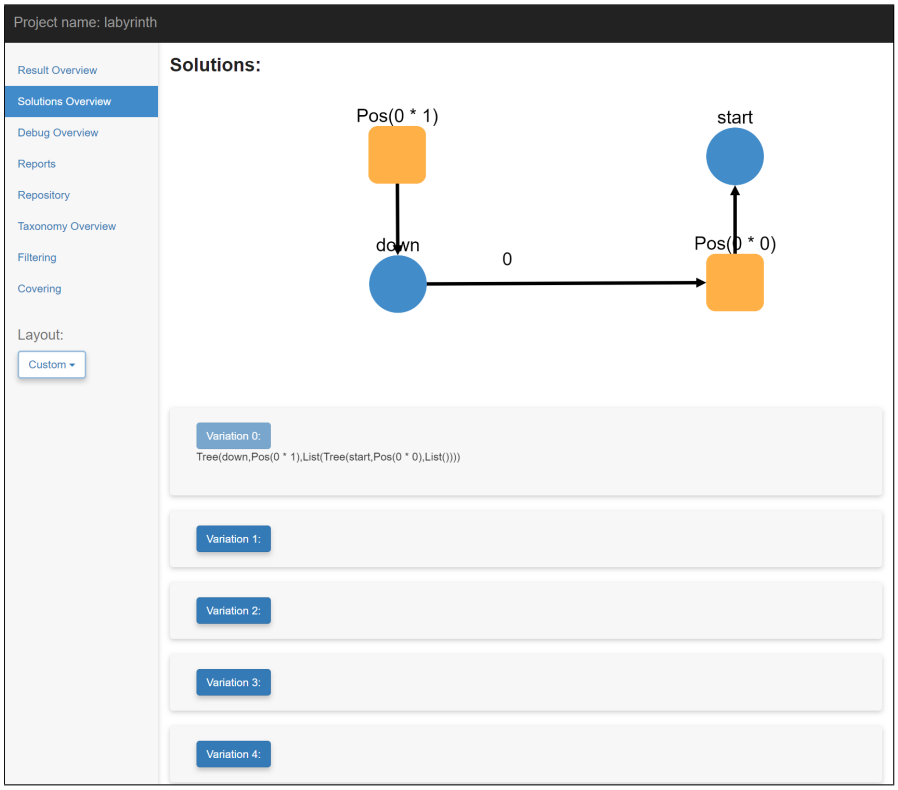

Figure 4.18: Representation of an inhabitant

Figure 4.19 presents a part of the generated inhabitants for the application case presented in Section 6.2, illustrating an extension of the basic functionalities. Beyond the VARIATION X buttons (cf. Figure 4.18), the interpreted terms can be downloaded, where X is the number of the term corresponding to the enumeration of the inhabitants. This feature requires certain combinators to construct the desired file. According to the presented use case, `.xml` and `.p` files are computed so that arbitrary results can be downloaded after the expectation (Figure 4.19, in the bottom-left corner). Every application case does not require such a download feature; by default applying this perspective, users can only discover the terms separately, but the results have shown that the implementation of this feature is manageable.

The visualisation in this perspective is crucial for the usability of the tool because it allows separate consideration of the inhabitants. The uninterpreted terms can then be compared. Moreover, the specifications of the combinators, which are responsible for the ambiguity, can be detected and restricted by using the filtering functionalities provided by the *Filtering* perspective. Section 3.3 details the filtering approach integrated into the IDE for (CL)S Framework. Furthermore, when the application case is too complex, the separate visualisation allows a more precise investigation of the results, which can be useful for the development of the repository to further support the user.

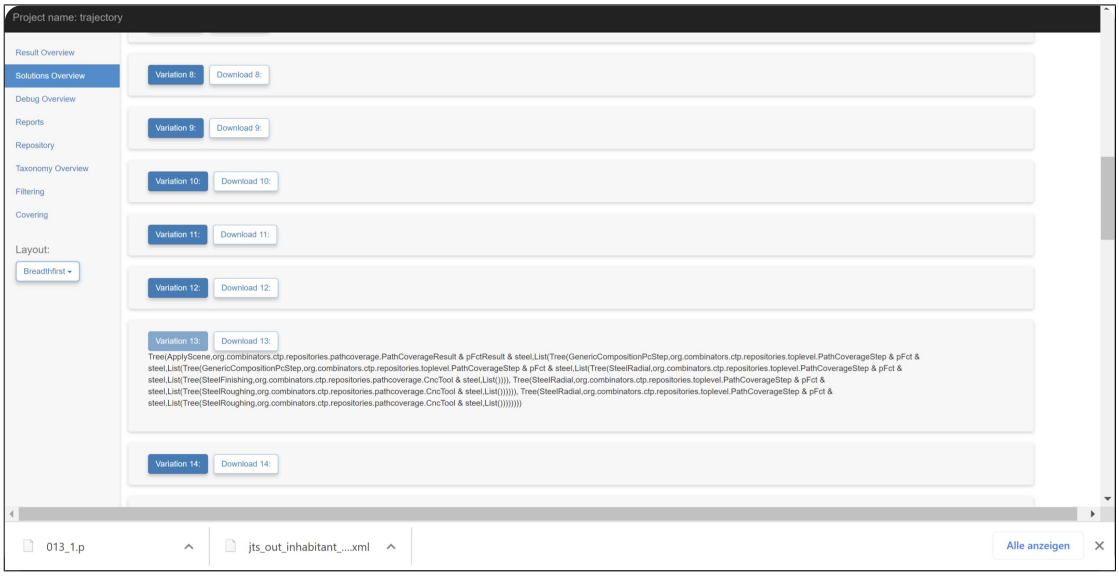

Figure 4.19: Overview of the solutions

4.3.4 Debugger Overview

This section presents the central part of the IDE: the *Debugger Overview* perspective. This perspective provides a step-wise visualisation of the generation of the results that is essential to understand the search process implemented by the inhabitation algorithm. Here, users can control the composition of the components from the given repository by selecting the steps of the inhabitation process. Using the buttons with the forward and backward arrows (see Figure 4.22), the search process can be discovered step by step. As mentioned in Section 2.6.2, in a hypergraph, the combinators are represented by blue cycles and types, by yellow squares with rounded edges. In each step, the current tree grammar is represented by the associated hypergraph. In the following, we consider the labyrinth example in Figure 4.9 with target type $Pos(0 * 1)$ (cf. Section 4.3.1). By marking the nodes in green, the algorithm presents the yet-to-do targets. The first step of the search process is represented by the target type (see Figure 4.20) and then each step illustrates the recursive targets in green. Figure 4.21 shows the first step, where the current target is $Pos(0 * 0)$. The visualised rule is $Pos(0 * 1) \rightarrowtail down(Pos(0 * 0))$, which is part of the tree grammar presented in Figure 4.11. As can be seen, the current type is the argument of the combinator $down$ that must be used for the inhabitation. In this example, the second step is also the last one. As shown in Figure 4.22, there are no more green nodes that must be handled. Moreover, the visualised hypergraph is the same as the one presented in the *Result Overview* perspective (cf. Figure 4.12).

Figure 4.20: Visual representation of the initial step

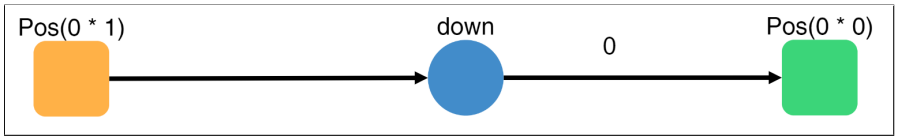

Figure 4.21: Visual representation of step 1

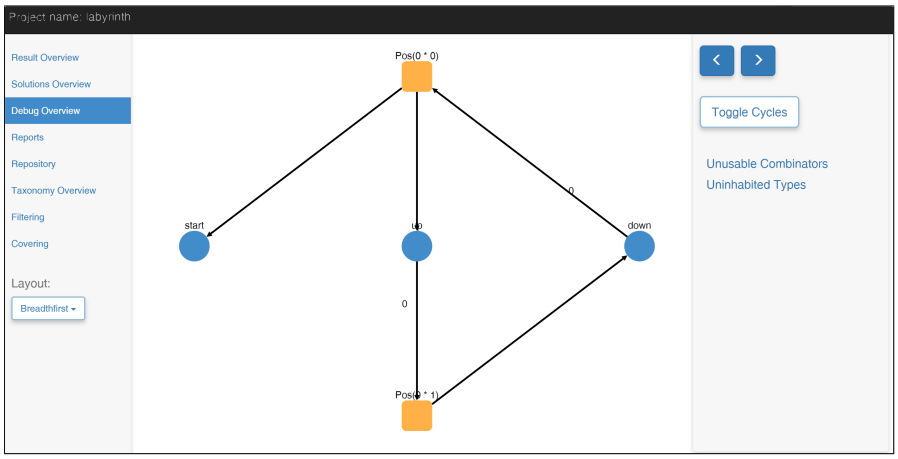

Figure 4.22: Visual representation of step 3

Aside from the manual building of the solution, this perspective provides short information about a possible non-inhabitation occurrences. In Section 3, we mentioned that there could occur productive and unproductive cycles. For example, the hypergraph presented in Figure 4.22 includes a productive cycle. In this case, the specification allows the usage of combinator up after combinator $down$ and vice versa. The reason, why the algorithm can compute results is that the combinator *start* can be used as an *exit* from the cycle and there are no arguments that must be handled so that the search process stops. However, there are also unproductive cycles that can be detected in the visualisation of the search process. In this case, the combinator nodes, which cause such states, are marked in red. Figure 4.23, Figure 4.24, and Figure 4.26 illustrate hypergraphs with unproductive cycles. The first figure represents a step from the inhabitation process, where a word cannot be found. In this example, the input is the repository presented in Figure 4.10 and the second target type $Pos(2*0)$ denoted by $\bigstar_2$ (cf. Figure 4.9). The inhabitation algorithm cannot produce any results, so a message in the *Result Overview* indicates the non-inhabitation. Only using the step-wise visualisation can users can consider when and where the problem occurs. Moreover, in the graphical representation, it is easy to comprehend, why these nodes are marked. Obviously, the computed tree grammar

contains a cyclic rule, such that no edge can ensure the exit of the cycles, as shown in the example above.

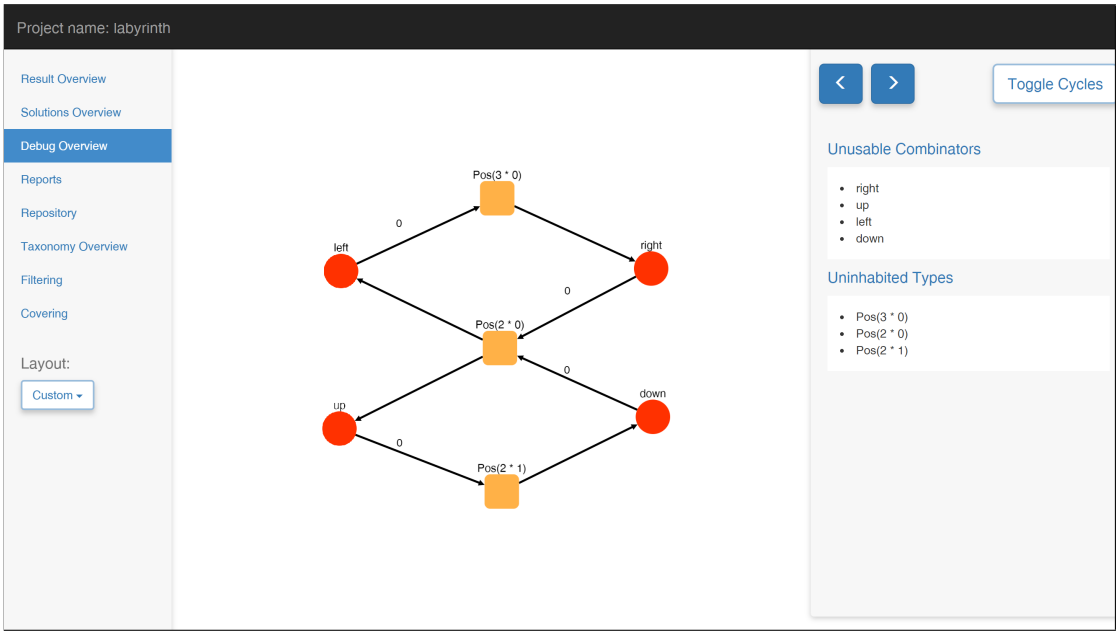

Figure 4.23: Example of cycle

Figure 4.24 displays another example of an unproductive cycle produced by the synthesis of the labyrinth use case presented in Figure 4.6. Here, a step of the searching process with target type $Pos(2*0)$ is illustrated. The example is more complex, but it also serves the evaluation of the developed IDE. The tree grammar generated by the inhabitation algorithm is shown in Figure 4.8. Using the step-wise function, users can investigate which specifications may be problematic. The *Debug Perspective* identifies that there are problem nodes. Let us consider Figure 4.25 to illustrate only the problem part of the hypergraph. For better recognition, the representation is enlarged and the layout is changed manually. In the current inhabitation step, in addition to the nodes in the cycle ($right$ and $left$), the upper combinator node $left$ is also marked in red. This behaviour results from the intersection types rules that lead to the following rule: $Pos(2*0) \rightarrowtail left(Pos(1*0) \cap Pos(3*3))$. The inhabitants for the argument types of combinator $left$ are generated from the presented cycle produced from the following rules:

$$(Pos(1*0) \cap Pos(3*3)) \rightarrowtail \{right(Pos(0*0) \cap Pos(2*3))\},$$
$$(Pos(0*0) \cap Pos(2*3)) \rightarrowtail \{left(Pos(1*0) \cap Pos(3*3))\}$$

Here, it can also be seen that there is no path with an outgoing connection so that the cycle cannot be broken.

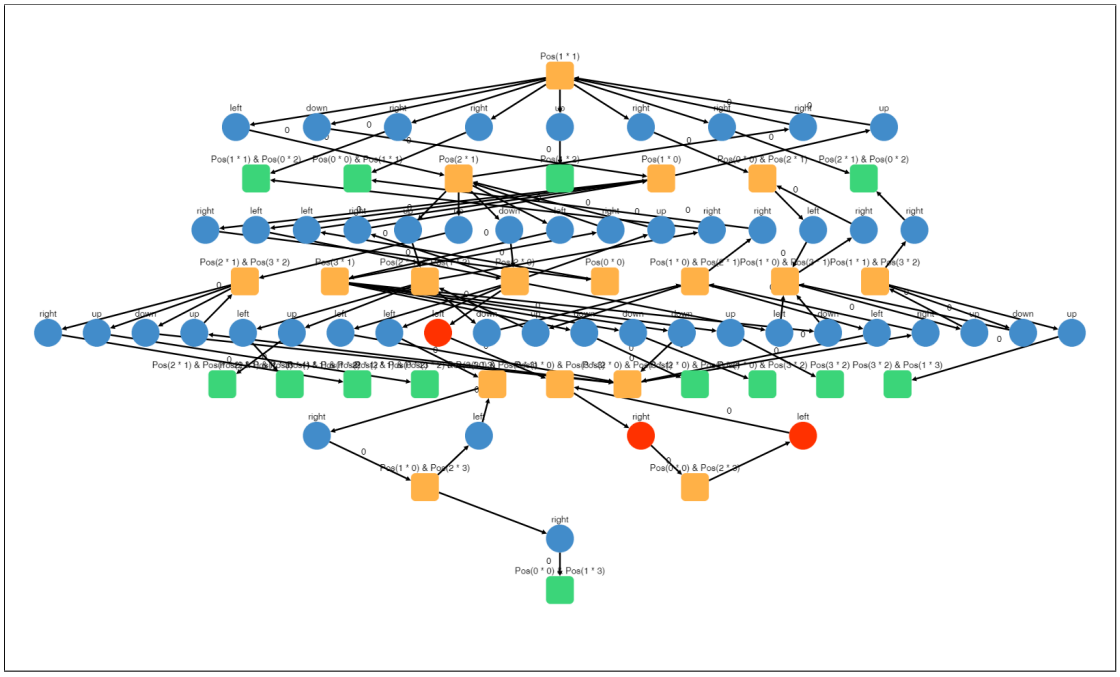

Figure 4.24: Example of unproductive cycle

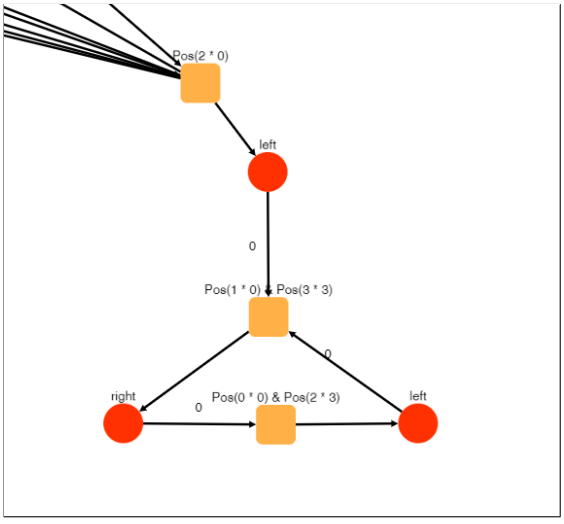

Figure 4.25: Example of unproductive cycle (enlarged)

Figure 4.26 illustrates the solution formed from the same repository as in the example considered above and the inhabitation request $Pos(1 * 0)$. As shown, an inhabitation step can contain more than one unproductive cycles. To improve user support, in the *Debugger Overview* perspective, we include the button *Toggle Cycle* to avoid overloaded and unclear representations of the results. The visualisation of unnecessary cyclic rules from the current inhabitation step, which cannot be used for the generation of a solution, can thereby be restricted. A user can

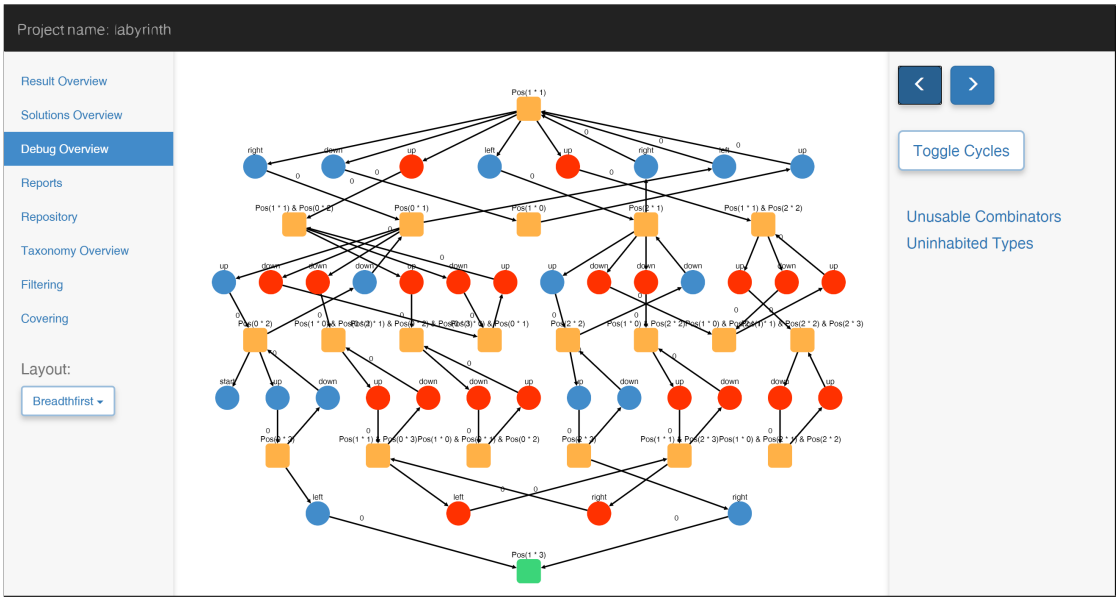

Figure 4.26: Debugger Overview

then investigate the combinators and type nodes that matter for the final result. The complete tree grammar visualisation can be toggled back with the same button.

If any problems have occurred during the search process, brief information about the eventual problems appears on the right side in the *Debugger Overview* perspective. As Figure 4.26 displays, under the toggle-button the options *Unusable Combinators* and *Uninhabited Types* can be selected. A list of combinators that cannot be used can be listed by selecting the button *Unusable Combinators*. Usually, combinators cannot be used because of their specification. If types cannot be inhabited, they will be listed after selection of the button *Uninhabited Types*. For such types, no combinators, which can be used, are available in the repository.

Moreover, with brief information about problems that can occur, a section *Warnings* informs the user about deficient specifications of combinators (see Figure 4.27). This notification is visible only if in the specification of a combinator, the number of the native and semantic types is not the same [37]. Figure 4.27 pictures an enlarged part for such a problem occurred in the search process. The example is from the student project presented in [13]. Here, the layout is also manually changed. As can be seen, an unusable type node is marked by a yellow square with rounded edges and a red border. The repository of this use case includes numerous combinators alongside dynamically generated combinators. In such complex applications, the overview over the correctness of the specifications can become lost. This specification deficiency is an additional example evidencing that user support is required during the implementation of complex application cases.

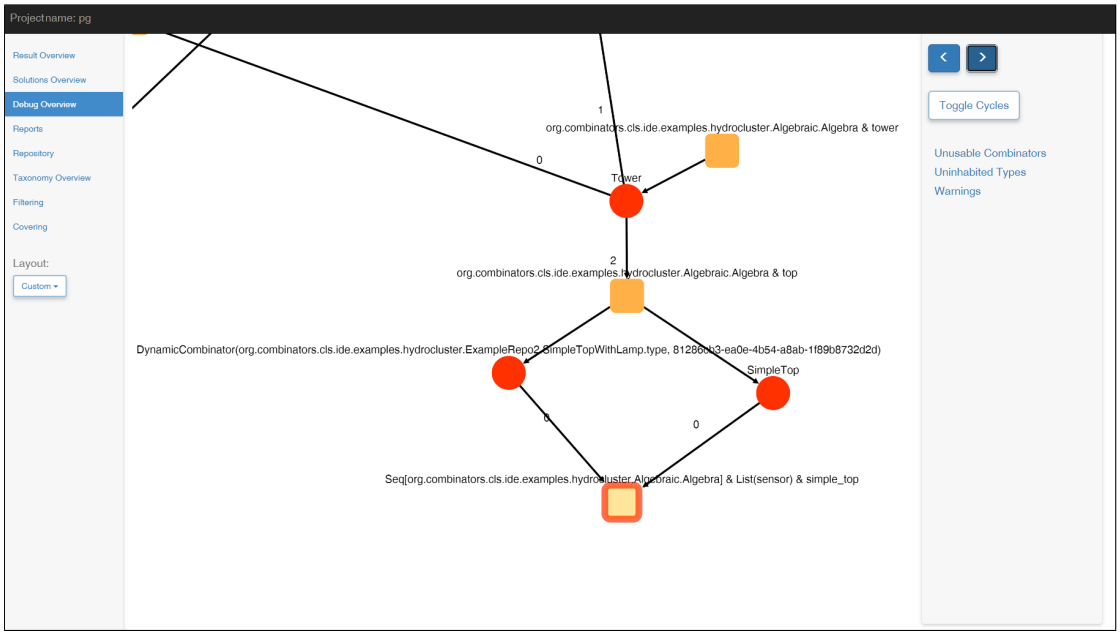

Figure 4.27: Unusable type

4.3.5 Reports

The visualisation of tree grammars computed by the (CL)S Framework can be obscure and sometimes uninformative. The figures in Section 4.3.4 present examples of complex tree grammars. In addition to the features that support the analysis of the search process, the IDE provides a *Reports* perspective. The information appearing here benefits the user's understanding of the inhabitation, particularly when there is no solution, when combinators cannot be used, or when unproductive cycles have occurred during the search process. As noted, concise problem information appears in the *Debugger Overview*. In the *Reports* perspective, the user gets detailed textual information about the problems and the location of the problem specifications. The information about the deficiencies that can occur is grouped according to the following three categories:

- **Uninhabited Types:** Types that cannot be inhabited are listed on the top of the perspective. The information concerning each uninhabited type encountered during the search processes is represented textually. Figure 4.28 displays a part of the list with uninhabited types resulting during the inhabitation of the labyrinth example presented in Figure 4.6, with target type $Pos(2 * 0)$. The argument type of the upper combinator $left$ presented in Figure 4.25 is also listed as uninhabitable. If the inhabitation is unsuccessful, the given target type will also be listed in this perspective.

- **Unused Combinators:** In this part of the *Reports* perspective, users receive information about the unused combinators. As mentioned, there is usually a problem with the

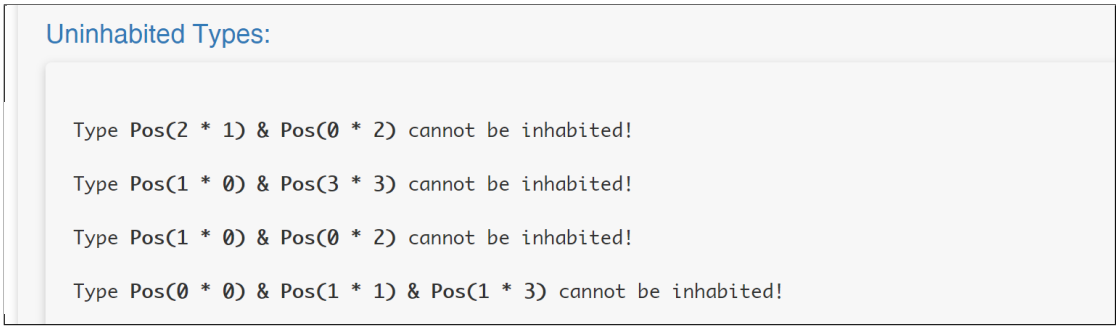

Figure 4.28: Representation of information about uninhabited types

quality of the input specification. Very often, simple typos can lead to non-inhabitation. In such a case, the poorly defined combinator can be found in this perspective. Figure 4.29 displays certain textual information regarding the unused combinators. Here, we also consider the example presented above with the same target type. The second combinator in the list is the upper $left$ combinator marked in red in Figure 4.25. As mentioned, it cannot be used because of the occurrence of an unproductive cycle. In Figure 4.28, we already saw that its argument type is in the list of uninhabitable types, which is also why the combinator cannot be used if the target type is $Pos(2*0)$. From this perspective, each listed combinator is represented with the current target and the type that cannot be inhabited. In this way, if there is a problem with the argument, the reason for non-inhabitation can be retraced.

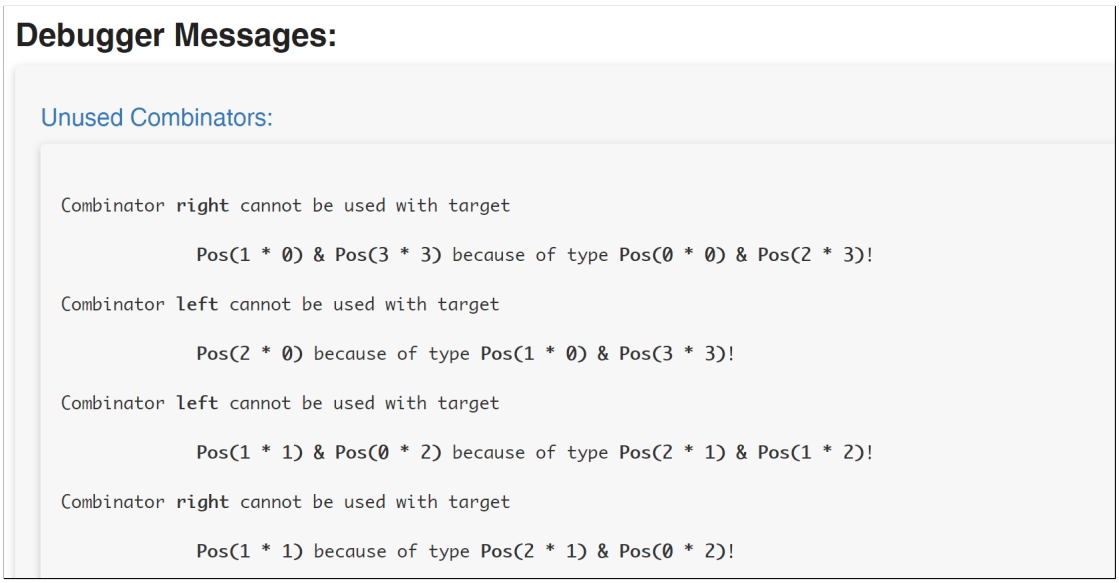

Figure 4.29: Representation of information about unused combinators

- **Warnings:** As mentioned, very often the results are unexpected because of the different number of native and semantic types within a specification of a combinator. In such

a case, the IDE provides information in the form of warnings. If in the step-wise visualisation such problems are detected, they are also listed in a *Warning* section in the *Reports* perspective, as shown in Figure 4.30. Here, the type specification of the example presented in Section 6.3 is listed.

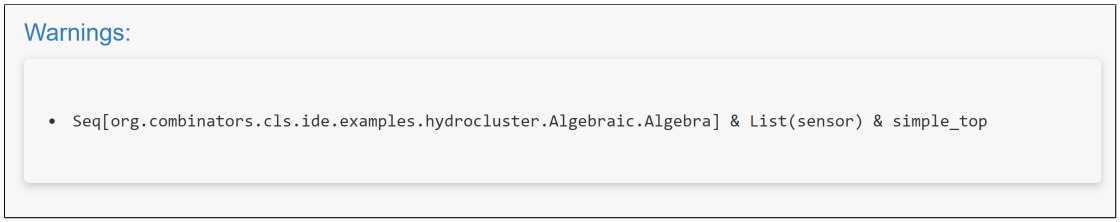

Figure 4.30: Representation of warnings

4.3.6 Repository

The *Repository* perspective presents users an overview of the currently used repository specification. As mentioned in Section 2.5.2, types can be extended by variables. If variables are used, a kinding declaration is defined at the beginning of the Scala program. The well-formed substitution belongs to the input specification. The *Repository* perspective allows an overview of the repository with variables as well as of the well-formed (without variables) representation. Figure 4.31 shows a repository without variables from the labyrinth example in Figure 4.7. In the example associated with this figure, variables are not used.

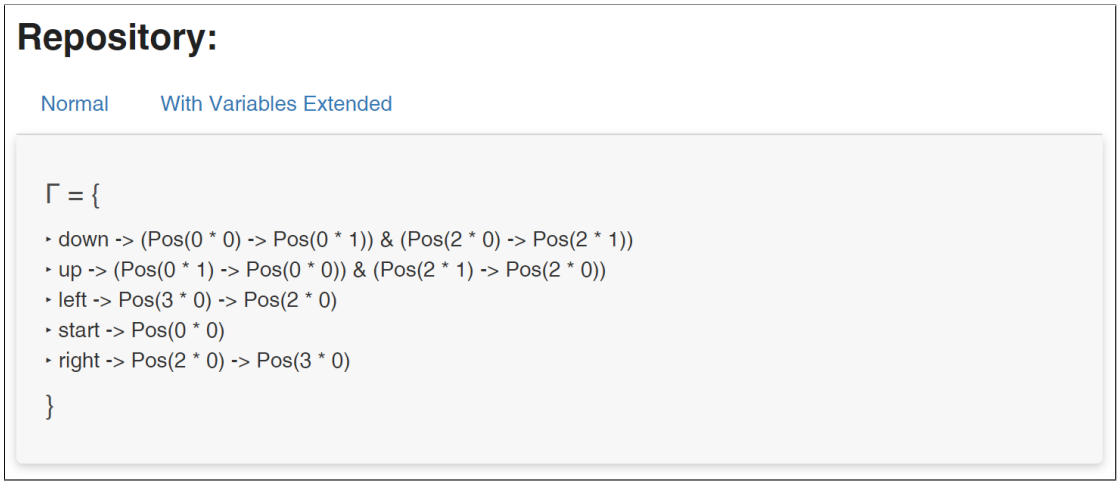

Figure 4.31: Repository representation

Using this perspective, the users can investigate and verify whether the programmatically constructed repository specification is correct. In the presented example, the repository is constructed from a two-dimensional Boolean array. Of course, the combinators can be entered manually, as explained in Section 2.5.2 . Regardless of how the user chooses to construct the

repository, unexpected problems can occur. Typos in the specifications are a typical example of such problems. From this perspective, a quick check of the specification correctness is provided. Changes can be made in the local IDE as usual.

4.3.7 Taxonomy Overview

Section 2.5.4 has considered the specification of the subtype environment. The *Taxonomy Overview* perspective presents a visualisation of these specifications. As mentioned, the taxonomy specifications combine the subtype relation with the semantic types. This perspective summarizes the supertypes in the form of a list. Listing 4.5 displays the programmatic specification of taxonomies. The Scala implementation presented in Section 2.5 is extended by two different subtype relations. The first is represented by the supertype `Direction` and the subtypes `Left`, `Right`, `Up`, and `Down`. The second one is the specification of supertype `Robot` defined to describe the kind of robot that can go from the start to the go position. According to the specification, a robot can be a vehicle or humanoid. Figure 4.32 illustrates the representation of the taxonomy specifications in the IDE.

```
lazy val taxonomy =
    Taxonomy (" Direction ")
        . addSubtype (" Left ")
        . addSubtype (" Right ")
        . addSubtype (" Up ")
        . addSubtype (" Down ")
        .merge(Taxonomy (" Robot ")
            . addSubtype (" Car ")
            . addSubtype (" Humanoid "))
```

Listing 4.5: Representation of taxonomy information

Figure 4.32: Representation of the supertypes.

Users can discover the subtype relations through consideration of a graphical representation. The visualisation presented in Figure 4.33 will be constructed in the *Taxonomy Overview* perspective with the selection of the supertype `Direction` from the list presented above.

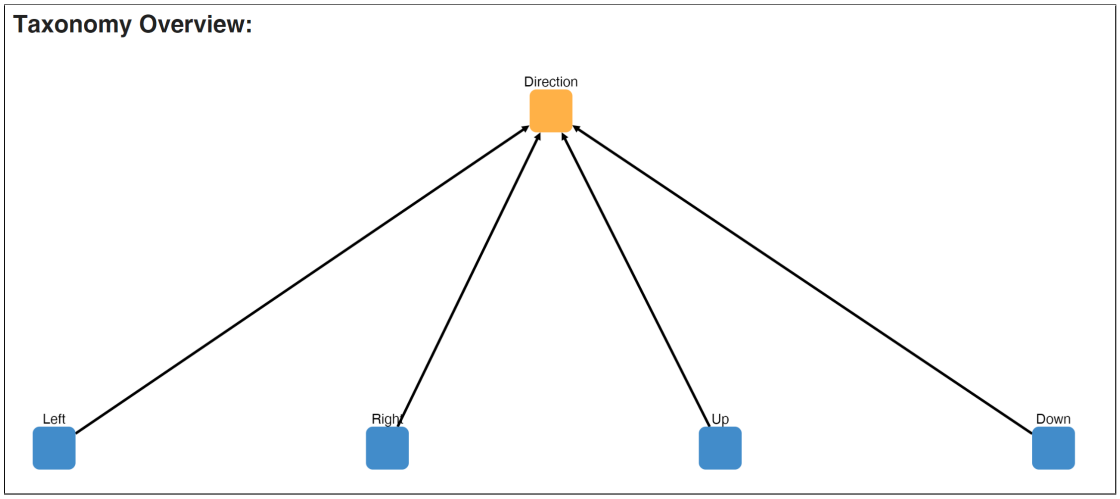

Figure 4.33: Representation of the subtype relation.

As demonstrated, the graphical visualisation of taxonomy specifications is, in general, similar to the representation of inheritance using UML notation [43]. Here, the yellow square with rounded edges denotes the supertype, and the blue squares with rounded edges denote the subtypes. The arrows represent the relation.

This perspective is useful for the implementation of the specifications in cases of numerous relations defined. For example, in the complex use case presented in [13], a visual representation of the taxonomy warrants additionally a better user support. Figure 4.34 illustrates all taxonomy specifications within the project for synthesising of cyber physical systems [13] that can be easily investigated using this feature of the IDE.

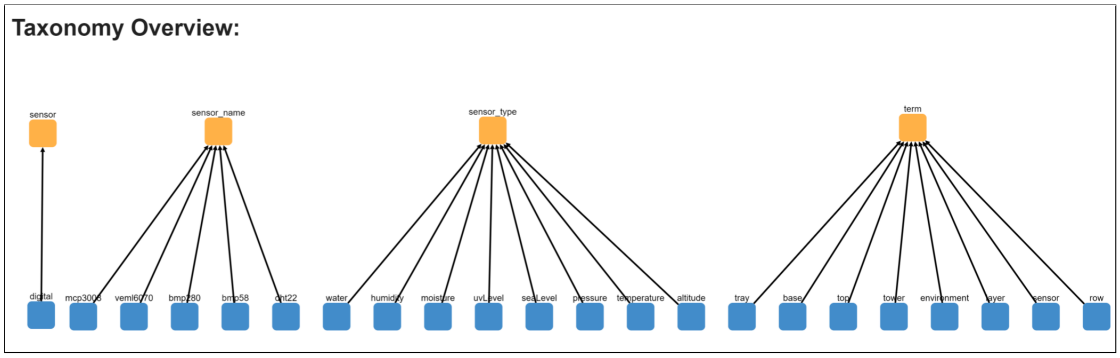

Figure 4.34: Representation of the subtype relation.

4.3.8 Filtering

In this section, we present the function of the filtering approach presented in Section 3.3 in the (CL)S IDE. As mentioned, the filtering algorithm to be considered here modifies the resulting tree grammar computed by the (CL)S Framework without recursion. The filtering approach works based on predefined constraints that can be specified by users. In this context, the *Filtering* perspective in the IDE for (CL)S Framework provides a text field where users can enter a desired pattern to restrict it from the solutions computed by (CL)S Framework (see Figure 4.35). After pressing the *Filter* button, the implemented filtering algorithm modifies the original tree grammar according to the rules presented in Section 3.3.1. The modified tree grammar that does not include the user-specified pattern is also represented by a hypergraph placed below the text field. A pruning algorithm ensures that the visualised tree grammar does not contain unusable rules. If there is an inhabitant with the target type, it is listed under the visualised tree grammar, as with the representation of the inhabitants shown in Figure 4.18. Moreover, in this perspective, the download feature mentioned in Figure 4.18 is provided if the use case supports the generation of files. Figure 4.35 presents the *Filtering* perspective using the labyrinth example offered in Figure 4.9, with target $Pos(0 * 1)$. To restrict the cycle in the original tree grammar, the pattern `down(up(*))` is defined.

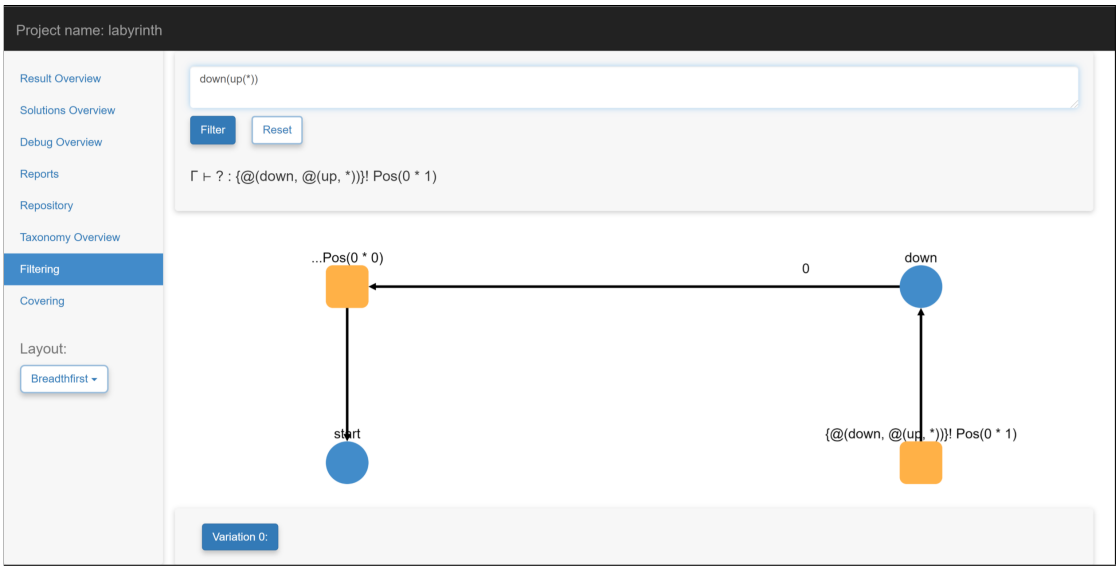

Figure 4.35: Filtering perspective

$$\mathcal{G} = \{@(down, @(up, *))!Pos(0 * 1) \mapsto (down(@(down, @(up, *)), @(up, *)!Pos(0 * 0))), @(down, @(up, *)), @(up, *)!Pos(0 * 0) \mapsto start\}$$

Figure 4.36: Modified tree grammar with pattern `down(up(*))`

Figure 4.36 presents the tree grammar produced by the filtering algorithm. As compared to the original algorithm, the pruned algorithm leaves us two rules in total. The algorithm inserts nonterminals with unique names based on the given pattern. The new target type is modified to `{@(down, @(up, *))}! Pos(0 * 1)`. The second type visualised in the hypergraph is labelled by $\ldots Pos(0 * 0)$ because of the clarity of the representation. The full name of the type can be displayed when the mouse hovers on the type node (cf. the representation of the type `{@(down, @(up, *))}! Pos(0 * 1)`); otherwise, only the original name of the type can be seen. The rule, which corresponds to the target type, cannot construct a term using the combinators `down` and `up` in that order to remove the cycle. That way, by the restriction of certain order of combinators' usage, the infinite number of trivial solutions can be avoided. In this case, the algorithm computes only the following word as a possible inhabitation result:

$$Tree(down, \{@(down, @(up, *))\}!\, Pos(0 * 1),\\ List(Tree(start, \{@(down, @(up, *)); @(up, *)\}!Pos(0 * 0), List()))).$$

The *Filtering* perspective provides the option to restrict more than one pattern. In this case, the patterns should be separated using the following symbol: |. For example, the entry

$$down(up(*)) \mid up(down(*))$$

in the text field means that the patterns $down(up(*))$ and $up(down(*))$ must be restricted. In this case, the algorithm also computes the new target and returns the new tree grammar in the form of a hypergraph. In this example, the new target type contains the bots patterns

$$\{@(up, @(down, *))\}!\{@(down, @(up, *))\}!Pos(0 * 1).$$

In the labyrinth example, such a pattern leads to an empty result set because all valid paths are restricted.

According to the scope of this example, the application of the filtering approach is not so interesting because the results are manageable. It is useful for more complex use cases because of the challenging enormity of the result set. For example, if the number of solutions is infinite, then in some cases, finiteness can be achieved by applying the patterns presented in Section 3.3.2.

4.3.9 Covering

To support the understanding of the search process, the IDE provides a possibility for a manual investigation of the type covering machine presented in [37]. This representation also supports the users through the development of repositories and increasing the quality of the specifications. That way, users can avoid so-called *dead code* [94] or in other words, unusable specifications. Certain combinators and their types can be investigated step-by-step

in the *Covering* perspective. Thus, the covering of the given target type can be proved. The covering support works based on the given type. As such, the coverage of the given type can be investigated. If the user wants to prove another type, then the target type must be changed according to the new type desired. Initially, all combinators from the repository are listed. Selecting a combinator reveals whether it can cover the target and whether additional specifications are required. Similar to [105], the algorithm constructs paths according to the type specification and groups these paths by their length. The IDE presents an overview of paths of arguments for the selected combinator (see Figure 4.37). For example, if we have the following combinator,

$$c : \sigma_1 \to \sigma_2 \to \tau \cap \sigma_1 \to \gamma,^1$$

the algorithm computes for the given combinator type the following paths:

$([::],\ \sigma_1 \to \sigma_2 \to \tau \cap \sigma_1 \to \gamma)$	with 0 arguments
$(([\sigma_1],\ \sigma_2 \to \tau), ([\sigma_1], \gamma))$	with 1 argument
$([\sigma_1,\ \sigma_2], \tau)$	with 2 arguments.

The visualisation of the paths includes, in the first place, a list of arguments and, in the second place, the target. The user must select the number of arguments, given by the computation of the algorithm. If the user selects paths with one argument, then the paths presented in Figure 4.37 are shown.

Figure 4.37: Path covering visualisation

[1] Parentheses are unnecessary because the intersection binds stronger than arrows so that type $(\sigma_1 \to \sigma_2 \to \tau \cap \sigma_1 \to \gamma)$ is equal to $((\sigma_1 \to \sigma_2 \to \tau) \cap (\sigma_1 \to \gamma))$.

All types, which must be covered for the inhabitation of the given type, will be listed in the section *to Cover*. Each target path that is not equal to or greater than a selected path will be shown into *to Cover* section. In this case, with a selection of path $([\sigma_1],\ \gamma)$, only the target path tau is shown in the *To Cover* section because $\gamma \leq \gamma$ that means γ is covered by the chosen path and γ is neither equal to nor greater than τ. If the section *To Cover* is empty, the selected combinator can be used for the inhabitation of the target type. If the framework does not return the expected inhabitation result, the user can discover the covering of the single combinators using this feature and add new combinators, change, or augment consisting one.

4.4 Critical Review

The usability into the scope of software development is a well-studied topic that can be considered as a separate research field. The developed IDE represents a basis for understanding combinatory logic with intersection types. The application of the IDE by nonexperts in the type theory shows it to be useful for the detection of specification deficiencies that can lead to unexpected results or non-results. For such a user group, this feature is significant because the IDE provides a more understandable representation of the search process. Chapter 6 presents certain examples in which the features of the IDE have proven useful. The application of the IDE to real use cases has shown that representing the results with graphs can be challenging within complex problems, where the (CL)S component repository includes hundreds of combinators. However, the provided step-wise construction of the hypergraphs and the separate visualisation of individual terms compensate for these disadvantages in such cases. Moreover, despite the unclear representation, the textual information about deficiencies in the specifications and in the results provides a significant support [13].

Very often, the usage of frameworks is associated with unnecessary effort. For example, one disadvantage can be dependency on other applications or operating systems or complicated installation procedures, where the versions of the components also play an essential role. This work aims primarily to provide an IDE that not only supports users in the field of type theory but also functions intuitively and accessibly. For this reason, platform independence was an essential part of the implementation. Moreover, in this way, complicated installation steps can be avoided. This feature is an advantage over all applications, in which the installation is based on such procedures. Furthermore, surveys in the field of usability [61; 120; 71] show that this aspect is essential to motivate users to apply the framework again later. If an application is easy to use, users are more inclined to reuse it. Otherwise, if users must deal with complex installation procedures, their motivation and interest dwindle. The number of additional technologies required by an application can also have a deterrent effect on users. The developed web-based IDE does not increase the number of technologies that must be used. That way, developers do not have to study new software or IDEs to specify the inhabitation input or to set up the application. They can continue to use the local Scala IDE that is already using and a browser that is also used by every Scala developer. Furthermore, the

implementation of the IDE allows for the implementation of more domain-specific addition extensions, so providing such functionalities by default goes against one of the main aims of this work, namely – the domain-independent generic realisation of user support for the (CL)S Framework. The mentioned advantages of the web-based application allow for wise dissemination and availability. These features are essential to attract different user groups and to increase user motivation. That is why these aims impact the selection of the technologies.

Chapter 5

Evaluation

The main goal of the implementation of the web-based (CL)S IDE was to increase the usability of the (CL)S Framework. The topic of usability within this work relates to the provision of features that support the understanding of the inhabitation algorithm and its behaviour, and not to the usability of a software. In the latter case, investigation should be based on measurement with surveys and with a large user group to achieve representative results. To investigate the usability of the web-based IDE, we provided the implementation to other software developers who were nonexperts in type theory. The applications and their impact are presented in Chapter 6. In this charter, a performance evaluation is presented. The application cases and the tests are also publicly available online [16]. The performance of the (CL)S Framework is discussed in [29]. The filtering algorithm integrated in the IDE is an important aspect that affects the efficiency of the framework. Section 5.1 provides an overview of the measurements of the performance of the filtering approaches presented in Section 3.2 and Section 3.3 followed by discussion of the results and justification of the decision on the applied algorithm.

5.1 Filtering Performance

This section considers the performance of the filtering approaches presented in Section 3.2 and Section 3.3. The first algorithm, based on SMT theories (see Section 3.1), will be not handled here for the reasons outlined in Section 3.1.3. The presented argumentation indicates the approach to be unsuitable for our application cases. There are two implementations of the filtering approach based on the modification of the tree grammars that will be investigated in this section. To recognise which is more suitable to the IDE, we measured the behaviour and the performance using a labyrinth example similar to that presented in [30]. All measurements were carried out on a Windows computer with an Intel i7-5500U (2.40 GHz) processor and 8GB RAM, and they were based on one and the same application case. In the following, we illustrate the application case, and thereafter, the results are presented and discussed.

For the evaluation, we considered a labyrinth example, where several iterations were executed. In each iteration, the start position was top-left and the goal bottom-right. Figure 5.1 illustrates the labyrinth example of size 3×3. An advantage of this example is the easy analysis of the scaling problem of the filtering approaches, increasing the size of the labyrinth systematically. The first measurement was with a labyrinth of size 5×5. Per iteration, we

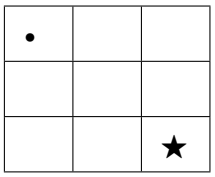

Figure 5.1: Labyrinth example 3×3

analysed the behaviour of the labyrinth with and without any obstacles. We measured the time of transformation of the original tree grammar generated by the (CL)S Framework to a filtered tree grammar computed by the algorithms. The obstacles are placed randomly. The inhabitation stops if a path from start to goal is generated. If this is not the case, the obstacles are repositioned, and the inhabitation starts with the new repository. With each iteration, the size of the labyrinth was increased, whereby the size of 18×18 represents the final iteration. More measurements were unnecessary because the results sufficiently established a clear tendency.

As mentioned, the results computed by (CL)S Scala Framework can include trivial terms such as:

$$down(up(down(up...))).$$

Such paths are considered as correct. However, to find the shortest possible path, it is counterproductive to go forwards and then backwards so that the pattern applied in the following experiments was $p = down(up(*,*),*)$. Below, the results of the performance measurement are presented.

The first line in the tables presents the size of the original grammar $\mathscr{G}$ computed by the inhabitation algorithm. Based on these results, the second line presents the size of the modified grammars $\mathscr{G}'$ generated by each filtering algorithm considered. Thereafter, the size of the pruned grammar is listed. As mentioned, applying a pruning algorithm is required because the filtering algorithms produce rules that are not usable or reachable. The time required to find an inhabitant is measured in seconds. The generation of labyrinths without obstacles leads to the computation of repositories with a larger number of combinators. It also yields, respectively, tree grammars with a large number of rules. Using the modification approaches, the number of rules in the tree grammars increases depending on the given pattern. Table 5.1 compares the resultant data from the measurement with the 5 × 5 labyrinth without obstacles. The algorithms begin with a tree grammar with 32 production rules. The size of the tree grammar modified by the filtering algorithm without recursion is four times larger than the original. The other algorithm produces a tree grammar almost three times larger. However, there is a significant difference (about 32%) between the numbers of rules in the modified tree grammars. The pruning algorithm halves the number of results in the respective grammars, but the difference remains minimal. Notably, the performance time of both the algorithms is similar; algorithm **A** needs 0.083 seconds and algorithm **B** – 0.085 seconds.

Table 5.1: Performance for 5 × 5 labyrinth without obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	32	
New Tree Grammar $\mathscr{G}'$ Size	128	86
Size after Pruning	64	47
Time (s)	0.083	0.085

The results for a labyrinth with size 5 × 5 with obstacles are shown in Table 5.2. As can be seen, the size of the tree grammar within the examples without obstacles is larger than for those ones in the examples with obstacles, since more valid paths are possible in the first case. Accordingly, the same observation applies for the modified tree grammars and for the pruned grammars. In this example, the tree grammar generated by algorithm **A** is larger than this one generated by algorithm **B**. However, the filtering algorithm without recursion performs faster than the other one.

Considering the results in Table 5.3, where the results for an example without obstacles are outlined, we can see that the original grammar is larger, leading to modified tree grammars with 648 (in case **A**) and 476 (in case **B**) rules. Here, the pruning algorithm also removes about the half of the rules, so the tendency remains unchanged. This stability also applies to the time results – filter **A** is faster. Table 5.4 outlines the results for a 10 × 10 labyrinth with obstacles. Here, the same behaviour as in the examples for the smaller labyrinth is observable.

Table 5.2: Performance for 5 × 5 labyrinth with obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	15	
New Tree Grammar $\mathscr{G}'$ Size	60	31
Size after Pruning	29	18
Time (s)	0.03	0.07

Significantly, the size of the modified tree grammar is great, but the algorithms needed for the filtering take between only 0.01 and 0.02 seconds. The difference is not significant, but here algorithm **A** performs better once more.

Table 5.3: Performance for 10 ×10 labyrinth without obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	162	
New Tree Grammar $\mathscr{G}'$ Size	648	476
Size after Pruning	324	247
Time (s)	0.15	0.19

Table 5.4: Performance for 10 ×10 labyrinth with obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	87	
New Tree Grammar $\mathscr{G}'$ Size	348	245
Size after Pruning	172	130
Time (s)	0.01	0.02

Tables 5.5 and 5.6 display the results of the last iteration with labyrinths of a size 18 × 18. We investigated examples of sizes 12 × 12 and 15 × 15, but there were no significant results, and for the sake of the readability, these measurements are not considered because the tendency stays stable. The size of the modified tree grammars is based on the original grammar and on the size of the pattern so that the results are unsurprising. The performance time is notable. In both cases, the algorithm without recursion performs again faster than the other, despite the size of the resulting grammar.

However, the investigation shows that the differences in the time taken during the perfor-

Table 5.5: Performance for 18 × 18 labyrinth without obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	578	
New Tree Grammar $\mathscr{G}'$ Size	2312	1702
Size after Pruning	1428	868
Time (s)	0.32	0.44

Table 5.6: Performance for 18 × 18 labyrinth with obstacles

	Filter Algorithm (A)	**Filter Algorithm with Recursion (B)**
Tree Grammar $\mathscr{G}$ Size	175	
New Tree Grammar $\mathscr{G}'$ Size	700	481
Size after Pruning	333	238
Time (s)	0.02	0.05

mance of the filtering algorithms are not significant. The filtering algorithm without recursion performs better than the other in almost all cases. Algorithm **B** shows a clear advantage over the other one, without recursion, in terms of the size of the new tree grammars. It generates tree grammars with notably fewer rules than the first algorithm. The results have shown that, on average, the algorithm with recursion produces tree grammars with 25% to 38% fewer rules. Despite the production of a larger number of rules by the filtering algorithm without recursion (algorithm **A**), we have estimated that this approach does not display significant disadvantages as compared to the other algorithm. The size of the tree grammars produced by this algorithm can be considered disadvantageous, but this fact matters only to the graphical visualisation. Obviously, a graph representation with 1 428 rules differs from one with 868 rules (cf. Table 5.5, size after pruning). The results show that in examples with labyrinths of size more than 10 × 10 and pattern p, the algorithm with recursion requires longer than the other. It is somewhat surprisingly that with larger application cases, the difference becomes significant.

Additionally, the performance was investigated with the following patterns:

$$p_s = down(up(*,*),*) \text{ and } p_l = down(down(up(down(up(*,*),*),*),*),*).$$

As expected, both filtering algorithms perform better with the pattern p_s and need more time with p_l. In the case with pattern p_s, the algorithm without recursion performs better than the other in all cases. The difference in the individual test cases is between 0.005 and 0.04 seconds. In the second case, with pattern p_l, the algorithm with recursion performs better. For an 18

× 18 labyrinth and p_l as input, algorithm **A** performs in 1.65 seconds and algorithm **B** under one second. These findings are not surprising. The algorithm without recursion produces a grammar with 18 496 (before the pruning) rules. The output of the other algorithm has only 578 rules. This number of rules is about 32 times less.

Certain advantages of the filtering implementations based on modifying tree grammars without recursion over the CLS-SMT approach and the filtering approach with recursion are as follows:

- It has a simpler and more straightforward implementation so that the filtering approach also benefits in the field of performance.
- Like the (CL)S Scala Framework, the filtering approach based on modifying tree grammars is implemented in the programming language *Scala*. In this way, the approach facilitates the combination of both technologies.
- There is no additional knowledge about other technologies required so that the maintainability and further development of the approach are easier.
- Additional technologies, such as constraint solver, to increase the complexity of the framework are unnecessary.
- It is not necessary to check the uniqueness of the new names.
- The completeness of the solutions, therefore, remains intact, and it is unnecessary to consider the completeness of other technologies.

Considering the above results, it can be concluded that both filtering algorithms are comparable and can handle tree grammars resulting from type environments with hundreds of combinators representing non-deterministic problems. In the presented experiments with pattern p, the results are available in under one second. However, the filtering algorithm without recursion generates a tree grammar with more rules than the algorithm with recursion, but in these cases, it performs better. In the case with a larger pattern, algorithm **B** performs better than the algorithm without recursion. This difference is still manageable. Compared to the algorithm presented in [39], these filtering algorithms performs better. The authors discuss the 30 × 30 labyrinth example. The linear case of the algorithm performs in 75.719 seconds with the pattern p and 30 × 30 labyrinth without obstacles. In contrast, the performance of the filtering algorithms based on a modification of tree grammars stays under one second. The filtering algorithm with recursion needs 0.85 seconds, and the algorithm without recursion finishes in 0.55 seconds. For the example with the size of 18 × 18 and pattern p, the algorithm presented in [39] computes the results in 3.39 seconds. These measurements were also carried out on the same Windows computer presented at the beginning of this section. Despite the disadvantages discussed, the algorithm without recursion was integrated into the (CL)S IDE because of the formal correctness of the filtering algorithm (cf. Section 3.3).

5.2 IDE Tests

The functionality of the implementation is tested using one of the most common frameworks for testing Scala programs - *ScalaTest* [128]. The tool supports different testing styles that provide flexibility. The applied style within the project introduced in this work was *FunSuit*. Parallel to the testing, a tool for code coverage analysis named *scoverage* [8] was used. This application helped to investigate the quality of the testing. Thus, the tested code development is covered almost 90%. Moreover, a tool for static code analysis was applied to ensure the quality of the program code. Certain rules listed in [112] were used to improve the maintainability of the implementation.

Chapter 6

Applications and Impact

The IDE for the (CL)S Framework is used within multiple projects where the synthesis of different application cases has been implemented successfully. This exploration by various applications helped us investigate and improve the functionalities and thus the usability of the IDE. For example, the use case presented in Section 6.1 has shown that the framework provides support not only in understanding the inhabitation process but also in presenting and analysing the solutions. Moreover, the idea of an optional download feature also resulted from the collaborative work presented in Sections 6.1 and 6.2. According to the feedback from the developers, additional features were implemented that were not part envisioned at the outset of this work. These functionalities could not be considered until confronted with a complex use case. An example of such a functionality is the filtering approach that constitutes a separate project. Inspired by joint work with other researchers, the filtering functionality was integrated into the IDE. Applying the filtering algorithm to the application case presented in Section 6.2, which deals with the synthesis of complex milling processes, specific improvements in the visualisation were also implemented, for example, the visualisation of the short version of the labels of the type nodes that ensures a more precise representation of the results. The taxonomy visualisation and the covering detection resulted from the application case presented in Section 6.3. The graphical visualisation was also improved (cf. Section 2.6). Applying such small use cases as those presented in [30; 37; 85], we investigated the visualisation using compound graphs and hypergraphs. Finally, the chapter ends with a use case that introduces the importance of visualisation of (CL)S results in collaboration work with researchers in logistics (see Section 6.4).

6.1 Automatic Composition of Factory Planning Projects

The dissertation project [133] of Jan Winkels deals with the development of an approach for automatic creation of fabric planning processes. He has investigated subjects such as production, logistics, and manufacturing to ensure the proper generation of such processes. By constraint solving, he restricts the solutions computed by the (CL)S Framework. Within this application case, the following features of the IDE were significant:

- The IDE can be extended according to the use case, with acceptable overhead. Moreover, this aim can be achieved without the support provided by the author of this work.
- The step-wise visualisation allows an explanation of the results' construction that is understandable for engineers without knowledge of programming or type theory.

Within this project, the author was able to implement extensions in the IDE, which were domain-specific. That implementation has shown that the development of additional features is manageable. For example, at the time of writing this work, the filtering algorithm presented in this work was not implemented so that the author implemented a filtering function which is domain-specific. This filtering approach could not be integrated in the IDE because of the use case independence. Another example is the option to download ready-made projects for factory planning. These features compared with the IDE were used to present the results to other researchers within the interdisciplinary project that are part of this work [133].

6.2 Planning of Machining Operations for Components using CAM

Computer-aided Manufacturing (CAM) systems are used to plan complex milling processes. The use of such techniques requires manual review by experts. This process increases the complexity of manufacturing projects. In [114], the authors present a new approach to support the milling processes by using (CL)S. The presented approach generates tool path solutions that support the planning process using CAM software. The results of the synthesis are infinite. Automated evaluation steps pick up solutions that CAM experts can consider in the next step.

The application demonstrates the following useful properties:

- The framework can be extended according to use case, without excessive.
- The IDE provides a convenient way to independently discover certain solutions.
- The filtering of unnecessary solutions supports the development of specifications and increases the quality of the results.
- Use-case-specific problems do not regard type specifications.

- The application of the framework for investigating complex milling processes is possible.

A helpful feature implemented within this project was the possibility of downloading the synthesised results. In this case, the required files can be easily downloaded in the *Solutions* perspective. After applying the filtering function, there is also the possibility to download the desired filtered files. This feature facilitates further use of the synthesised results by the CAM software. The implementation of the interface for the download was kept very simple to enable its application in other use cases where such functionality is required. Within this project, we have used associativity constraints for the restriction of trivial solutions. Together with an investigation of the single terms, a detection of user-specific patterns was possible. This feature helped to achieve better and faster results and to increase clarity for the application case.

6.3 Synthesising of Cyber Physical Systems

A compulsory course within the computer science masters programme at TU Dortmund University requires attendance at a project group, where students in small groups develop software. The students have a year to complete the design, implementation, and of testing the software. One of the projects in 2019 was the evaluation of possibilities for synthesising of cyber physical systems [13]. This project aimed to analyse language-agnostic synthesis. Students were not experts in type theory, but they applied the (CL)S for the synthesis and used the IDE for the analysis of the repository, discovering problems successfully. The application case was based on the configuration of intelligent plant management systems. The integration of abstract syntax trees (ASTs) for OpenSCAD models [86] was part of the development of the solution. The wide range of synthesised solutions included, in addition to the 3D printer models, configuration files and shell scripts, some production plans and abstract machine models [13].

The user support provided by the IDE was especially in the following cases very useful:

- The extension of the (CL)S IDE by the taxonomy representation has proven to be beneficial in complex applications.
- The step-wise visualisation allowed for the discovery of the repositories and detection of deficiencies and faulty specifications.

According to the participants, without the (CL)S IDE, the implementation and debugging of the metamodel synthesis would be much more time-consuming, perhaps even impossible, in the time available for the project. The graphical visualisation often makes visible small as well as serious errors in the dynamically generated combinators, whereas debugging by test scripts and command line only provides comparably little information about what is going

wrong. Thus, the IDE has proven an indispensable tool for implementation. The dynamically generated combinators contribute to the generation of very large graphs, pushing the graph-based representation to its limits. A repository with up to 294 combinators is difficult to visualise adequately. However, the listing of unusable combinators and uninhabitable types, as well as the repository and the step-by-step construction of the solutions in *breadthfirst* layout, allows even these very large cases to become manageable.

6.4 Motion Planning in Logistic Lab Environment

Logistics Research Lab [26] represents a Cyber-Physical-System (CPS) including wheel-driven robots, drones, and laser-based support for a visualisation (e.g., of possible movements or forbidden areas). Figure 6.1 shows the research area. The laser visualisation illustrates the *field of view* of the transport robot and the forbidden area (the circle around the person in the background), where the robot does not have access. This research lab is used from Fraunhofer Institute for Material Flow and Logistics (IML) and from chair of Materials Handling and Warehousing at TU Dortmund University. A collaboration with these researchers in logistics shows

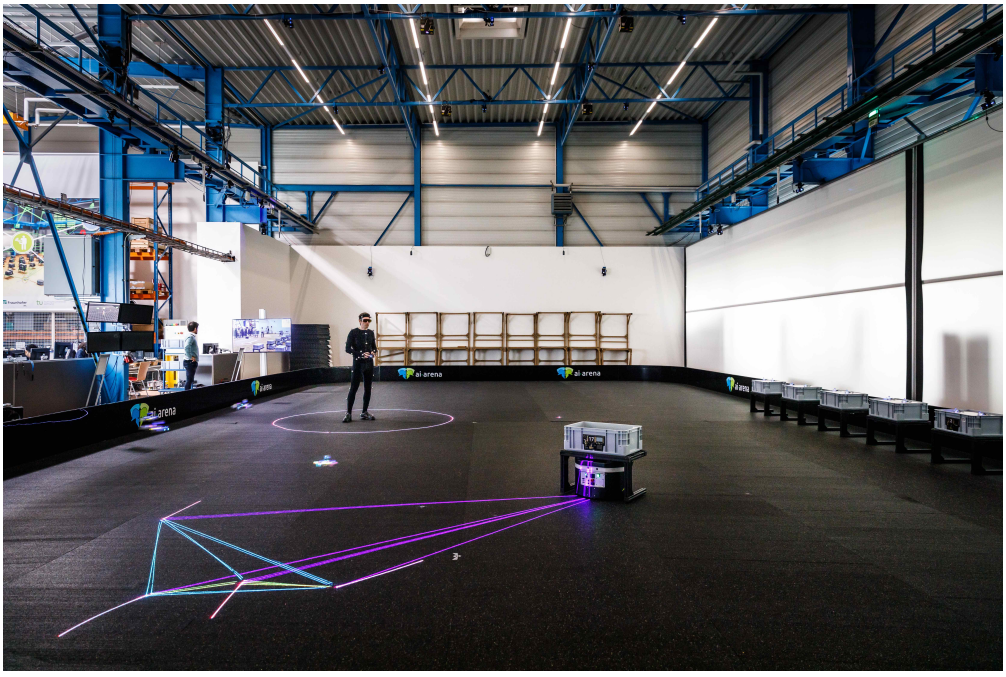

Figure 6.1: Logistics research lab overview

one more time that (CL)S with its IDE can be used from nonexperts in the field of type-theory [38]. The scenario is based on the labyrinth example presented in Section 4.3.1. As mentioned, it was previously used to analyse the performance of the type-based synthesis framework and to explain the tree grammars. Beyond these advantages, this example illustrates the base of

path finding algorithms that are a central problem within autonomous diving. The resultant prototype demonstrates the following properties:

- (CL)S and its IDE can be used by engineers with knowledge in programming, but who are nonexperts in type theory.
- Domain-relevant extensions can be considered without impact on existing technologies.
- The application of the framework in a logistic lab is possible.
- Debugging and testing of the results can be provided that preserve the real objects.
- Use-case-specific problems are regardless of type specifications.
- The application of (CL)S IDE impacts cost-efficiency and is resource-friendly.

Figure 6.2 illustrates the architecture of the applied approach. The first layer represents the syntheses of all possible solutions in a given environment by the (CL)S Framework. The framework synthesises movement commands for robots. The second layer represents a virtual representation of the result. An implemented interface allows the investigation of all possible solutions by a 3D simulation based on Unity 3D game engine [9]. Several observation systems are installed in the lab. The simulation represents not only a simple visualisation of the results but it can also send information to installed laboratory equipment (third layer) through an interface based on a simple MQTT [80] network protocol. One of these systems in the research lab, the laser projection system, can be used to visualise all 3D models in the research area. Another important tool is the Motion Capturing (MoCap) system. Using 40 infrared cameras by Vicon [129], it enables the tracking of all objects and obstacles in the experimentation space. The objects must be suitably marked to allow the recognition not only of simple real objects, but also of persons with an accuracy of 0.3 mm, a rate of 200 Hz, and millisecond latencies depending on the number of objects. Figure 6.1 illustrates a person wearing a marker-suit. Information about the position and rotation of the objects is available in three dimensions. The real objects are part of the fourth layer. [38]

The connection of the different frameworks and the automatisation of the approach was an essential part of the development. An extension is represented by implementing an interface for the communication with the simulation. As mentioned, the ISO-standardised MQTT network protocol is used in the lab for the connection of Unity 3D (C# implementation) with the installed equipment (the programming language depends on the used system). The first layer provides an implementation in Scala. For this reason, we used Eclipse Paho [65] for the connection of Unity 3D with (CL)S. The environments could then be investigated and changed using 3D visualisation. Moreover, the start and end positions can also be modified using the IDE or the Unity 3D representation. These changes can be sent to the (CL)S as input for the following synthesis. Figure 6.3 illustrates the data flow into the architecture that realises the synthesis of operation programs for logistic objects. The use case does not bias the data

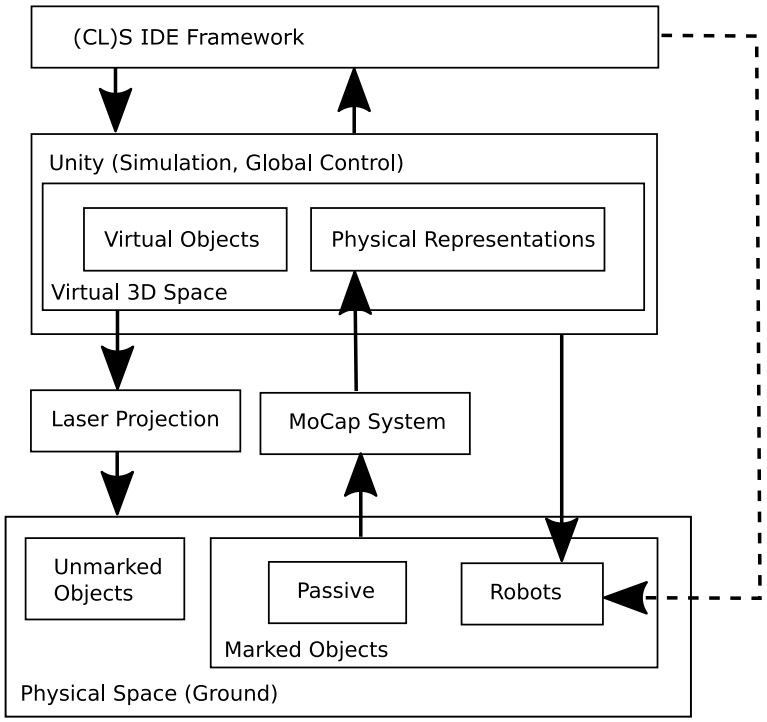

Figure 6.2: Path generation overview [38]

flow between (CL)S and its IDE. Here, the Robotnik [131] is illustrated, but the application of other real objects is possible (e.g., drones, or Loadrunner) [26; 122]. The application of all computed paths by the (CL)S Framework was tested in the logistics research lab using the laser supported visualisation (s. Figure 6.4). Figure 6.5 shows the 3D representation of the environment represented by the laser system using Unity 3D. A video of this experiment is available online [17]. Such testing is protective not only for physical objects such as robots and drones but also for the researcher in the lab. Moreover, it lowers costs. A test with real robots raises the risk of real, expensive damages. For example, the current research considers the development of the Loadrunner. This self-driving high-speed vehicle reaches a speed up to 10 m/s, with acceleration of up to 5 m/s^2 [122]. Such a speed can be not only dangerous for researchers in the lab but also expensive, on account of hardware costs. Accordingly, computer simulations of the behaviour and visualisation with the laser system have to be applied first. The simulation with Unity 3D can be applied regardless of the lab. It ensures the exact implementation of the simulated behaviour in reality so that the paths computed by (CL)S and investigated by the IDE can be simulated by Unity 3D, and thereafter the chosen path can be represented by real physical objects.

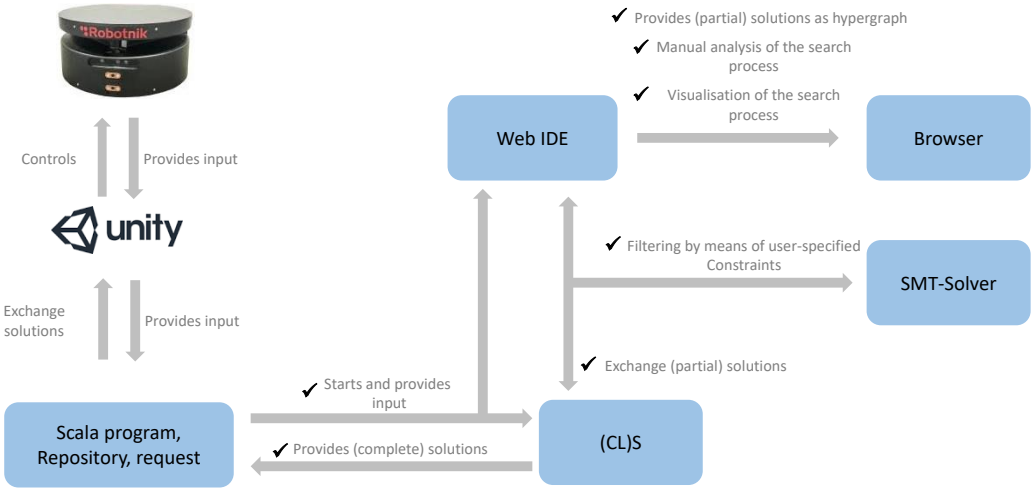

Figure 6.3: Data flow

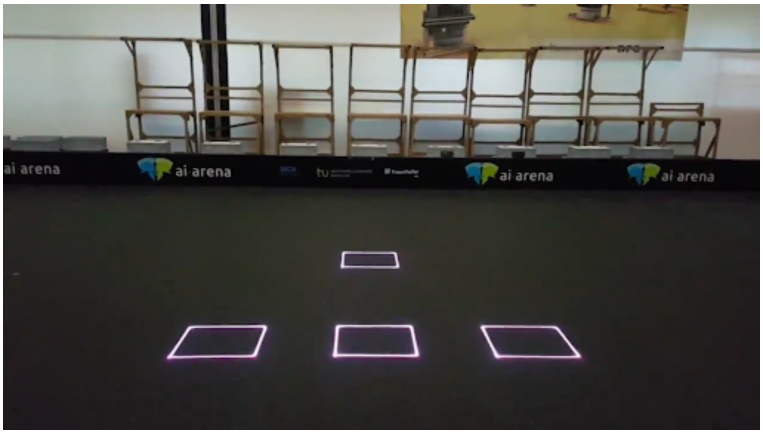

Figure 6.4: Laser Projection System Representation

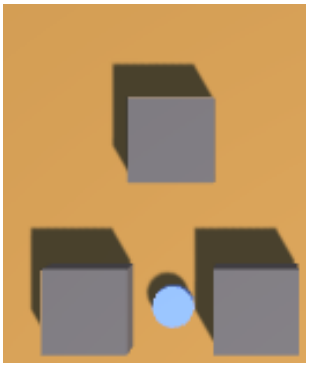

Figure 6.5: Labyrinth example represented by Unity 3D

Chapter 7

Conclusion and Outlook

The usability of software products can increase the motivation to apply these technologies. Therefore, this work provides a solution to improve the usability of software development technologies represented by the (CL)S Framework. It has considered the resulting approach for user support based on the collection of different functionalities, providing an innovative approach to support nonexperts in the field of formal methods. The main features are the debugging of intersection type specifications and the filtering of the solution set represented by inhabitation results. The combination of formal grammars and graph theory represents the foundation of a clear step-by-step graph visualisation in a web-based IDE to provide an additional, user-friendly feature. Several visualisation and filtering approaches were developed and investigated during the research process to provide the one best suited as a web-based IDE. The reason for this process of iteration was the unexpected insights arising during the development and investigation of complex application cases. For example, we have shown that the first filtering approach based on SMT (see Section 3.1) developed within the scope of this work is not always fail-safe [39; 85]. The practical experiments presented in Chapter 5 and Chapter 6 have shown that the algorithm integrated into the (CL)S IDE (see Section 3.3) is practically feasible and represents a clear improvement on other filtering approaches. Functionalities such as overviews of the resulting tree grammar, analysis of individual terms, and textual information about problems that could occur during the search process support the user's understanding of the decisions of the inhabitation algorithm (cf. Chapter 4). Moreover, using application cases (see Chapter 6), we have shown that the IDE provides user support that can increase the quality of the specifications and thus of the inhabitation results.

Some research questions appropriate for future work arise from this project. For example, the application to other software development technologies resulting in tree grammars can be analysed to investigate the usability of related frameworks using hypergraphs, beyond software synthesis based on combinatory logic with intersection types. For example, consider a representation of the framework presented by Feng et al. [64], where if an extension using

hypergraphs is implemented, then the IDE presented in this work can be used as an additional feature for visualisation and debugging.

Another possible area of future work is the extension of the functionalities of the IDE by a subtyping representation. The results of the subtype machine presented in [37] can be visualised step-wise. Moreover, the visualisation of the taxonomies using *Hasse diagrams* can be discovered [44; 47]. A filtering approach with recursion might be considered to better visualise the resulting filtered tree grammars. The investigation presented in Section 5.1 shows that this algorithm performs in some complex application cases faster than the algorithm without recursion. Furthermore, the number of the rules in the resulting grammar is less than the other filter algorithm based on modification, which is a clear advantage for the IDE visualisation. This feature may compensate for the disadvantages when the performance is not so good. Other approaches for pattern filtering may include a restriction on graph elements. The idea is to forbid a pattern by removing combinator nodes and the edges corresponding to it. For example, consider the tree grammar $\mathscr{G}$, as presented in Figure 7.1, and the corresponding graph visualisation, in Figure 7.2.

$$\begin{aligned}\mathscr{G} = \{&\sigma_0 \mapsto c_1(\sigma_1, \sigma_2),\\ &\sigma_1 \mapsto c_2(),\\ &\sigma_2 \mapsto c_3(\sigma_3),\\ &\sigma_3 \mapsto c_4(), c_5(\sigma_0)\}\end{aligned}$$

Figure 7.1: Tree grammar $\mathscr{G}$

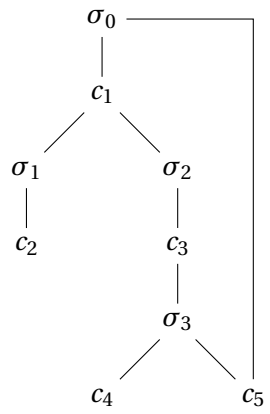

Figure 7.2: Graph visualisation of $\mathscr{G}$

If the restriction combinator c_5 is desired, then the node that represents this combinator can be deleted from the hypergraph. Thus, this activity results in the tree grammar $\mathscr{G}'$ presented in Figure 7.3 and a graph visualisation presented in Figure 7.4, where combinator c_5 does not occur. A disadvantage of this approach is that it requires more rigorous and in-depth research in the field of graph theory. Moreover, an analysis of the languages $\mathscr{L}(\mathscr{G})$ and $\mathscr{L}(\mathscr{G}')$ is required because it is not obvious whether the restriction removes words that do not include the pattern.

Additional to the *Covering* perspective, a static analysis of the input specifications can be developed to increase the quality of the repository and to avoid bad inhabitation (non)results. In the context of code quality in Java programs, a rule set was developed to avoid *code smells* [93]. According to this approach, a rule set for the specification of the repository Γ can be developed to provide a more formal understanding and insights on intersection types. Finally,

$$\mathscr{G}' = \{\sigma_0 \rightarrowtail c_1(\sigma_1, \sigma_2),$$
$$\sigma_1 \rightarrowtail c_2(),$$
$$\sigma_2 \rightarrowtail c_3(\sigma_3),$$
$$\sigma_3 \rightarrowtail c_4()\}$$

Figure 7.3: Filtered tree grammar $\mathscr{G}'$

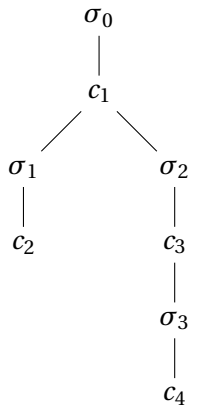

Figure 7.4: Graph visualisation of $\mathscr{G}'$

the application of the IDE for the (CL)S Framework by a large group of nonexperts can be investigated to prove its usability. To achieve this aim, measurement techniques similar to those presented in [111] can be used. Here, the approach is constituent of the regular university computer science education, such that per iteration, much information can be collected and compared using a survey. Representative results can thereby be achieved.

Bibliography

[1] Bootstrap 4. https://www.w3schools.com/bootstrap4/bootstrap_get_started.asp. Accessed: 11.12.2021.

[2] Akka HTTP. URL https://doc.akka.io/docs/akka-http/current/index.html. Accessed: 11.12.2021.

[3] Bootstrap. https://getbootstrap.com/. Accessed: 11.12.2021.

[4] CVC4. URL https://cvc4.github.io/index.html. Accessed: 11.12.2021.

[5] Isabelle. URL https://isabelle.in.tum.de/. Accessed: 11.12.2021.

[6] scala-parser-combinators. URL https://github.com/scala/scala-parser-combinators. Accessed: 11.12.2021.

[7] Scala Build Tool (SBT). https://www.scala-sbt.org/. Accessed: 11.12.2021.

[8] scoverage. URL https://github.com/scoverage/scalac-scoverage-plugin. Accessed: 11.12.2021.

[9] Unity 3D. URL https://docs.unity3d.com/Manual/UnityManual.html. Accessed: 11.12.2021.

[10] HVC 2010 - Haifa Verification Conference 2010, 2007. URL https://www.research.ibm.com/haifa/conferences/hvc2010/award.shtml. Accessed: 11.12.2021.

[11] Guice, 2020. URL https://github.com/google/guice. Accessed: 11.12.2021.

[12] Graph for Scala | Graph for Scala - Home, 3.4.2020. URL http://www.scala-graph.org/. Accessed: 11.12.2021.

[13] P. 624. Synthese Cyberphysischer Systeme. *Technische Universität Dortmund, intern*, 2020. in German.

[14] M. D. Adams and M. Might. Restricting grammars with tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):82, 2017.

[15] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, Oct 2013. doi: 10.1109/FMCAD.2013.6679385.

[16] Anna Vasileva and Jan Bessai. cls-scala-ide. URL https://github.com/combinators/cls-scala-ide. Accessed: 11.12.2021.

[17] Anna Vasileva and Moritz Roidl. Laser Demonstration. URL https://github.com/combinators/labyrinth. Accessed: 11.12.2021.

[18] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon System for Verification, Synthesis and Repair. URL http://leon.epfl.ch/. Accessed: 11.12.2021.

[19] E. J. G. Arias, B. Pin, and P. Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. In *Proceedings of the 12th Workshop on User Interfaces for Theorem Provers, UITP 2016, Coimbra, Portugal, 2nd July 2016.*, pages 15–27, 2016. URL https://doi.org/10.4204/EPTCS.239.2.

[20] G. Ausiello and G. F. Italiano. On-line algorithms for polynomially solvable satisfiability problems. *The Journal of logic programming*, 10(1):69–90, 1991.

[21] G. Ausiello, P. G. Franciosa, and D. Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In *Italian conference on theoretical computer science*, pages 312–328. Springer, 2001.

[22] K. Bar, A. Kissinger, and J. Vicary. Globular: an online proof assistant for higher-dimensional rewriting. *Logical Methods in Computer Science*, 14(1), 2018. URL https://doi.org/10.23638/LMCS-14(1:8)2018.

[23] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983. doi: 10.2307/2273659.

[24] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, 2017. Available at `www.SMT-LIB.org`.

[25] T. Bauernhansl, M. Ten Hompel, and B. Vogel-Heuser. *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung-Technologien-Migration.* Springer, 2014. In German.

[26] H. Bayhan, A. Karthik Ramachandran Venkatapathy, J. Dregger, F. Zeidler, M. Roidl, and M. ten Hompel. A Concept of an Industry 4.0 Research Lab for Future Intralogistics Technologies and Services. *3rd Interdisciplinary Conference on Production, Logistics and Traffic, ICPLT*, 2017.

[27] E. A. Bender and S. G. Williamson. *Lists, Decisions and Graphs.* S. Gill Williamson, 2010.

[28] C. Berge. *Graphs and hypergraphs.* North-Holland, 1973.

[29] J. Bessai. A Type-Theoretic Framework for Software Component Synthesis. 2019.

[30] J. Bessai and A. Vasileva. User Support for the Combinator Logic Synthesizer Framework. *Electronic Proceedings in Theoretical Computer Science*, 284:16–25, 2018. doi: 10.4204/EPTCS.284.2.

[31] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Logic Synthesizer. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, pages 26–40, 2014. URL https://doi.org/10.1007/978-3-662-45234-9_3.

[32] J. Bessai, B. Düdder, G. T. Heineman, and J. Rehof. Combinatory Synthesis of Classes Using Feature Grammars. In *Revised selected papers of the 12th International Conference on Formal Aspects of Component Software*, pages 123–140, 2015. URL https://doi.org/10.1007/978-3-319-28934-2_7.

[33] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Process Synthesis. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 266–281, 2016. URL https://doi.org/10.1007/978-3-319-47166-2_19.

[34] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Process Synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, pages 266–281, 2016. doi: 10.1007/978-3-319-47166-2_19.

[35] J. Bessai, T.-C. Chen, A. Dudenhefner, B. Düdder, U. de'Liguoro, and J. Rehof. Mixin Composition Synthesis based on Intersection Types. *Logical Methods in Computer Science*, Volume 14, Issue 1, Feb. 2018. doi: 10.23638/LMCS-14(1:18)2018. URL https://lmcs.episciences.org/4319.

[36] J. Bessai, B. Düdder, G. T. Heineman, et al. (CL)S Framework, 2018. URL http://www.combinators.org. Accessed: 2018-04-30.

[37] J. Bessai, J. Rehof, and B. Düdder. Fast Verified BCD Subtyping. In *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *Lecture Notes in Computer Science*, pages 356–371. Springer, Cham, [S.l.], 2019. ISBN 978-3-030-22347-2. doi: 10.1007/978-3-030-22348-9_21.

[38] J. Bessai, M. Roidl, and A. Vasileva. Experience Report: Towards Moving Things with Types–Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis. *arXiv preprint arXiv:1912.10628*, 2019.

[39] J. Bessai, L. Czajka, F. Laarmann, and J. Rehof. Restricting tree grammars with term rewriting. *7th International Conference on Formal Structures for Computation and Deduction, FSCD*, 2022.

[40] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[41] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system: verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 1:1–1:10, 2013. URL http://doi.acm.org/10.1145/2489837.2489838.

[42] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with Why3. *STTT*, 17 (6):709–727, 2015. URL https://doi.org/10.1007/s10009-014-0314-5.

[43] G. Booch. *The unified modeling language user guide*. Pearson Education India, 2005.

[44] R. Brüggemann, A. Kaune, J. Klein, and R. Zellner. Anwendung der Hasse-Diagrammtechnik. *Umweltwissenschaften und Schadstoff-Forschung*, 8(2):89–96, 1996. (In German).

[45] J. Carme, J. Niehren, and M. Tommasi. Querying unranked trees with stepwise tree automata. In *International Conference on Rewriting Techniques and Applications*. Springer, 2004.

[46] S. Chasins and J. L. Newcomb. Using SyGuS to Synthesize Reactive Motion Plans. *arXiv preprint arXiv:1611.07620*, 2016.

[47] C. Chevalley et al. Helmut Hasse, Über die Klassezahl abelscher Zahlkörper. *Bulletin of the American Mathematical Society*, 59(3):281–282, 1953. (In German).

[48] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2): 56–68, 1940. doi: 10.2307/2266170.

[49] H. Comon and F. Jacquemard. Ground reducibility is EXPTIME-complete. *Information and Computation*, 187(1):123–153, 2003.

[50] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available online: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[51] M. Coppo and M. Dezani-Ciancaglini. A new type assignment for λ-terms. *Arch. Math. Log.*, 19(1):139–156, 1978. URL https://doi.org/10.1007/BF02011875.

[52] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[53] A.-H. Dediu, R. Klempien-Hinrichs, H.-J. Kreowski, and B. Nagy. Contextual hypergraph grammars–a new approach to the generation of hypergraph languages. In *International Conference on Developments in Language Theory*, pages 327–338. Springer, 2006.

[54] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[55] T. Dietze. Equivalences between Ranked and Unranked Weighted Tree Automata via Binarization. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, pages 1–10, 2016.

[56] U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper. Efficient methods and readily customizable libraries for managing complexity of large networks. *PloS one*, 13(5), 2018.

[57] B. Düdder. *Automatic Synthesis of Component & Connector-Software Architectures with Bounded Combinatory Logic*. PhD thesis, Technische Universität Dortmund, Fakultät für Informatik, Dortmund, 2014, 2014.

[58] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Bounded Combinatory Logic. In P. Cégielski and A. Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. ISBN 978-3-939897-42-2. doi: 10.4230/LIPIcs.CSL.2012.243. URL http://drops.dagstuhl.de/opus/portals/extern/index.php?semnr=12009.

[59] B. Düdder, J. Rehof, and G. T. Heineman. Synthesizing Type-Safe Compositions in Feature Oriented Software Designs Using Staged Composition. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 398–401, 2015. doi: 10.1145/2791060.2793677. URL http://doi.acm.org/10.1145/2791060.2793677.

[60] A. Dudenhefner. *Algorithmic aspects of type-based program synthesis*. PhD thesis, 2019.

[61] Elmar/P/Wach. Aus welchen Gründen haben Sie einen bestimmen eShop ausgewählt?[For what reason did you choose a particular eShop], 2011. URL https://de.statista.com/statistik/daten/studie/188807/umfrage/gruende-der-kunden-fuer-die-auswahl-von-online-shops/. Accessed: 11.12.2021.

[62] J. Engelfriet and L. Heyker. Context-Free Hypergraph Grammars have the Same Term-Generating Power as Attribute Grammars. *Acta Inf.*, 29(2):161–210, 1992. URL https://doi.org/10.1007/BF01178504.

[63] A. P. Felty and A. Middeldorp, editors. *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, 2015. Springer. ISBN 978-3-319-21400-9. URL https://doi.org/10.1007/978-3-319-21401-6.

[64] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex APIs. pages 599–612, 2017. URL http://dl.acm.org/citation.cfm?id=3009851.

[65] E. Foundation. Paho. URL https://www.eclipse.org/paho/. Accessed: 11.12.2021.

[66] J. Frankle, P.-M. Osera, D. Walker, and S. Zdancewic. Example-directed Synthesis: A Type-theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 802–815, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837629. URL http://doi.acm.org/10.1145/2837614.2837629.

[67] M. Franz, C. T. Lopes, G. Huck, Y. Dong, S. O. Sümer, and G. D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309–311, 2016. URL https://doi.org/10.1093/bioinformatics/btv557.

[68] N. Franzese, A. Groce, T. Murali, and A. Ritz. Hypergraph-based connectivity measures for signaling pathway topologies. *PLoS computational biology*, 15(10), 2019.

[69] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees. In *18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings.* IEEE, 2003.

[70] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. Directed hypergraphs and applications. *Discrete applied mathematics*, 42(2-3):177–201, 1993.

[71] Google. Important features of mobile websites according to smartphone users in Canada as of March 2015 [Graph], 2015. URL https://www.statista.com/statistics/438350/features-mobile-websites-canada/. Accessed: 11.12.2021.

[72] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of PLDI'11*, pages 62–73. ACM, 2011.

[73] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet Data Manipulation using Examples. volume 55, pages 97–105, January 2012. URL https://www.microsoft.com/en-us/research/publication/spreadsheet-data-manipulation-using-examples/. Invited to CACM Research Highlights.

[74] S. Gulwani, A. Polozov, and R. Singh. *Program Synthesis*, volume 4. NOW, August 2017. URL https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[75] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 27–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462192. URL http://doi.acm.org/10.1145/2491956.2462192.

[76] G. T. Heineman, J. Bessai, B. Düdder, and J. Rehof. A Long and Winding Road Towards Modular Synthesis. In *Leveraging Applications of Formal Methods, Verification and*

Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, pages 303–317, 2016. doi: 10.1007/978-3-319-47166-2_21.

[77] P. Hilton, E. Bakker, and F. Canedo. *Play for Scala: Covers Play 2*. Manning Publications Co., 2013.

[78] J. R. Hindley and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction*, volume 13. Cambridge University Press Cambridge, 2008.

[79] J. E. Hopcroft, R. Motwani, and J. D. Ullman. Introduction to automata theory, languages, and computation. 2006.

[80] International Organization for Standardization (ISO). ISO/IEC 20922:2016: Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1. *ISO: Geneva, Switzerland*, pages 1–73, 2016.

[81] ISO 9241-11:2018. Ergonomics of human-system interaction – Part 11: Usability: Definitions and concepts. Standard, International Organization for Standardization, Geneva, CH, 2018.

[82] JetBrains. IntelliJ IDEA. URL https://www.jetbrains.com/idea/. Accessed: 11.12.2021.

[83] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-Guided Component-Based Program Synthesis. May 2010. URL https://www.microsoft.com/en-us/research/publication/oracle-guided-component-based-program-synthesis/.

[84] C. Kaliszyk. Web Interfaces for Proof Assistants. *Electr. Notes Theor. Comput. Sci.*, 174(2): 49–61, 2007. URL https://doi.org/10.1016/j.entcs.2006.09.021.

[85] F. Kallat, T. Schäfer, and A. Vasileva. CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories. *arXiv preprint arXiv:1908.09481*, 2019.

[86] Kintel, Marius. OpenSCAD - The Programmers Solid 3D CAD Modeller. URL https://openscad.org/about.html. Accessed: 11.12.2021.

[87] H. Kreowski. A Comparison Between Petri-Nets and Graph Grammars. In *Graphtheoretic Concepts in Computer Science, Proceedings of the International Workshop WG '80, Bad Honnef, Germany, June 15-18, 1980*, pages 306–317, 1980. URL https://doi.org/10.1007/3-540-10291-4_22.

[88] H.-J. Kreowski, S. Kuske, and A. Lye. Transformation of petri nets into context-dependent fusion grammars. In *International Conference on Language and Automata Theory and Applications*, pages 246–258. Springer, 2019.

[89] P. Krill. JavaFX will be removed from the Java JDK. URL https://www.infoworld.com/article/3261066/java/javafx-will-be-removed-from-the-java-jdk.html. Accessed: 11.12.2021.

[90] O. Laurent. Intersection Subtyping with Constructors. In *Proceedings DCM 2018 and ITRS 2018*, pages 73–84, Oxford, UK, 2018. doi: 10.4204/EPTCS.293.6.

[91] Lightbend. Play framework. URL https://www.playframework.com. Accessed: 11.12.2021.

[92] S. Liu, W. Cui, Y. Wu, and M. Liu. A survey on information visualization: recent advances and challenges. *The Visual Computer*, 30(12):1373–1393, 2014.

[93] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[94] R. C. Martin. *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG, 2013.

[95] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4): 424–453, 1996.

[96] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[97] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[98] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88, 1990. doi: https://doi.org/10.1016/0890-5401(90)90013-8.

[99] J. Nielsen. *Usability Engineering*. Elsevier Science, 1994.

[100] Oracle. JavaFX. URL http://www.oracle.com/technetwork/java/javase/overview/javafx-overview-2158620.html. Accessed: 11.12.2021.

[101] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 522–538, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908093. URL http://doi.acm.org/10.1145/2908080.2908093.

[102] S. Ranise and C. Tinelli. The SMT-LIB format: An initial proposal. In *Proceedings of the 1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03), Miami, Florida*, pages 94–111, 2003.

[103] J. Rehof. Towards Combinatory Logic Synthesis. In *BEAT 2013, 1st International Workshop on Behavioural Types*. ACM, 2013.

[104] J. Rehof and P. Urzyczyn. Finite Combinatory Logic with Intersection Types. In C. L. Ong, editor, *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, volume 6690 of *Lecture Notes in Computer Science*, pages 169–183. Springer, 2011. ISBN 978-3-642-21690-9. doi: 10.1007/978-3-642-21691-6_15. URL http://dx.doi.org/10.1007/978-3-642-21691-6_15.

[105] J. Rehof and P. Urzyczyn. Finite Combinatory Logic with Intersection Types. In *Typed Lambda Calculi and Applications - 10th International Conference, TLCA 2011, Novi Sad, Serbia, June 1-3, 2011. Proceedings*, pages 169–183, 2011. URL https://doi.org/10.1007/978-3-642-21691-6_15.

[106] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In *International Conference on Computer Aided Verification*, pages 640–655. Springer, 2013.

[107] A. Reynolds, V. Kuncak, C. Tinelli, C. Barrett, and M. Deters. Refutation-based synthesis in SMT. *Formal Methods in System Design*, Feb 2017. ISSN 1572-8102. doi: 10.1007/s10703-017-0270-2. URL https://doi.org/10.1007/s10703-017-0270-2.

[108] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-642-63859-6. doi: 10.1007/978-3-642-59126-6.

[109] T. Schäfer, F. Möller, A. Burmann, Y. Pikus, N. Weißenberg, M. Hintze, and J. Rehof. A methodology for combinatory process synthesis: process variability in clinical pathways. In *International Symposium on Leveraging Applications of Formal Methods*, pages 472–486. Springer, 2018.

[110] D. Schmedding and A. Vasileva. Integration von Qualitätsaspekten in einen Entwicklungsprozess. In *Konferenz zur Software und IT Messung und Bewertung, MetriKon*, 2015. In German.

[111] D. Schmedding and A. Vasileva. Reviews-ein Instrument zur Qualitätsverbesserung von UML-Diagrammen. In *SEUH*, pages 8–19, 2017. (In German).

[112] D. Schmedding, A. Vasileva, and J. Remmers. Clean Code-ein neues Ziel im Software-Praktikum. In *SEUH*, pages 81–91, 2015.

[113] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische annalen*, 92(3-4):305–316, 1924.

[114] T. Schäfer, J. A. Bergmann, R. G. Carballo, J. Rehof, and P. Wiederkehr. A Synthesis-based Tool Path Planning Approach for Computer Aided Manufacturing. *54th CIRP Conference on Manufacturing Systems*, 2021.

[115] S. Skhiri dit Gabouje and E. Zimanyi. A new compound graph layout algorithm for visualizing biochemical networks. In *Poster Proceedings Volume of the 4th International Workshop on Efficient and Experimental Algorithms*. CTI Press, 2005.

[116] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 281–294, 2005.

[117] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006.

[118] D. Spath, O. Ganschar, S. Gerlach, M. Hämmerle, T. Krause, and S. Schlund. *Produktionsarbeit der Zukunft-Industrie 4.0*, volume 150. Fraunhofer Verlag Stuttgart, 2013. In German.

[119] Y. Srikant and P. Shankar. *The compiler design handbook: optimizations and machine code generation*. CRC Press, 2007.

[120] Statista. "Inwiefern ist der Aspekt "Nutzerfreundlichkeit/Bedienbarkeit" bei Onlineshops für Sie persönlich besonders wichtig?"[To what extent is the aspect "usability" of online shops particularly important for you personally?], 2017. URL https://de.statista.com/prognosen/784769/umfrage-zu-nutzerfreundlichkeit-im-internet-als-kriterium-fuer-fashion-onlineshops. Accessed: 11.12.2021, (In German).

[121] K. Sugiyama and K. Misue. Visualization of structural information: Automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(4): 876–892, 1991.

[122] M. ten Hompel, H. Bayhan, J. Behling, L. Benkenstein, J. Emmerich, G. Follert, M. Grzenia, C. Hammermeister, H. Hasse, D. Hoening, et al. Technical Report: LoadRunner®, a new platform approach on collaborative logistics services. *Logistics Journal: nicht referierte Veröffentlichungen*, 2020(10), 2020.

[123] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. TRANSIT: Specifying Protocols with Concolic Snippets. *SIGPLAN Not.*, 48(6):287–296, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462174. URL http://doi.acm.org/10.1145/2499370.2462174.

[124] B. Unhelkar. *Verification and validation for quality of UML 2.0 models*, volume 2. Wiley Online Library, 2005.

[125] A. Vasileva and D. Schmedding. Clean Java–Von Anfang an! *Programmiersprachen und Grundlagen der Programmierung, KPS*, 2015. (In German).

[126] A. Vasileva and D. Schmedding. Vom Clean Model zum Clean Code. *Modellierung 2016*, 2016. (In German).

[127] A. Vasileva and D. Schmedding. How to improve code quality by measurement and refactoring. In *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 131–136. IEEE, 2016.

[128] B. Venners, G. Berger, and C. C. Seng. ScalaTest. URL https://www.scalatest.org/getting_started_with_fun_suite. Accessed: 11.12.2021.

[129] Vicon. Motion Tracking Devices. URL https://www.vicon.com/. Accessed: 11.12.2021.

[130] J. Warren. A hierarchical basis for reordering transformations. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 272–282, 1984.

[131] R. Wiki. Documentation-ros wiki. *URL: http://www. ros. org*, 2021.

[132] J. Winkels. Automatisierte Komposition und Konfiguration von Workflows zur Planung mittels kombinatorischer Logik. 2019.

[133] J. Winkels, J. Graefenstein, T. Schäfer, D. Scholz, J. Rehof, and M. Henke. Automatic composition of rough solution possibilities in the target planning of factory planning projects by means of combinatory logic. In *International Symposium on Leveraging Applications of Formal Methods*, pages 487–503. Springer, 2018.