View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Eldorado - Ressourcen aus und für Lehre, Studium und Forschung

A Type-Theoretic Framework for Software Component Synthesis

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

Jan Bessai

Dortmund

2019

Tag der mündlichen Prüfung: 29. Oktober 2019
Dekan: Prof. Dr.-Ing. Gernot Fink
Gutachter:

Prof. Dr. Jakob Rehof (Technische Universität Dortmund, Deutschland)
Prof. Dr. George T. Heineman (Worcester Polytechnic Institute, USA)

Acknowledgments

First, I'd like to thank my advisor, Prof. Jakob Rehof, who gave me the chance to get this far. There is no other way to say it: without your patience and believe, none of this would have been possible. My gratitude is not only for the job, guidance, and encouragement, but also for the freedom and the awesome time I was able to have with you and your group: from my first student seminar talk on the history of functional programming, which was almost 10 years ago, up until now it has been the most well-typed experience.
A huge part of this is also thanks to Prof. George Heineman. No week could be bad when we had our video conference. Hacking together on some problem via Skype was only to be topped by hacking together live. Thank you, for your kindness, encouragement, humor, the hospitality at your place, which I will never forget, all the things was able to learn, and also for taking this last step of my doctoral journey with me. For the last part I would of course also like to thank the other committee members, Prof. Peter Padawitz and Prof. Falk Howar.
Much of this wonderful time would not have been half as good without my former colleague, Prof. Boris Düdder, who is now in Denmark, and who has shared most of the way with me. Apart from all the support – especially when I was freshly starting out –, and all the things I was able to learn, thank you also for the great company during all of our travels together.
Going back even longer than 10 years on my personal academic journey in Dortmund, I also want to express my deep gratitude toward my mentor, Prof. Ernst-Erich Doberkat, who has been there from the first lecture in the very first semester until now. It was during a student employment at his former chair, when I first came in contact with functional programming, during my Bachelor's thesis, which he co-supervised, when I first learned about category theory, and his category theory lectures together with the type theory lectures by Prof. Rehof, and the lecture on abstract algebra lecture by Prof. Padawitz are what shapes large parts of the formal development in this text.
I'd also like to thank Prof. Ugo de'Liguoro for our discussions and the collaboration, which inspired many thoughts to find a starting point.
On the professional and personal level, I want to thank all my colleagues at the chair. This especially includes Andrej Dudenhefner, Anna Vasileva, and Tristan Schäfer who were so often the few people understanding where my thoughts lead me. It also especially includes Ute Joschko, who literally saved me from unemployment, when the HR department lazy-evaluated the renewal of my contract.
Finally, this list would be incomplete without my parents and friends. With all the fun, parts of this journey have been, I really look forward to being mentally and physically back with you. Thank you for all the support and thank you for being so patient!

Abstract

A language-agnostic approach for type-based component-oriented software synthesis is developed from the fundamental principles of abstract algebra and Combinatory Logic. It relies on an enumerative type inhabitation algorithm for Finite Combinatory Logic with Intersection Types (FCL) and a universal algebraic construction to translate terms of Combinatory Logic into any given target language. New insights are gained on the combination of semantic domains of discourse with intersection types. Long standing gaps in the algorithmic understanding of the type inhabitation question of FCL are closed. A practical implementation is developed and its applications by the author and other researchers are discussed. They include, but are not limited to, vast improvements in the context of synthesis of software product line members. An interactive theorem prover, Coq, is used to formalize and check all the theoretical results. This makes them more reusable for other developments and enhances confidence in their correctness.

Zusammenfassung

Es wird ein sprachunabhängiger Ansatz für die typbasierte und komponentenorientierte Synthese von Software entwickelt. Hierzu werden grundlegende Erkenntnisse über abstrakte Algebra und kombinatorische Logik verwendet. Der Ansatz beruht auf dem enumerativen Typinhabitationsproblem der endlichen kombinatorischen Logik mit Intersektionstypen, sowie einer universellen algebraischen Konstruktion, um Ergebnisterme in jede beliebe Zielsprache übersetzen zu können. Es werden neue Einblicke gewonnen, wie verschiedene semantische Domänen des Diskurses über Softwareeigenschaften miteinander verbunden werden können. Offene Fragestellungen im Zusammenhand mit der Algorithmik des Typinhabitationsproblems für Intersektionstypen werden beantwortet. Eine praktische Implementierung des Ansatzes wird entwickelt und ihre bisherigen Anwendungen durch den Autor und andere Wissenschaftler werden diskutiert. Diese beinhalten starke Verbesserungen im Zusammenhang mit der Synthese von Ausprägungen von Software Produktlinien. Ein interaktiver Theorembeweiser wir genutzt, um alle Ergebnisse der Arbeit zu formalisieren und mechanisch zu überprüfen. Dies trägt zum einen zur Wiederverwendbarkeit der theoretischen Ergebnisse in anderen Kontexten bei, und erhöht zum andern das Vertrauen in ihre Korrektheit.

Contents

List of Figures

Chapter 1

Introduction

In 1918's Vienna, Ludwig Wittgenstein finished his most famous work, the "Logisch-philosophische Abhandlung" that is also known as "Tractatus logico-philosophicus" [202], or Tractatus for short. It was a reaction to the – in his opinion – confused state of Philosophy, where many problems resulted from getting lost in the limitations of language. The ideas presented in this text are motivated by summer heated discussions about detailed limitations of object-oriented languages with Ugo de'Liguoro, Jakob Rehof, Andrej Dudenhefner, Boris Düdder, and Moritz Martens at the Vienna Summer of Logic 2014 [119], and have been made fully manifest with its publication 100 years after the Tractatus in 2019. A core motivation is to address the – in the author's opinion – confusing state of software synthesis, where many problems result from getting lost in limitations of programming languages. Statement 6.002 of the Tractatus postulates:

> "Ist die allgemeine Form gegeben, wie ein Satz gebaut ist, so ist damit auch schon die allgemeine Form davon gegeben, wie aus einem Satz durch eine Operation ein anderer erzeugt werden kann."

Translated to English, this means all sentences are constructed by applying operations to other sentences. This idea is not only fundamental to the philosophy of language, but also to abstract algebra and the theory of Combinatory Logic. The latter was discovered by Moses Schönfinkel, who was the first to successfully modularize the rules of logic in his groundbreaking work "Über die Bausteine der mathematischen Logik" in 1924 [174]. Abstract algebra [71] is the mathematical way of assigning meaning to sentences by interpreting them in other languages. The main contribution of this text, which is humble in light of the aforementioned precursors, is to recognize the connection of software synthesis with abstract algebra and Combinatory Logic. This provides a synthesis framework based on types and algebraic sorts, which is agnostic to the details and limitations of individual programming languages. Further reasons, contributions, prior publications, and the overall organization of the text are outlined in the rest of this introduction.

1.1 Related Work on Software Synthesis and the Need for Language Agnosticism

The most general definition of software synthesis is the process of translating an abstract program specification to an executable program. This definition is broad enough to have spawned multiple regular conferences, workshops, and conference tracks including ASE [88], LOPSTR [124], SYNTH [182], IWLS [34], a special track at POPL [6], and many more. Gulwani et al. [90] provide one of the most recent overview articles on the subject. The article is in part based on a broad range of input, collected at a Dagstuhl Seminar organized by its authors [31]. In their categorization, the approach described here is a mixture of enumerative search and constraint solving. To better understand the field and its challenges, a short historical overview focused on the more immediate influences on this text will be provided next.

Thomas [185] describes a 1957 talk given by Church as the first formal treatment of the problem. Church posed the question, if functions operating on bit strings can be synthesized, such that inputs and outputs satisfy a given logic formula. While undecidable for arbitrary logics, the problem was later solved for monadic second order logic [36]. As it is the case for many contributions by Church, the synthesis problem quickly gave rise to a vast amount of research. By the late 1970s Manna and Waldinger [130] already acknowledge that the literature on the subject is too vast to summarize in a single paper. Almost 40 years of continuous research later, it is too vast to be even comprehended by a single person. Therefore, a few landmark approaches and general directions will be discussed here, without any claim for completeness. One of them was by said authors, who were among the first to treat program synthesis as a theorem proving problem [131]. This idea is influential until today and also a core component of the approach taken in this text, which uses an algorithm for theorem proving in Combinatory Logic as its basis. A shortcoming of their specific implementation of this idea becomes evident in their later exemplification of synthesizing an unification algorithm [132] and Traugott's subsequent work on synthesizing sort functions [190]: the logic statements used for specification of the synthesis goal are similar to Hoare-Logic [102] and therefore hinder abstract component-oriented modeling, which is crucial for any large scale application. A Dagstuhl workshop organized by Rehof and Vardi [166] later identified component orientation as one of the main challenges in the field. The problem of choosing an adequate specification language was also addressed by Pnueli and Rosner [158], who focused on temporal specification aspects rather than mere descriptions of static input/output relations. This has since spawned a sub-field known as reactive synthesis, which has its own standardized competitions [110] and is a driving factor for practical advances in temporal logic, as well as game and automaton theory. This work adopts the taxonomic classification and idealization of complex system properties by semantic types from the area of reactive synthesis. It was first applied to linear temporal logic based synthesis by Steffen et al. [181].

The initial definition is also compatible with any form of executable algebraic specification. Ehrig and Mahr [71] describe the origins of algebraic specification as a reaction to the software

specification (or lack thereof) crisis in the 1970s. Going back to Wittgenstein's idea of composing all sentences out of operations on sentences, algebraic software specifications describe software by imposing rules on the application of operations. These rules are standardized in form of signatures. In contrast to Hoare-Logic, which constraints memory states before and after running imperative programs, signatures constrain the meaning of operations by their input and output sorts (types), and add additional relations, which are most often equations. This avoids the need to find agreeing memory models and memory location names. It thereby improves compositionality. It is also more flexible in the range of possible specifications, because it is not tied to the imperative programming model. Examples will be illustrated in this text, where algebraic specifications are used to synthesize automata, applicative terms in Combinatory Logic, and even music. Here, a combination of many-sorted signatures [70], subsorting [83], and signature indexing, which is a generalization of parameterization [148], is used. There are notable and practically successful approaches to synthesize programs solely from algebraic specifications. These include the suite of software developed by the Kestrel institute [87] and its SpecWare component [178], the Maude System [127], the Alloy specification language [108] and in a broader sense also the Expander theorem prover [149]. The approaches have in common, that they are interested in producing one model to satisfy the specification or a counter model to show contradictions in the specification. To this end, the specification has to be very precise and needs to exist in the respective formalism. In contrast, the approach taken in this text assumes that most software components exist without precise specifications. Here, synthesis is meant to be a tool of automation which reduces the cost of creating software. Formal specifications require expert users and time, which outweighs the cost benefits of synthesis in many projects. This is especially the case, if large software system parts would have to be re-implemented in another language and formal correctness guarantees are not a core project requirement. The idea is instead to work with possibly under specified systems with exchangeable target languages and multiple possible models. Similar to oracles in complexity theory [152], it should be easier to test and select a solution than to correctly construct it.

The need for language agnosticism does not just arise from the lack of experts on one language and the increasing complexities faced when trying to make it generic enough to solve all problems of one given software project (getting lost in the limitations of the language). Recent empirical evidence shows that even within single software projects multiple languages are present [188; 133; 134]. When synthesis is understood as the automation of composing preexisting software artifacts, it is thereby unrealistic to assume they all exist in one language – much more so if the language has been created to cater the needs and limitations of the given synthesis approach instead of the problem at hand. In the realm of automated software composition, the importance of language agnosticism is not widely acknowledged yet. The FeatureHouse framework by Apel et al. [5] is a notable exception, but it only automates composition after artifacts have been manually selected and brought in order, while synthesis would be responsible for the former task. It remains to discuss some more recent synthesis approaches in light of the language-agnostic requirement.

Haack et al. [92] enumerate instantiates of ML-Functors, which are parameterized signatures. This idea is very similar to the proposal in this text, but their algorithms are not formally specified and the authors conjecture that they are incomplete. Their implementation is limited to the ML programming language.

Feng et al. [74] generate calls to abstract programming interfaces (APIs) by translation to Petri Nets in which places are types and transitions are API calls. Synthesis is then reduced to solving a place reachability problem. This is language independent in principle, but the complications of Petri Nets really only make sense for imperative languages, where variable sharing is exposed to the synthesis algorithm. For other scenarios the authors discuss a simpler hyper-graph based representation, but fail to address why the standard technique of hash-consing [75] is insufficient to create variable sharing. They also allude to a possible hyper-multigraph extension. This extension turns out to be equal [18] to the Tree Grammars, which are generated in the approach developed here. While the authors of [74] conjecture that enumeration of these hyper-multigraphs might be difficult, and their construction is easy, the opposite is the case: the most complex part of the algorithm is creating the Tree Grammars. This misprediction is due to the fact that [74] uses a much weaker type system, which only supports nominal subtyping instead of intersection types with support for nominal and structural subtyping. Without richer type specifications, their approach has to rely on user supplied tests. Many of these can be avoided by the semantic type concept used and discussed here.

A number of more direct type based synthesis approaches exist. These answer the type inhabitation question, that is: relative to a fixed type system, given a context of type assumptions Γ and a target type τ does there exists a term M, with type τ, i.e. $\Gamma \vdash M : \tau$? For the simply typed Lambda Calculus, Statman [179] has shown that this problem is PSPACE-complete. The first algorithm for actually finding inhabitants instead of just proving their existence is attributed to Ben-Yelles [12; 99]. Given the Curry-Howard isomorphism between proofs and types [176], the observation that type inhabitation can be used to construct programs is in line with the earlier results by Manna and Waldinger [131]. Multiple variations of the algorithm by Ben-Yelles [12] exist with slightly different goals, e.g. finding principal (think "best fitting for its type") inhabitants [35; 63], or proofs for equivalent Sequent Calculi [69]. The latter was among the first put into practice with the Haskell tool Djinn [7; 135]. Later, Gvero et al. [91] implemented InSynth, a similar tool for Scala, which is based on a stratified version of simply typed Lambda Calculus to avoid redundant results. One of the authors, Piskac [157], proposes and proves an extension to Hindley-Milner polymorphism [139], but the claim to completeness is not clear, since a later study on the undecidability of this problem by Benke et al. [13] exists. Both [13] and work on type inhabitation with Combinatory Logic, intersection types, and type schematism [57] suggest finite restrictions on the space of possible substitutions in order to obtain algorithms instead of semi-algorithms. This approach is also taken here. Osera [147] does the same, where additionally synchronization, a form of Sequent Calculus [176], and refinement by examples are combined to generate lambda expressions. The core system without polymorphism is also described in [146]. It was later enhanced to represent

refinements with intersection types [78]. The key idea is to encode multiple user-provided input/output examples into types, which is possible since intersection types can represent arbitrary finite function tables [163]. It thereby bridges a long standing gap to the field of example-directed synthesis. Example-directed synthesis has applications especially when used by experts in some other domain than computer science. Leading work in this area has been done by Gulwani, and is even integrated into wide-spread commercially used tools such as Excel, with [73; 151] as a latest prominent accounts of development in this area. There also exist example-based techniques relying on Tree-Automata, the machine equivalent of the Tree Grammars used here, with an application to code-completion [200]. Comparing [78] to other recent work on type directed synthesis with refinement types by Polikarpova et al. [160], examples put less burden on the user, by not requiring formal specifications. In contrast, traditional refinement types are very much in spirit of the initial work by Manna and Waldinger [131]. Demanding more help from the user, these specifications can provide additional guidance to the synthesis algorithm improving its performance, and adding safety as well as resource consumption guarantees to the generated programs [159; 116]. Lambda Calculus with intersection types yields undecidable type inhabitation [193] and typability problems [82]. However, a recent development by Dudenhefner and Rehof has characterized a large fragment of the type system, with decidable typability [64], inhabitation [65], and useful notions of principality [66]. This might, in future, close the gap in expressiveness between refinement with intersection type based examples used by Frankle et al. [78] and the program logic oriented refinement types used by Polikarpova et al. [160]. Here, a different approach will be taken. As explained earlier in reference to [181], instead of examples or full logical specifications, intersection types are used to encode semantic concepts ordered by a taxonomy. Formal specifications remain optional and can be added per use-case, exploiting the vast existing knowledge [71] on how to do this with algebraic specifications. Type-based approaches are also used in the field of automation of interactive theorem-provers. Here, they are known as Hammers and a dedicated workshop exists for their study [113]. Prominent examples of Hammers are Sledegehammer for Isabelle/HOL [32], HOLyHammer for HOL Light [112], and CoqHammer for Coq [44]. Theorem provers are usually complex target languages with undecidable inhabitation problems and almost no further generalization across theorem provers is possible. Hence, Hammers are custom designed tightly integrated tools with little common ground, except that they usually try to encode fragments of their target languages into a logic solvable by automated theorem provers. In principle, the approach presented here can be used for the specific purpose of synthesizing algebraically generated expressions in Coq, because it is implemented and formalized in an executable fragment of Coq without any axioms. Results from this may however be limited, because the execution of code in Coq is very slow. At the time of writing, some preliminary effort has been undertaken to build the infrastructure necessary to use the findings of this text for AST-fragment based synthesis of Coq code [205] within a faster Scala implementation.

While this text tries to overcome the language specific limitations of type based synthesis, other language-agnostic synthesis methods exist. Syntax-guided synthesis (SyGus) [2] is an umbrella

term for a joint effort to synthesize programs from their syntactic structure. Many approaches to SyGus are implemented using Satisfyability Modulo Theory (SMT) solvers or variants thereof [167]. Just relying on the grammar of the synthesis target language results in a vast search space (all possible programs) and additional invariants, again in the form of language-specific equations, have to be taken into account to make it useful. Logics resulting from this are of very high complexity (mostly PSPACE-hard or above). The aforementioned Vienna Summer of Logic [119] spawned a dedicated set of competitions for SyGus implementations [3]. Recent work by Reynolds et al. [168] reconciles SyGus with many-sorted algebraic signatures and integrates counter-examples to speed up the search process. Signatures used in [168] neither support parametric nor subtype polymorphism, which makes them strictly weaker than the approach discussed here. A problem with SyGus is that program syntax is usually not component-oriented. Inala et al. [105] try to overcome this for a specific domain, abstract data type transformation, with a template based approach. They tie their approach to a specific template language. Other than not being language-agnostic, this is very much in the spirit of using combinators and abstract algebra, where the result of combinator/operation application can be any complex syntax tree manipulation, including template instantiation.

Finally, there are some synthesis approaches, the spirit of which can be described as using meta techniques. Some of these are discussed next.

Based on insights into the monadic nature of backtracking by Kiselyov et al. [115], the miniKaren language by Byrd and Friedman [37] tries to solve the most general synthesis problem possible, by enumerating terms relative to an arbitrary given relation. There are a homepage [39], book [79], and a dedicated workshop [38], which illustrates the traction of synthesis toward more fundamental solutions. The miniKaren language is a spiritual successor to the logic programming language Prolog [106], and could as such have serverd as an implementation vehicle for the approach discussed here. In fact, early prototype versions have been based on Prolog [56], but it quickly became apparent, that a more targeted programming effort was needed to improve performance, maintainability, and usability.

The application of machine learning to synthesis tasks has also gained some prominence [120; 72; 89; 175]. Work by Liang et al. [126] uses Combinatory Logic to generate programs. However, they restrict combinators to a fixed base, which severely limits practicality. One of the main challenges in that area are code-base updates without expensive retraining. Parisotto et al. [153] try to achieve this with a domain specific language, but then have to resort to extensive input/output examples to convey synthesis targets. The problem can be traced to the topic of the Dagstuhl seminar [166] on component oriented synthesis: machine learning approaches usually do not compose, when new or updated program components become available.

1.2 Combinatory Logic Synthesis, Contributions, and Organization

First described by Schönfinkel [174], Combinatory Logic is a formal component-oriented approach to logic and programming languages, which predates Turing Machines and Lambda Calculus [191]. An excellent historical account of its almost 100 years of development is given by Cardone and Hindley [40]. Here, two fundamental extensions to what is usually known as Combinatory Logic with simple types [176; 100] are used. The first extension are Barendregt-Coppo-Dezani (BCD) intersection types. Originally, intersection types have been added to Lambda Calculus [8] to obtain a system characterizing exactly the set of normalizing lambda terms [82]. They later have been used to type Combinatory Logic [49]. The second extension generalizes the combinator base form the two Turing-complete standard combinators S and K to an arbitrary set of combinators, which are given as input. This results in Hilbert-Style systems of logic relative to the combinators as axioms [176]. The resulting system of Finite Combinatory Logic with Intersection Types (FCL) has been studied by Rehof and Urzyczyn [165]. It provides inherent modularity with combinators, an expressive polymorphic type language with intersection types, and it is well-suited for type based synthesis, because type inhabitation is decidable. The system can be extended with bounded schematic polymorphism [57] and the (CL)S Framework is developed around it [56; 55; 20].

This text describes the author's contributions to the understanding of Combinatory Logic Synthesis and their practical application in form of the (CL)S Framework. In short they can be outlined as follows:

1. A rigorous mechanized formalization of the theory of Combinatory Logic Synthesis is provided in Chapter 2. The formalization [15] is purely constructive and valid in the Calculus of inductive Constructions [186] without additional axioms. It is mechanically checked by the Coq proof assistant [184] and formally verified algorithms can be extracted to Haskell and OCaml. To the best knowledge of the author, this makes (CL)S the only approach to synthesis with a sound and complete fully theorem prover verified theory.

    1.1. An extension of intersection types with constructors and products is presented in Section 2.2 and its effects on the BCD subtype relation are studied in Section 2.3.

    1.2. The currently best known verified algorithm for deciding the BCD type relation is developed in Section 2.3.

    1.3. The first mechanized proof for decidability of type-checking in FCL is given in Section 2.4.

    1.4. A proper theory of combining domains of discourse with semantic types is developed in Section 2.4.

    1.5. The first fully formalized algorithm for answering the enumerative type inhabitation problem of FCL is developed in Section 2.5.

1.6. The question of Rehof and Urzyczyn [165] about the possibility to use Tree Automata for FCL including subtyping without double exponential blow-up is positively answered in Section 2.5.

1.7. Indexed many-sorted algebraic signature families with subsorting are defined in Section 2.6.1 and their general sort emptiness problem is proven to be undecidable in Section 2.6.2.

1.8. All finite restrictions of the sort emptiness problem of indexed many-sorted algebraic signature families with subsorting are proven to be decidable via reduction to the enumerative type inhabitation algorithm for FCL in Section 2.6.3.

1.9. An algebra independent, sound, complete, and unique translation of synthesized applicative terms of Combinatory Logic to any target language is constructed in Section 2.6.3.

2. A practical Scala implementation of the resulting synthesis framework is developed in Chapter 3.

   2.1. Scala is extended to allow the seamless specification of combinators in Section 3.1 and Section 3.2.

   2.2. Support mechanisms to integrate metaprogramming into the synthesis framework are developed in Section 3.3.

   2.3. Software tests for the resulting framework are developed in Section 3.4.

3. The impact of the framework on other people's work is evaluated in Chapter 4. This includes a novel technique for the model driven development of Software Productlines.

4. A critical discussion of current limitations and salient points of future work is provided in Chapter 5.

The rest of this chapter describes the relation of the author's prior academic publications to this text.

1.3 Publications

First ideas of this text were presented in [20], where the author outlines his ideas for extensions of an older F# based implementation of (CL)S in sections 4.2 and 5 of that text. The application is synthesis of configurations for dependency injection frameworks, which is also investigated in the master's thesis of the author [14]. Ideas for language independent metaprogramming are sketched out in [20], but still focus on Staged Composition Synthesis with modal intersection types [59]. Modal intersection types were later dropped from the framework. Work with Heineman et al. [96] showed that metaprogramming is a useful component of synthesis. However, the binding analysis to ensure all template components are closed (a requirement to use modal box types) turned out to be infeasible and impractical. Strictly speaking, none of the published examples in [20; 96] meet this requirement, because all target language code uses libraries and thereby has unbound names. A target language specific analysis of compositional safety properties with the meta language would have been necessary to ensure that modal-types give the guarantees they are used for. Additionally, the target language needs a compatible type system and compiler with programmatically accessible binding analysis features to check if box-types are used used correctly. In practice, even the availability of usable parsers and syntax trees is a problem for most target languages [5]. All attempts quickly turned out to be an instance of "getting lost in the details of language", which this text motivates to avoid. In an older formalization [17] of the work presented here, the Modal Calculus used by Davies and Pfenning [47] is shown to be representable by the new algebraic approach. Subsequently, the main framework developers including the author made the joint decision to drop support for modal types. Another less formal observation resulted from many discussions with other researchers and feedback from rejected papers. It turned out that explaining the concept of modal types raised the mental burden of conveying the ideas of the framework.

In a separate line of work, applications of Combinatory Logic Synthesis to the generation of object-oriented code are investigated [19; 22; 25]. Specifically, mixin application chains are synthesized, where a Mixin is a function mapping classes to classes. A Lambda Calculus with records and intersection types is developed to represent objects, classes, and mixins. Results regarding synthesis are positive. However, the specification mechanism is difficult to get right and very fine-tuned, favoring the needs of synthesis and type-theory rather than being easy to understand. These insights add to the motivation to have a simpler, language independent approach. The usefulness of records motivated adding distributing covariant type constructors to the framework.

Applications to product lines have been studied. In [23] a product line of robot control programs is synthesized. Schäfer [170] subsequently improved the results. They no longer require modal types, work with new versions of (CL)S, and are generalized to support multiple implementation strategies, including arrows. In [21] valid configurations of feature diagrams are synthesized and then translated to product line code.

Feature diagrams and their encoding into Combinatory Logic are formalized in Coq. Success with this formalization motivated the other applications of theorem proving in this text. A first Coq verified algorithm to decide the BCD subtype relation is analyzed in [23]. It is based on the idea of prime ideals, which is also discussed in Section 2.3. The algorithm has been improved since then, leading to the form presented in [28]. Section 2.2 and Section 2.3 are largely based on the latter publication.

A snapshot of the framework at that point in time was presented at PEPM 2018 [26]. The formalization has since then been improved to use indexed signature families instead of open and closed sorts. While even more expressive, it simplifies encodings of sorts, and provides completeness and uniqueness instead of completeness and uniqueness up to choice of substitutions. Internally, the formalization has also been completely rewritten to allow a clearer presentation and code that can be extracted from Coq to Haskell and OCaml.

The latest addition to the (CL)S framework is a debugger, jointly developed with Anna Vasileva and discussed in detail in [18].

Chapter 2

Theory

This chapter describes the theory of a language-agnostic synthesis framework. All proofs are formalized in Coq and available online [15]. Proofs in the text try to give a high-level intuition about the necessary low-level steps. Table A.1 maps all statements to the formalization, allowing to trace every step in every proof. The chapter is organized to first give some details on the formalization in Section 2.1. This is followed by the basic definitions required for intersection types in Section 2.2. Then, in Section 2.3, the Barendregt-Coppo-Dezani [8] subtype relation and a decision procedure for it are studied in depth. Finite Combinatory Logic with Intersection Types (FCL) is defined and studied in Section 2.4. An enumerative type inhabitation algorithm for FCL is developed in Section 2.5. A new result presented in Section 2.6 links this algorithm to language-agnostic synthesis.

2.1 Formal Verification Setup

All algorithms in this text are specified using a number of standard functions and data-structures that are available in almost all functional programming languages and have well-studied algorithmic [162; 145] and category theoretic [136] properties. They are implemented in the Mathematical Components library [129] for Coq, which was used in version 1.8.0 together with Coq version 8.8.2. This library also includes hundreds of proven lemmas about these structures, which are silently assumed in this text, but explicitly mentioned in the accompanying machine checked formalization. The text requires basic knowledge of standard notions from Set and Category theory. Readers unfamiliar with them may find a precise modern introduction in [52]. The relationship between sets and types in Coq is clarified in [10]. Here, the intuitive understanding is used in which types act as sets. The impredicative sort (type variables can be instantiated by expressions of the same sort) Prop in Coq represents predicates. In the machine-checked formalization, no Axioms are added to the type theory of Coq, which is explained in [186]. This means all proofs can be done in an intuitionistic (no excluded middle), purely constructive (no axiom of choice), proof relevant (two proofs of the same proposition are not necessarily the same), and intensional (functions mapping the same inputs to the same outputs are not necessarily the same) way. The presentation in the text avoids some of the detours imposed by this weaker logic and mentions differences when they become relevant. In practice, the purist approach to logic pays off by allowing extraction to Haskell and OCaml. Ramifications from not being able to use the principle of the excluded middle (P or not P is always true) are mild, because the Small Scale Reflection extensions for Coq [85] can be used to perform a double negation translation [193] of any intuitionistic predicate P with little syntactic overhead whenever a decision procedure for P is available. Proof irrelevance can also be restored whenever P is decidable by any function P_{dec}: types with decidable equality can be truncated (all their equality proofs are equal) [94] and thus P can be replaced by $P_{\text{dec}} = \text{true}$. The technique of handcrafted small inversions by Monin and Shi [141] justifies performing case-analysis on the complex dependently typed operational semantics step relations used in this text, even without (equality-)proof irrelevance or the equivalent Axiom K, which is otherwise required for dependent pattern matching [41].

The rest of this section will introduce some standard [129; 52; 150; 136; 53; 162; 145] structures and notations used throughout this text.

Definition 1 (Basic Structures)

- The category **Set** has sets as objects, functions as morphisms. For any set A, the identity morphism is the identity function $\text{id}_A : A \to A$ with $id_A(x) = x$. Composition is function composition $(f \circ g)(x) = f(g(x))$.
- The singleton unit set is given by $\mathbb{1} = \{()\}$.
- The set of booleans is given by $\mathbb{B} = \{\text{true}, \text{false}\}$.

- The set of natural numbers is given by $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$.
- An I-indexed family of objects/morphisms A of some category $\mathscr{C}$ is a map from I into $\mathscr{C}$, i.e. for any $i \in I$, A_i is a object/morphism.
  - Families of sets are used to define indexed data types. A simple example is family $LT = \{k \in \mathbb{N} \mid k < n\}_{n \in \mathbb{N}}$, which has LT_{10}, the set of natural numbers less than 10, as its member.
  - Families of morphisms model generic functions, e.g. $\mathrm{up} : (LT_n \to LT_{n+1})_{n \in \mathbb{N}}$ can be defined by $\mathrm{up}_n(k) = k + 1$.
  - The category of I-indexed families over **Set** is denoted by $\mathbf{Set}^I$. It has I-indexed families of sets as objects and I-indexed families of functions as morphisms. The identity morphism is the family of identity functions $\mathrm{id}_A : (A_i \to A_i)_{i \in I}$ with $\mathrm{id}_A(i, x) = x$. Composition is function composition at a given index, $(f \circ g)_s = f_s \circ g_s$.
- Products $\prod_{x \in I} A_i$ over some set I are objects equipped with unique projections $\pi : (\prod_{j \in I} A_j \to A_i)_{i \in I}$.
  - Tuples with $n \in \mathbb{N}$ elements over some family of sets A are written $\prod_{k \in \{1,2,\ldots,n\}} A_n = \prod_{k=1}^{n} A_n = A_1 \times A_2 \times \cdots \times A_n$.
  - The special case of $n = 2$ is the Cartesian product $A_1 \times A_2$ and projections are abbreviated in suffix notation $\pi_1(x, y) = (x, y).1 = x$ and $\pi_2(x, y) = (x, y).2 = y$.
  - Function families $f : (A_i \to B_i)_{i \in I}$ are isomorphic to $f' : \prod_{i \in I} A_i \to B_i$ with $f_i = \pi_i(f')$. Notations are sometimes mixed to avoid clutter in repeated indexes.
- Sums $\bigoplus_{i \in I} A_i$ over some set I are objects equipped with unique injections $\mathrm{inj} : (A_i \to \bigoplus_{j \in I} A_j)_{i \in I}$.
  - Sums $\bigoplus_{i \in I} A_i$ in **Set** are given by $\mathrm{inj}_i(x) = (i, x)$ and encode tuples where the second component depends on the first.
  - The notation $\Sigma_{i \in I} A_i$ is more common in literature, but avoided to resolve a name clash with signatures, which also use capital letter Σ.
- Lists A^* over any set A are given by the grammar:

  $$A^* \ni \Delta ::= [::] \mid [::\, x \& \Delta]$$

  where $x \in A$.
  - Singleton lists are abbreviated using $[::\, x \& [::]] = [::x]$.
  - Element x is in $[::\, y \& \Delta]$ iff $x = y$ or x is in Δ.
  - The subsequence relation $\sqsubseteq \subseteq A^* \times A^*$ is the least relation closed under the rules

  $$\frac{}{[::] \sqsubseteq \Delta} \qquad \frac{\Delta_1 \sqsubseteq \Delta_2}{\Delta_1 \sqsubseteq [::\, x \& \Delta_2]} \qquad \frac{\Delta_1 \sqsubseteq \Delta_2}{[::\, x \& \Delta_1] \sqsubseteq [::\, x \& \Delta_2]}$$

- Relation suffix $\subseteq A^* \times A^*$ is the least relation closed under the rules

$$\frac{\text{suffix}(\Delta_1, \Delta_2)}{\text{suffix}(\Delta_1, [::\, x\&\Delta_2])} \qquad \frac{}{\text{suffix}(\Delta, \Delta)}$$

- Lambda functions $\lambda x.M_x$ behave like normal functions $f(x) = M_x$, but do not require an explicit name. □

Lists come with multiple pre-defined functions.

Definition 2 (Functions on Lists) If A and B are sets, and $\Delta, \Delta_1, \Delta_2 \in A^*$, $\tilde{\Delta} \in B^*$, $z \in A$, $s \in B$, $n \in \mathbb{N}$, $f : A \to B$, $p : A \to \mathbb{B}$, and $g : A \to \bigoplus_{i \in \{0,1\}} \{(i, X) \mid X = B \text{ or } i = 0 \text{ and } X = \mathbb{1}\}_i$ then function

- $\text{behead}(\Delta) = \begin{cases} \Delta & \text{for } \Delta = [::\, x\&\Delta'] \\ [::] & \text{otherwise} \end{cases}$ removes the first element of a list.

- $\text{ohead}(\Delta) = \begin{cases} (1, x) & \text{for } \Delta = [::\, x\&\Delta'] \\ (0, ()) & \text{otherwise} \end{cases}$ returns the first element of a list (if it exists).

- $\text{foldl}(f, s, \Delta) = \begin{cases} \text{foldl}(f, f(s, x), \Delta') & \text{for } \Delta = [::\, x\&\Delta'] \\ s & \text{otherwise} \end{cases}$ folds a list to the left.

- $\text{foldr}(f, s, \Delta) = \begin{cases} f(x, \text{foldr}(f, s, \Delta)) & \text{for } \Delta = [::\, x\&\Delta'] \\ s & \text{otherwise} \end{cases}$ folds a list to the right.

- $\text{map}(f, \Delta) = \begin{cases} [::\, f(x)\&\text{map}(f, \Delta')] & \text{for } \Delta = [::\, x\&\Delta'] \\ [::] & \text{otherwise} \end{cases}$
  applies function f to all the elements in the list.

- $\text{filter}(p, \Delta) = \begin{cases} [::\, x\&\text{filter}(p, \Delta')] & \text{for } \Delta = [::\, x\&\Delta'] \text{ and } p(x) = \text{true} \\ \text{filter}(p, \Delta') & \text{for } \Delta = [::\, x\&\Delta'] \text{ and } p(x) = \text{false} \\ [::] & \text{otherwise} \end{cases}$
  removes all elements x of a list where test $p(x)$ fails

- $\text{pmap}(g, \Delta) = \begin{cases} \text{pmap}(g, \Delta') & \text{for } \Delta = [::\, x\&\Delta'] \text{ and } g(x) = (0, ()) \\ [::\, y\&\text{pmap}(p, \Delta')] & \text{for } \Delta_1 = [::\, x\&\Delta'] \text{ and } g(x) = (1, y) \\ [::] & \text{otherwise} \end{cases}$
  collects all successful applications of g on elements of a list.

- $\Delta_1 +\!\!+ \Delta_2 = \begin{cases} [::\, x\&\Delta_1' +\!\!+ \Delta_2] & \text{for } \Delta_1 = [::\, x\&\Delta_1'] \\ \Delta_2 & \text{otherwise} \end{cases}$ concatenates two lists

- $\text{lsize}(\Delta) = \begin{cases} 1 + \text{lsize}(\Delta') & \text{for } \Delta = [::\, x \& \Delta'] \\ 0 & \text{otherwise} \end{cases}$ computes the length of a list.

- $\text{rcons}(\Delta, z) = \begin{cases} [::\, x \& \text{rcons}(\Delta', z)] & \text{for } \Delta = [::\, x \& \Delta'] \\ [::z] & \text{otherwise} \end{cases}$
  adds an element to the end of a list.

- $\text{last}(z, \Delta) = \begin{cases} x & \text{for } \Delta = [::x] \\ \text{last}(z, [::\, y \& \Delta']) & \text{for } \Delta = [::\, x \& [::\, y \& \Delta']] \\ z & \text{otherwise} \end{cases}$
  returns the last element of a list or its first argument if the list is empty.

- $\text{rev}(\Delta) = \text{catrev}(\Delta, [::])$

  $\text{where catrev}(\Delta, \Delta_2) = \begin{cases} \text{catrev}(\Delta', [::\, x \& \Delta_2]) & \text{for } \Delta = [::\, x \& \Delta'] \\ \Delta_2 & \text{otherwise} \end{cases}$
  reverses a list.

- $\text{zip}(\Delta, \tilde{\Delta}) = \begin{cases} [::\, (x, y) \& \text{zip}(\Delta', \tilde{\Delta}')] & \text{for } \Delta = [::\, x \& \Delta'] \text{ and } \tilde{\Delta} = [::\, y \& \tilde{\Delta}'] \\ [::] & \text{otherwise} \end{cases}$
  combines two lists.

- $\text{nth}(z, \Delta, n) = \begin{cases} x & \text{for } n = 0 \text{ and } \Delta = [::\, x \& \Delta'] \\ \text{nth}(z, \Delta', n') & \text{for } n = n' + 1 \text{ and } \Delta = [::\, x \& \Delta'] \\ z & \text{otherwise} \end{cases}$
  returns the n-th element of a list or the default element z if the list is too short.

- $\text{take}(n, \Delta) = \begin{cases} [::\, x \& \text{take}(n', \Delta')] & \text{for } n = n' + 1 \text{ and } \Delta = [::\, x \& \Delta'] \\ [::] & \text{otherwise} \end{cases}$
  extracts up to n elements from the start of a list.

- $\text{subseqs}(\Delta) = \begin{cases} \text{map}(\lambda \Delta''.[::\, x \& \Delta''], \text{subseqs}(\Delta')) +\!\!+ \Delta' & \text{for } \Delta = [::\, x \& \Delta'] \\ [::[::]] & \text{otherwise} \end{cases}$
  computes all subsequences $\Delta' \sqsubseteq \Delta$ of a list Δ.

- $\text{nseq}(n, z) = \begin{cases} [::\, z \& \text{nseq}(n', z)] & \text{for } n = n' + 1 \\ [::] & \text{otherwise} \end{cases}$ returns a list of n repetitions of z.

- $\text{enum}(A) = [::\, x_1 \& [::\, x_2 \& \ldots [::x_n] \ldots]]$
  enumerates set A as a list if A is the finite set $[::\, x_1 \& [::\, x_2 \& \ldots [::x_n] \ldots]]$

- $\text{undup}(\Delta) = \begin{cases} \text{undup}(\Delta') & \text{for } \Delta = [::\, x \& \Delta'] \text{ and } x \text{ is in } \Delta' \\ [::\, x \& \text{undup}(\Delta')] & \text{for } \Delta = [::\, x \& \Delta'] \text{ and not } x \text{ is in } \Delta' \\ [::] & \text{otherwise} \end{cases}$
  removes all duplicates in a list. □

2.2 Intersection Types with Products and Type Constructors

Prior to any formalization, syntactic conventions for intersection types have to be established.

Definition 3 (Intersection Types with Products and Type Constructors) Intersection types are formed over the following syntax:

$$\mathbb{T} \ni A, B ::= \omega \mid c(A) \mid (A \star B) \mid (A \to B) \mid (A \cap B)$$

where $c \in \mathbf{C}$ is a constructor drawn from a countable set $\mathbf{C}$, which will be chosen per use-case.

Throughout this text, capital Latin letters $A, B, C, \ldots$ are used to denote intersection types, while small Latin letter denote constructors. Lists of types are denoted by Δ with some index or apostrophe for disambiguation. Products $\star$, arrows $\to$ and intersections $\cap$ are to be read just like their set-theoretic counterparts $\times$, $\to$ and $\cap$. Parentheses and unnecessary syntax are reduced using the following conventions:

- Intersections bind stronger than products, which bind stronger than arrows:

$$A \cap B \star C \to D \times E \cap F = (((A \cap B) \star C) \to (D \times (E \cap F)))$$

- Intersections and arrows associate to the right:

$$A \cap B \cap C \to D \to E = ((A \cap (B \cap C)) \to (D \to E))$$

- Products associate to the left:

$$A \star B \star C = ((A \star B) \star C)$$

- Constructors with ω-arguments are abbreviated:

$$c = c(\omega)$$

Equality $A = B$ is strict syntactic equality following the conventions above. □

The syntax presented in Definition 3 is identical to the syntax in [28]. Type constructors are formally studied in their most generic form with arbitrary arities in [125]. Here, the system is made more uniform by restricting them to the universal type of anything ω, constructor symbols applied to one argument, arrows, intersections and a single binary product constructor. This restriction retains the possibility to express interesting types. Notation conventions highlight that type constants, e.g. `int` or `String`, can be represented by having a single ω argument: $\texttt{int} = \texttt{int}(\omega)$ and $\texttt{String} = \texttt{String}(\omega)$. Parameterized types such as $\texttt{List}(\texttt{String})$ are directly supported. Products can encode multiple parameters, e.g. $\texttt{Graph}(N \star E)$ for a

graph with nodes of some type N and edges of some type E. Unlike other presentations [8; 194; 57] there are no type variables. Usually, type variables either model an arbitrary fixed set of type constants or they are used for schematic polymorphism. The former use-case is handled by constructor symbols as explained above and no point in the formalization requires the latter use-case.

The following functions allow some basic analysis and manipulation of the structure of types.

Definition 4 (Functions to analyze and manipulate types) Lists of types can be converted to an intersection associated to the right:

$$\text{intersect}(\Delta) = \begin{cases} \omega & \text{for } \Delta = [::] \\ A & \text{for } \Delta = [::A] \\ A_1 \cap \text{intersect}[::A_2 \& \Delta'] & \text{for } \Delta = [::A_1 \& [::A_2 \& \Delta']] \end{cases}$$

For any list Δ (not necessarily of types), the shorthand

$$\bigcap_{A_i \in \Delta} M = \text{intersect}(\text{map}(\lambda A_i.M, \Delta))$$

is used to compute its intersection and (if necessary) convert its contents using expression M. The arity of a type is the type of the arguments of its outermost type constructor:

$$\text{arity}(A) = \begin{cases} \mathbb{1} & \text{for } A = \omega \\ \mathbb{T} & \text{for } A = c(B) \\ \mathbb{T} \times \mathbb{T} & \text{otherwise} \end{cases}$$

Types can be measured counting the elements in their syntax tree, the maximal length of an arrow they include, or the maximal depth of their syntax tree (ignoring intersections):

$$\text{size}(A) = \begin{cases} 1 & \text{for } A = \omega \\ 1 + \text{size}(B) & \text{for } A = c(B) \\ 1 + \text{size}(B) + \text{size}(C) & \text{for } A = B \cap C \text{ or } A = B \to C \text{ or } A = B \star C \end{cases}$$

$$\text{length}(A) = \begin{cases} 1 + \text{length}(C) & \text{for } A = B \to C \\ \max\{\text{depth}(B), \text{depth}(C)\} & \text{for } A = B \cap C \\ 1 & \text{otherwise} \end{cases}$$

$$\text{depth}(A) = \begin{cases} 1 & \text{for } A = \omega \\ 1 + \text{depth}(B) & \text{for } A = c(B) \\ 1 + \max\{\text{depth}(B), \text{depth}(C)\} & \text{for } A = B \to C \text{ or } A = B \star C \\ \max\{\text{depth}(B), \text{depth}(C)\} & \text{for } A = B \cap C \end{cases}$$

□

2.3 Subtyping

The intuition of the intersection type operator $\cap$ is revealed when types are preordered by the relation $\leq$ introduced by Barendregt, Coppo, and Dezani-Ciancaglini [8]. In "$A \leq B$" type A is a subtype of B, meaning that its values can take the shape of values of type B and used whenever values of type B are expected. This is also referred to as subtype polymorphism. Relation $\leq$ is called BCD Subtyping, abbreviating last name initials in honor of its inventors. Many programming languages are equipped with some similar notion. See [9] for a discussion of how BCD subtyping can provide a basis for the formalization of other subtype relations. Also Section 2.6 will come back to this point.

Definition 5 (BCD Subtyping) The subtype relation $A \leq B$ is the least relation closed under the rules:

$$\frac{c \leq_{\mathbf{C}} d \qquad A \leq B}{c(A) \leq d(B)}\ (\text{CAx}) \qquad \frac{}{c(A) \cap c(B) \leq c(A \cap B)}\ (\text{CDist})$$

$$\frac{}{A \leq \omega}\ (\omega) \qquad \frac{}{\omega \leq \omega \to \omega}\ (\to \omega)$$

$$\frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}\ (\text{Sub}) \qquad \frac{}{(A \to B_1) \cap (A \to B_2) \leq A \to B_1 \cap B_2}\ (\text{Dist})$$

$$\frac{A_1 \leq B_1 \qquad A_2 \leq B_2}{A_1 \star A_2 \leq B_1 \star B_2}\ (\text{ProdSub})$$

$$\frac{}{(A_1 \star A_2) \cap (B_1 \star B_2) \leq A_1 \cap B_1 \star A_2 \cap B_2}\ (\text{ProdDist})$$

$$\frac{A \leq B_1 \qquad A \leq B_2}{A \leq B_1 \cap B_2}\ (\text{GLB}) \qquad \frac{}{B_1 \cap B_2 \leq B_1}\ (\text{LUB}_1) \qquad \frac{}{B_1 \cap B_2 \leq B_2}\ (\text{LUB}_2)$$

$$\frac{A \leq B \qquad B \leq C}{A \leq C}\ (\text{Trans}) \qquad \frac{}{A \leq A}\ (\text{Refl})$$

where $\leq_{\mathbf{C}}$ is an arbitrary fixed preorder chosen together with the constructor symbols $\mathbf{C}$. □

Rules (CAx), (CDist), (ProdSub), and (ProdDist) are extensions to the original relation [8] and account for the extended set of intersection types used in this text. The extensions are again a restriction of those studied in [125]. Rule (ω) positions type ω as the universal type of any value. This is analogous to the type `Object` in Java or C#. Rule ($\to \omega$) accounts for the lack of information on functions where the only known property is that they compute ω, i.e. anything. Combined with rule (ω) this information is equally (un-)specific as the information

that such a function could be anything. Rule (SUB) expresses the subtype polymorphism property for functions. A function of type $A_1 \to A_2$ is usable in place of a function of type $B_1 \to B_2$ if it operates on less specific inputs ($B_1 \leq A_1$, contravariance) and produces more specific outputs ($A_2 \leq B_2$, covariance). Note how rule (PRODSUB) is the same for products with the crucial difference that components A_1 and B_1 are also covariant ($A_1 \leq B_1$) in its first premise. Similarly, unary constructors are ordered by (CAX). Here, the first premise is replaced by a customizable relation $\leq_{\mathbf{C}} \subseteq \mathbf{C} \times \mathbf{C}$ that can be chosen with the set of constructor symbols $\mathbf{C}$. This extension mechanism provides the basis for embedding signature subsorting in Section 2.6. Rules (GLB), (LUB_1), and (LUB_2) turn the intersection operator $\cap$ into the meet of a semi-lattice [86]. If values of type A can be used for B_1 and B_2, they can be used whenever the intersection $B_1 \cap B_2$ is expected. Vice versa, values of the intersection of types B_1 and B_2 can be used as values of both of these types. These rules are compatible with interpreting types as sets of values with $\leq$ as subset inclusion $\subseteq$ and $\cap$ as the intersection of sets. Rules (DIST), (CDIST), and (PRODDIST) allow distribution of intersections over covariantly related type components. Distribution over the contravariant left component of arrows $(A_1 \to B_1) \cap (A_2 \to B_2) \leq A_1 \cap A_2 \to B_1 \cap B_2$ is derivable from the other rules and does not need to be explicitly included. Finally (TRANS) and (REFL) close $\leq$ under transitivity and reflexivity, turning it into a preorder. Equivalent formulations of the rules above are discussed in [194].

The decision problem for the BCD Subtype relation is the following: given two types A and B, are they related by $A \leq B$? At first glance, most rules of the subtype relation seem to indicate that a decision procedure can simply try to construct a proof tree by reading the rules from conclusion to premise. However, this approach breaks down because of type B in rule (TRANS), which is not present in the conclusion and would thus have to be guessed. Rules with variables in their premises that do not occur in their conclusion are called called cut-rules and B is called a cut-type. Reformulating a proof system without cut rules is called cut-elimination and the textbook example of cut-elimination is Gentzen's Hauptsatz [176]. Laurent [125] proves that cut elimination is possible for the rules of $\leq$. Solutions to the BCD Subtype decision problem can be traced along a line of work started over 30 years ago with a decidability proof by Pierce [156]. It was subsequently improved to decision procedures with exponential worst-case runtime [45; 46; 118]. Later, algorithms with polynomial $\mathcal{O}(n^4)$ [165] and $\mathcal{O}(n^5)$ [180] runtimes have been obtained. The first procedure with quadratic runtime $\mathcal{O}(n^2)$ was developed in [67]. Algorithms with mechanized proofs but without runtime guarantees have followed [24; 125; 30], where issues with the original proof have been discovered in [30]. The decision procedure used in this text is the first with formally verified quadratic bounds on the number of recursive steps involved to find a solution. At the time of writing a detailed account of its development is peer-reviewed and accepted for publication [28] with authorship of the presented parts of that publication coinciding with the declared authorship of this text.

Besides the aforementioned cut in rule (TRANS), the most difficult part in designing and verifying a decision procedure for $\leq$ is created by the first premise in rule (SUB). It toggles the positions of types to $B_1 \leq A_1$ instead of $A_1 \leq B_1$. This effectively prevents structural recursion on types from working and more advanced techniques have to be employed for ensuring

termination. Larchey-Wendling and Monin [123] suggest a way to obtain a termination certificate Dom that allows recursion decreasing on its structure for such situations. The trick is to turn a proof about a recursively defined function f into a proof about an inductively defined relation R. The relation is then proven to be functional (xRy_1 and xRy_2 implies $y_1 = y_2$). Termination certificate Dom x is defined to be a proof tree for the statement that there exists y such that xRy (x is in the domain of the relation). If Dom is total (for all x there exists a proof Dom x) function f can then be defined by structural recursion on the proof tree Dom x for any input x. Bounding the size of Dom puts a bound on the number of recursive calls in f, yielding information about worst case performance. Inductively defining R is also a good way to understand the algorithm: it effectively specifies its operational semantics by means of rules for each possible input. This is useful for debugging purposes, because the algorithm can be "halted" at each rule invocation. Furthermore, the relation provides an alternative cut-free representation for $\leq$.

Some auxiliary functions are required for the main decision procedure.

Definition 6 (Auxiliary functions to decide BCD Subtyping) Testing for $\omega \leq A$ is purely syntactic:

$$\text{isOmega}(A) = \begin{cases} \text{true} & \text{for } A = \omega \\ \text{isOmega}(A_2) & \text{for } A = A_1 \to A_2 \\ \text{isOmega}(A_1) \text{ and } \text{isOmega}(A_2) & \text{for } A = A_1 \cap A_2 \\ \text{false} & \text{otherwise} \end{cases}$$

Function $\text{cast}_B : \mathbb{T} \to \text{arity}(B)$ collects type components to be recursively compared when checking $A \leq B$. Its helper function cast'_B avoids inefficient functional list concatenations using an accumulator parameter Δ.

$$\text{cast}_B(A) = \begin{cases} [::\omega] & \text{for } B = \omega \\ [::(\omega,\omega)] & \text{for } B = B_1 \to B_2 \text{ and } \text{isOmega}(B_2) \\ \text{cast}'_B(A, [::]) & \text{otherwise} \end{cases}$$

$$\text{cast}'_B(A, \Delta) = \begin{cases} [::\ A' \& \Delta] & \text{for } A = c(A') \text{ and } B = d(B') \text{ and } c \leq_{\mathbf{C}} d \\ [::\ (A_1, A_2) \& \Delta] & \text{for } A = A_1 \to A_2 \text{ and } B = B_1 \to B_2 \\ [::\ (A_1, A_2) \& \Delta] & \text{for } A = A_1 \star A_2 \text{ and } B = B_1 \star B_2 \\ \text{cast}'_B(A_1, \text{cast}'_B(A_2, \Delta)) & \text{for } A = A_1 \cap A_2 \\ \Delta & \text{otherwise} \end{cases}$$

□

The relational specification of subtyping can be thought of as a machine performing steps to transform inputs to outputs.

Definition 7 (BCD Subtype Machine) The BCD Subtype relation is decided by a machine transforming inputs

$$\mathbb{I}_{\text{Sub}} \ni i ::= [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B] \mid [\ \texttt{tgt\_for\_srcs\_gte}\ \ A\ \ \texttt{in}\ \ \Delta]$$

to outputs

$$\mathbb{O}_{\text{Sub}} \ni o ::= [\ \texttt{Return}\ \ b] \mid [\ \texttt{check\_tgt}\ \ \Delta']$$

where b is a Boolean, $A, B \in \mathbb{T}$, Δ is a list of pairs of types, and Δ' is a list of types.
Its operational semantics is specified by the least relation $\rightsquigarrow \subset \mathbb{I}_{\text{Sub}} \times \mathbb{O}_{\text{Sub}}$ closed under the following rules:

$$\frac{}{[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ \omega] \rightsquigarrow [\ \texttt{Return true}]}\ (\text{STEP}_{\omega})$$

$$\frac{[\ \texttt{subty}\ \ \bigcap_{A_i \in \text{cast}_{c(B)}(A)} A_i\ \ \texttt{of}\ \ B] \rightsquigarrow [\ \texttt{Return}\ \ b]}{[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ c(B)] \rightsquigarrow [\ \texttt{Return}\ \ \text{cast}_{c(B)}(A) \neq [::] \text{ and } b]}\ (\text{STEP}_{\text{CTOR}})$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ \ \bigcap_{A_i \in \text{cast}_{B_1 \star B_2}(A)} A_i.1\ \ \texttt{of}\ \ B_1] \rightsquigarrow [\ \texttt{Return}\ \ b_1] \\ [\ \texttt{subty}\ \ \bigcap_{A_i \in \text{cast}_{B_1 \star B_2}(A)} A_i.2\ \ \texttt{of}\ \ B_2] \rightsquigarrow [\ \texttt{Return}\ \ b_2]\end{array}}{[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \times B_2] \rightsquigarrow [\ \texttt{Return}\ \ \text{cast}_{B_1 \star B_2}(A) \neq [::] \text{ and } b_1 \text{ and } b_2]}\ (\text{STEP}_{\star})$$

$$\frac{\begin{array}{c}[\ \texttt{tgt\_for\_srcs\_gte}\ \ B_1\ \ \texttt{in}\ \ \text{cast}_{B_1 \to B_2}(A)] \rightsquigarrow [\ \texttt{check\_tgt}\ \ \Delta] \\ [\ \texttt{subty}\ \ \bigcap_{A_i \in \Delta} A_i\ \ \texttt{of}\ \ B_2] \rightsquigarrow [\ \texttt{Return}\ \ b]\end{array}}{[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \to B_2] \rightsquigarrow [\ \texttt{Return}\ \ \text{isOmega}(B_2) \text{ or } b]}\ (\text{STEP}_{\to})$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ \ B\ \ \texttt{of}\ \ A.1] \rightsquigarrow [\ \texttt{Return}\ \ b] \\ [\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ \Delta] \rightsquigarrow [\ \texttt{check\_tgt}\ \ \Delta']\end{array}}{\begin{array}{c}[\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ [::A\&\Delta]] \rightsquigarrow \\ [\ \texttt{check\_tgt}\ \ \text{if } b \text{ then } [::A.2\&\Delta'] \text{ else } \Delta']\end{array}}\ (\text{STEP}_{\text{CHOOSETGT}})$$

$$\frac{}{[\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ [::]] \rightsquigarrow [\ \texttt{check\_tgt}\ \ [::]]}\ (\text{STEP}_{\text{DONETGT}})$$

$$\frac{\begin{array}{c}[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1] \rightsquigarrow [\ \texttt{Return}\ \ b_1] \\ [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_2] \rightsquigarrow [\ \texttt{Return}\ \ b_2]\end{array}}{[\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \cap B_2] \rightsquigarrow [\ \texttt{Return}\ \ b_1 \text{ and } b_2]}\ (\text{STEP}_{\cap})$$

□

Behavior of the subtype machine can be explained considering each rule individually: Rule (STEP_ω) is just rule (ω) from the BCD relation. Constructors are analyzed recursively by ($\text{STEP}_{\text{CTOR}}$). Auxiliary function $\text{cast}_{c(B)}$ collects all arguments to recursively compare. For example, if $d \leq_{\mathbf{C}} c$, $f \leq_{\mathbf{C}} c$, and not $e \leq_{\mathbf{C}} c$ the result of casting is $\text{cast}_{c(B)}(d(A_1) \cap (A_2 \to A_3) \cap e(A_4) \cap f(A_5)) = [::\ A_1 \& [::A_5]]$. If the list returned by cast is empty, either type shapes did not match or there was no compatible constructor symbol. It is crucial to check this, because the premise $[\ \texttt{subty}\ \bigcap_{A_i \in \text{cast}_{c(\omega)}(A)} A_i\ \texttt{of}\ \omega] \rightsquigarrow [\ \texttt{Return}\ b]$ is fulfilled for any A, even if it is structurally fully incompatible with $c(\omega)$, e.g. $A = (A_1 \star A_2) \cap (A_3 \to A_4)$. Formal verification in Coq spotted forgetting this check at an early stage of development. Rule ($\text{STEP}_\star$) for the product constructor is analogous to ($\text{STEP}_{\text{CTOR}}$). Arrows are more difficult to process because of the aforementioned switch in variance of their first argument. Three rules are required. The entry point, ($\text{STEP}_\to$), ensures subtyping of the second argument and identifies arrows collapsing to ω using isOmega. The second argument cannot be directly compared. If multiple arrows are present in A, their distribution by (DIST) can be required for subtyping. However, just collecting all targets in A does not work: if $A \leq A_1$, $A \leq A_2$, and not $A \leq A_3$ distribution in $(A_1 \to B_1) \cap (A_2 \to B_2) \cap (A_3 \to B_3)$ results in $A_1 \cap A_2 \cap A_3 \to B_1 \cap B_2 \cap B_3$ which is not a subtype of $A \to B_1 \cap B_2$, even though just distributing the first two arrows results in $(A_1 \to B_1) \cap (A_2 \to B_2) \cap (A_3 \to B_3) \leq A_1 \cap A_2 \to B_1 \cap B_2 \leq A \to B_1 \cap B_2$. Instruction $[\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \text{cast}_{B_1 \to B_2}(A)]$ in the first premise of ($\text{STEP}_\to$) is responsible for selecting the correct arrows in A. Rules ($\text{STEP}_{\text{CHOOSETGT}}$) and ($\text{STEP}_{\text{DONETGT}}$) implement this computation. If the input list created by $\text{cast}_{B_1 \to B_2}(A)$ is fully processed or empty no additional arrow target is to be selected ($\text{STEP}_{\text{DONETGT}}$). If there is a remaining arrow, its first argument is compared to B_1 and the second argument is added to the list of targets to compare if this check is successful. In the above example, this results in B_1 and B_2 being added to Δ' while B_3 is dropped. The final rule ($\text{STEP}_\cap$) is an implementation of (GLB): components of intersections are just recursively compared. Unlike $\leq$, relation $\rightsquigarrow$ does not include cut rules and the choice of rule to apply is fully determined by the input instruction. The following proofs will establish that $\rightsquigarrow$ can be implemented by a recursively defined function and that it is correct (sound and complete) with respect to relation $\leq$.

Lemma 1 (Functionality of the BCD Subtype Machine) *For all instructions $i \in \mathbb{I}_{\text{Sub}}$ and outputs $o_1, o_2 \in \mathbb{O}_{\text{Sub}}$, if $i \rightsquigarrow o_1$ and $i \rightsquigarrow o_2$ then $o_1 = o_2$.*

PROOF Induction on the proof of $i \rightsquigarrow o_1$ followed by case analysis on $i \rightsquigarrow o_2$ ■

The termination certificate Dom can be defined next.

Definition 8 (Termination certificate for the BCD Subtype Machine) For any input $i \in \mathbb{I}_{\text{Sub}}$, termination certificate Dom i is a finite proof tree constructed from the following rules:

$$\frac{}{\text{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ \omega]} \qquad \frac{\boxed{1}\ \text{Dom}[\ \texttt{subty}\ \bigcap_{A_i \in \text{cast}_{c(B)}(A)} A_i\ \texttt{of}\ B]}{\text{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ c(B)]}$$

$$\frac{\begin{array}{c}\boxed{1}\ \mathrm{Dom}[\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \mathrm{cast}_{B_1\to B_2}A] \\ \boxed{2}\ \text{for all } \Delta\text{, if } [\ \texttt{tgt\_for\_srcs\_gte}\ B_1\ \texttt{in}\ \mathrm{cast}_{B_1\to B_2}A] \rightsquigarrow [\ \texttt{check\_tgt}\ \Delta] \\ \text{then } \mathrm{Dom}[\ \texttt{subty}\ \bigcap_{A_i\in\Delta} A_i\ \texttt{of}\ B_2]\end{array}}{\mathrm{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ B_1 \to B_2]}$$

$$\frac{\boxed{1}\ \mathrm{Dom}[\ \texttt{subty}\ B\ \texttt{of}\ A.1] \quad \boxed{2}\ \mathrm{Dom}[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ \Delta]}{\mathrm{Dom}[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::A\&\Delta]]}$$

$$\frac{}{\mathrm{Dom}[\ \texttt{tgt\_for\_srcs\_gte}\ B\ \texttt{in}\ [::]]}$$

$$\frac{\begin{array}{c}\boxed{1}\ \mathrm{Dom}[\ \texttt{subty}\ \bigcap_{A_i\in\mathrm{cast}_{B_1\star B_2}(A)} A_i.1\ \texttt{of}\ B_1] \\ \boxed{2}\ \mathrm{Dom}[\ \texttt{subty}\ \bigcap_{A_i\in\mathrm{cast}_{B_1\star B_2}(A)} A_i.2\ \texttt{of}\ B_2]\end{array}}{\mathrm{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ B_1\star B_2]} \qquad \frac{\begin{array}{c}\boxed{1}\ \mathrm{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ B_1] \\ \boxed{2}\ \mathrm{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ B_2]\end{array}}{\mathrm{Dom}[\ \texttt{subty}\ A\ \texttt{of}\ B_1\cap B_2]}$$

Childproofs of a tree $p \in \mathrm{Dom}\, i$ are selected using $p.1$ and $p.2$ where the boxed number in front of each premise specifies which projection is chosen. □

The rules of Dom closely follow those of $\rightsquigarrow$. Each recursive use of $\rightsquigarrow$ is modeled by a child proof in Dom, while other information - especially the result value - is erased. This leads to Dom being just weak enough to poof totality.

Lemma 2 *Totality of the Termination Certificate* Dom *For all instructions $i \in \mathbb{I}_{\mathrm{Sub}}$ there exists a proof tree* Dom i.

PROOF First if A is ω, $\mathrm{Dom}[\ \texttt{subty}\ \omega\ \texttt{of}\ B]$ is obtained by induction on the structure of B. For the other cases, induction on the maximal depth of A and B is followed by induction on the structure of B. Function $\mathrm{cast}_B(A)$ either returns components with a smaller depth than A, or $[::\omega]$ or $[::(\omega,\omega)]$. The ω-cases are dispatched by the earlier proof. Otherwise the list returned by $\mathrm{cast}_B(A)$ contains only types with depth less than A. For (STEP$_{\mathrm{CTOR}}$), (STEP$_\star$), and (STEP$_\cap$) the induction hypotheses solve the goal. For rule (STEP$_\to$) target collection in the first premise is shown to compute a list of targets with depth less than A by induction on the length of $\mathrm{cast}_{B_1\to B_2}(A)$ using the outer induction hypothesis for checking if B_1 is a subtype of any of the sources. This is possible in spite of the toggled position of B_1, because the maximal depth of types A and B ignores the order of A and B. Any target list returned by invocations of (STEP$_{\mathrm{ChooseTgt}}$) and (STEP$_{\mathrm{DoneTgt}}$) only contains targets produced by cast $\mathrm{cast}_{B_1\to B_2}(A)$ and their intersection has a depth smaller than B_2, because depth does not increase when performing intersection operations. This is sufficient to complete the proof for the second premise of (STEP$_\to$) using the induction hypotheses. ■

Functionality and totality are enough to translate the operational semantics $\rightsquigarrow$ into denotational semantics, a recursive procedure to decide subtyping.

Definition 9 (Procedure deciding the BCD Subtype Relation)

$$
\begin{array}{l}
\text{subtypes}(i \in \mathbb{I}_{\text{Sub}}) : \text{Dom}\, i \to \{o \in \mathbb{O}_{\text{Sub}} \mid i \rightsquigarrow o\} \\
\text{subtypes}(i)(p) = \\
\left\{
\begin{array}{l}
[\ \texttt{Return}\ \ \text{true}] \text{ if } i = [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ \omega] \\
[\ \texttt{Return}\ \ \text{cast}_{c(B)}(A) \neq [::] \text{ and } b] \\
\quad \text{if } i = [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ c(B)] \text{ and} \\
\quad \text{for } A' := \bigcap_{\text{cast}_{c(B)}(A)} A_i \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A'\ \ \texttt{of}\ \ B])(p.1) = [\ \texttt{Return}\ \ b] \\
[\ \texttt{Return}\ \ \text{isOmega}(B_2) \text{ or } b] \\
\quad \text{if } i = [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \to B_2] \text{ and} \\
\quad \text{for } A_1 := \text{cast}_{B_1 \to B_2}(A) \\
\quad \text{subtypes}([\ \texttt{tgt\_for\_srcs\_gte}\ \ B_1\ \ \texttt{in}\ \ A_1])(p.1) = \\
\qquad [\ \texttt{check\_tgt}\ \ \Delta] \text{ and} \\
\quad \text{for } A_2 := \bigcap_{A_i \in \Delta} A_i \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A_2\ \ \texttt{of}\ \ B_2])(p.2) = [\ \texttt{Return}\ \ b] \\
[\ \texttt{Return}\ \ \text{cast}_{B_1 \star B_2}(A) \neq [::] \text{ and } b_1 \text{ and } b_2] \\
\quad \text{if } i = [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \star B_2] \text{ and} \\
\quad \text{for } A_1 := \bigcap_{A_i \in \text{cast}_{B_1 \star B_2}(A)} A_i.1 \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A_1\ \ \texttt{of}\ \ B_1])(p.1) = [\ \texttt{Return}\ \ b_1] \text{ and} \\
\quad \text{for } A_2 := \bigcap_{A_i \in \text{cast}_{B_1 \star B_2}(A)} A_i.2 \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A_2\ \ \texttt{of}\ \ B_2])(p.2) = [\ \texttt{Return}\ \ b_2] \\
[\ \texttt{Return}\ \ b_1 \text{ and } b_2] \\
\quad \text{if } i = [\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1 \cap B_2] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_1])(p.1) = [\ \texttt{Return}\ \ b_1] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ \ A\ \ \texttt{of}\ \ B_2])(p.2) = [\ \texttt{Return}\ \ b_2] \\
[\ \texttt{check\_tgt}\ \ \text{if } b \text{ then } [::\ A.2 \& \Delta'] \text{ else } \Delta'] \\
\quad \text{if } i = [\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ [::\ A \& \Delta]] \text{ and} \\
\quad \text{subtypes}([\ \texttt{subty}\ \ B\ \ \texttt{of}\ \ A.1])(p.1) = [\ \texttt{Return}\ \ b] \text{ and} \\
\quad \text{subtypes}([\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ \Delta])(p.2) = \\
\qquad [\ \texttt{check\_tgt}\ \ \Delta'] \\
[\ \texttt{check\_tgt}\ \ [::]] \text{ if } i = [\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ [::]]
\end{array}
\right.
\end{array}
$$

□

Each case in function subtypes implements a rule of $\rightsquigarrow$. It thus respects its restricted range $\{o \in \mathbb{O}_{\text{Sub}} \mid i \rightsquigarrow o\}$. Recursive calls always use a child trees of $p \in \text{Dom}\, i$ and thus terminate

because p is finite. Lemma 2 ensures existence of p for any input instruction i. Lemma 1 proves that subtypes does not lose any solutions. In Coq the termination certificate is part of the universe of propositions, which is erased when extracting the specification to an executable functional target language such as Haskell or OCaml: The constructive proof of Lemma 2 is enough to ensure Dom proof trees exist and can be constructed without actually ever wasting resources for constructing them. Most executable programming languages accept recursive definitions without totality proofs and eliminate this overhead.

The proof that $\rightsquigarrow$ actually implements $\leq$ requires to show to

$$[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}] \text{ iff } A \leq B.$$

The first part of this bi-implication is easier. It requires establishing some key properties of $\leq$.

Lemma 3 (Properties of the BCD Subtype Relation)

1. $\text{intersect}(\text{map}(\lambda A_i.M)(\Delta_1 +\!\!+ \Delta_2)) \leq$
   $\text{intersect}(\text{map}(\lambda A_i.M)(\Delta_1)) \cap$
   $\text{intersect}(\text{map}(\lambda A_i.M)(\Delta_2))$
2. $\text{isOmega}(B)$ *implies* $A \leq B$
3. $\text{cast}_{c(B)}(A) \neq [::]$ *implies* $A \leq c(\bigcap_{A_i \in \text{cast}_{c(B)}(A)} A_i)$
4. $A \leq \bigcap_{A_i \in \text{cast}_{B_1 \to B_2}(A)} (A_i.1 \to A_i.2)$
5. $A \leq \bigcap_{A_i \in \text{cast}_{B_1 \times B_2}(A)} (A_i.1 \star A_i.2)$
6. $(A_1 \to B_1) \cap (A_2 \to B_2) \leq (A_1 \cap A_2) \to (B_1 \cap B_2)$
7. $\bigcap_{A_i \in \Delta} (A_i.1 \star A_i.2) \leq \quad (\bigcap_{A_i \in \Delta} A_i.1) \star (\bigcap_{A_i \in \Delta} A_i.2)$ *if* $\Delta \neq [::]$
8. *Let* X *be a set,* $f : X \to \mathbb{T}$*, and* $\Delta_1, \Delta_2 : X^*$*.*
   If for all x *in* Δ_1*:* x *is in* Δ_2 *then*
   $\bigcap_{A_i \in \Delta_1} f(x) \leq \bigcap_{A_i \in \Delta_2} f(x)$

PROOF Structural induction and case analysis. No difficult cases exist. ■

Lemma 4 (Soundness of the BCD Subtype Machine) *For all types* $A, B \in \mathbb{T}$*:*

$$\textit{if } [\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}] \text{ then } A \leq B.$$

PROOF The change in variance of arrows again requires to first perform induction on the maximal depth of types A and B. This is followed by structural induction on the derivation of $[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}]$. If the derivation ends in (STEP$_{\text{CTOR}}$), (STEP$_{\text{PROD}}$), (STEP$_{\omega}$) or (STEP$_{\cap}$) the inner structural hypothesis and Lemma 3 are enough to finish the proof. Rule (STEP$_{\to}$) additionally requires the first induction hypothesis for its first premise. ■

The proof of completeness requires to show that $\rightsquigarrow$ supports transitivity of subtyping as well as several other properties established by the following lemma.

Lemma 5 (Properties of the BCD Subtype Machine)

1. $\mathrm{isOmega}(B)$ *implies* $[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}]$
2. $[\texttt{ tgt\_for\_srcs\_gte } B1 \texttt{ in } \Delta] \rightsquigarrow [\texttt{ check\_tgt } \Delta']$ *implies* $\Delta' \sqsubseteq \mathrm{map}(\lambda A_i.A_i.2)(\Delta)$
3. $[\texttt{ tgt\_for\_srcs\_gte } B_1 \texttt{ in } \mathrm{cast}_{B_1 \to B_2} A] \rightsquigarrow [\texttt{ check\_tgt } \Delta]$ *and* $\mathrm{isOmega}(A)$ *implies* $\mathrm{isOmega}(A_i)$ *for all* A_i *in* Δ
4. $[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}]$ *and* $\mathrm{isOmega}(A)$ *implies* $\mathrm{isOmega}(B)$
5. $\Delta_2 \sqsubseteq \Delta_1$ *and* $[\texttt{ tgt\_for\_srcs\_gte } B_1 \texttt{ in } \Delta_1] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_1]$ *and* $[\texttt{ tgt\_for\_srcs\_gte } B_1 \texttt{ in } \Delta_2] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_2]$ *implies* $\Delta'_2 \sqsubseteq \Delta'_1$
6. $\Delta \sqsubseteq \Delta'$ *and* $[\texttt{ subty } \bigcap_{A_i \in \Delta} A_i \texttt{ of } A] \rightsquigarrow [\texttt{ Return true}]$ *implies* $[\texttt{ subty } \bigcap_{B_i \in \Delta'} B_i \texttt{ of } A] \rightsquigarrow [\texttt{ Return true}]$
7. $[\texttt{ subty } A \texttt{ of } \bigcap_{A_i \in \Delta_1} A_i] \rightsquigarrow [\texttt{ Return } b_1]$ *and* $[\texttt{ subty } A \texttt{ of } \bigcap_{A_i \in \Delta_2} A_i] \rightsquigarrow [\texttt{ Return } b_2]$ *implies* $[\texttt{ subty } A \texttt{ of } \bigcap_{A_i \in \Delta_3} A_i] \rightsquigarrow [\texttt{ Return } b_1 \wedge b_2]$ *for* $\Delta_3 = \Delta_1 +\!\!+ \Delta_2$
8. $[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}]$ *implies* $[\texttt{ subty } \bigcap_{A_i \in \mathrm{cast}_{c(C)} A} A_i \texttt{ of } \bigcap_{B_i \in \mathrm{cast}_{c(C)} B} B_i] \rightsquigarrow [\texttt{ Return true}]$
9. $[\texttt{ tgt\_for\_srcs\_gte } B \texttt{ in } [::(\omega, A)]] \rightsquigarrow [\texttt{ check\_tgt } \Delta]$ *implies* $\Delta = [::A]$
10. $[\texttt{ subty } A \texttt{ of } A] \rightsquigarrow [\texttt{ Return true}]$
11. $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_1] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_1]$ *and* $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_2] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_2]$ *implies* $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_3] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_3]$ *for* $\Delta_3 = \Delta_1 +\!\!+ \Delta_2$ *and* $\Delta'_3 = \Delta'_1 +\!\!+ \Delta'_2$
12. $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_1 +\!\!+ \Delta_2] \rightsquigarrow [\texttt{ check\_tgt } \Delta']$ *implies* *there exist* Δ'_1, Δ'_2, *s.t.* $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_1] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_1]$ *and* $[\texttt{ tgt\_for\_srcs\_gte } A \texttt{ in } \Delta_2] \rightsquigarrow [\texttt{ check\_tgt } \Delta'_2]$ *and* $\Delta' = \Delta'_1 +\!\!+ \Delta'_2$
13. $[\texttt{ subty } A \texttt{ of } B_1 \cap B_2] \rightsquigarrow [\texttt{ Return true}]$ *implies* $[\texttt{ subty } A \texttt{ of } B_1] \rightsquigarrow [\texttt{ Return true}]$ *and* $[\texttt{ subty } A \texttt{ of } B_2] \rightsquigarrow [\texttt{ Return true}]$
14. $C = c(C')$ *or* $C = C_1 \star C_2$ *and* $\mathrm{cast}_C B \neq [::]$ *and* $[\texttt{ subty } A \texttt{ of } B] \rightsquigarrow [\texttt{ Return true}]$ *implies* $\mathrm{cast}_C A \neq [::]$

15. [`subty` A `of` B] $\rightsquigarrow$ [`Return true`] *and*
    [`subty` B `of` C] $\rightsquigarrow$ [`Return true`] *implies*
    [`subty` A `of` C] $\rightsquigarrow$ [`Return true`]

16. [`subty` $c(A_1) \cap c(A_2)$ `of` $c(A_1 \cap A_2)$] $\rightsquigarrow$ [`Return true`]

17. [`subty` $A \cap A$ `of` A] $\rightsquigarrow$ [`Return true`]

PROOF The statements are proven in the order they are presented. Most just require induction and/or case analysis. The weakening property in case 6 is inspired by [125]. It is needed for the reflexivity proof for $A = A_1 \cap A_2$. Transitivity in case 14 again needs nested structural induction with an outer induction on the maximum of the depth of the types involved. ■

Lemma 6 (Completeness of the BCD Subtype Machine) *For all types $A, B \in \mathbb{T}$:*

if $A \leq B$ then [`subty` A `of` B] $\rightsquigarrow$ [`Return true`].

PROOF Induction on the derivation of $A \leq B$ using Lemma 5. Lemma 5.14 takes care of the problematic transitivity rule (TRANS). ■

Correctness of the decision procedure in Definition 8 can now be proven.

Theorem 1 (Subtype decision procedure correctness) *For all $A, B \in \mathbb{T}$ there exists a termination certificate $p \in \mathrm{Dom}$* [`subty` A `of` B].
For all termination certificates $p \in \mathrm{Dom}$ [`subty` A `of` B] *there exists a Boolean b such that*

subtypes([`subty` A `of` B])(p) = [`Return` b] with b = true if and only if $A \leq B$.

PROOF The first part is Lemma 2. In the second part subtypes([`subty` A `of` B])(p) = [`Return` b] and $b = true$ can be replaced by either [`subty` A `of` B] $\rightsquigarrow$ [`Return true`] or [`subty` A `of` B] $\rightsquigarrow$ [`Return false`] because of the definition of subtypes and functionality of $\rightsquigarrow$. The result then immediately follows from Lemma 4 and Lemma 6. ■

Further analysis of the termination certificate provides some insight on a bound of the number of possible recursive calls.

Lemma 7 (Size of the Termination Certificate) *For $i \in \mathbb{I}_{\mathrm{Sub}}$, $p \in \mathrm{Dom}_i$ define*

$$\mathrm{measure}_i(p) = \begin{cases} 1 & \textit{for } i = [\texttt{ subty } A \texttt{ of } \omega] \text{ or} \\ & \quad [\texttt{ tgt\_for\_srcs\_gte } B \texttt{ in } [::]] \\ 1 + \mathrm{measure}(p.1) & \textit{for } i = [\texttt{ subty } A \texttt{ of } c(B)] \\ 1 + \mathrm{measure}(p.1) + \mathrm{measure}(p.2) & \textit{otherwise} \end{cases}$$

and obtain

$$\text{measure}_i(p) \leq \begin{cases} 2 \cdot \text{size}(A) \cdot \text{size}(B) & \textit{for } i = [\texttt{ subty } A \texttt{ of } B] \\ 1 + \text{size}(B) \cdot \sum_{j=1}^{k} (1 + 2 \cdot \text{size}(A_j.1)) & \\ \quad \textit{for } i = [\texttt{ tgt\_for\_srcs\_gte } B \texttt{ in } [::A_1 \& [::A_2 \& \ldots [::A_k] \ldots]]] & \end{cases}$$

PROOF The key insight is that types returned by $\text{cast}_B(A)$ always have sizes smaller or equal than the size of A. The result then follows by induction on p with a special case for $i = [\texttt{ subty } A \texttt{ of } B_1 \to B_2]$ where $B_2 = \omega$ and $\text{cost}_i(p) \leq 3$. ■

Kurata and Takahashi [118] identified an algorithmically important property of the restricted set of types considered in precursors to the full BCD system [43]. This property has been the basis for the algorithms presented in [165] and has been algebraically classified [24] as primality of the ideal induced by $\leq$.

Definition 10 (Primality) The ideal of $A \in \mathbb{T}$ is the non-empty set $\{B \in \mathbb{T} | B \leq A\}$ and A is its principal element. An ideal with principal element C is prime if for all $A \cap B$ in the ideal, $A \leq C$ or $B \leq C$. Type C is prime if it is the principal element of a prime ideal. □

Prime ideals are desirable because they avoid distribution. For example if C prime, then in $(A_1 \to B_1) \cap (A_2 \to B_2) \leq C$ the distribution $(A_1 \cap A_2 \to B_1 \cap B_2) \leq C$ never has to be taken into account, because one of the types $(A_1 \to B_1)$ and $(A_2 \to B_2)$ is less than C. One of the particularly beautiful facts about intersection types is that primality is a purely syntactic criterion and the precursor system [43] correctly captured that notion. Here, the syntactic criterion is extended to account for constructor symbols and products.

Lemma 8 (Syntactic criterion for Primality) *The set*

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

is identical to the set of prime types.

PROOF By Lemma 4 and Lemma 6 it is sufficient to show

$[\texttt{ subty } A \cap B \texttt{ of } \pi] \rightsquigarrow [\texttt{ Return true}]$ implies

$[\texttt{ subty } A \texttt{ of } \pi] \rightsquigarrow [\texttt{ Return true}]$ or

$[\texttt{ subty } B \texttt{ of } \pi] \rightsquigarrow [\texttt{ Return true}]$

Induction on C uniquely identifies the last rule used to derive $[\texttt{ subty } A \cap B \texttt{ of } \pi] \rightsquigarrow [\texttt{ Return true}]$ in each case and the result then easily follows. ■

Let $\mathbb{T}_\pi = \{\pi \in \mathbb{T}_\pi^\omega \mid \text{ not isOmega}(\pi)\}$ denote the set of prime types that do not collapse to ω under subtyping. In [43] this set is called the set of tail-proper types. In [57] and [67] these types are called paths and it is proven that every type can be organized into a subtype equal set of paths.

Definition 11 (Prime factorization of Types) The following function computes a minimal sequence of prime factors of type $A \in \mathbb{T}$:

$$\text{primeFactors}(A) = \text{primeFactors}'(A, \text{id}, [::])$$

$$\text{primeFactors}'(A, \text{contextualize}, \Delta) = \begin{cases} \Delta & \text{for } A = \omega \text{ and } \text{isOmega}(\text{contextualize}(A)) \\ \text{addAndFilter}(\Delta, \text{contextualize}(A)) & \text{for } A = \omega \text{ and not } \text{isOmega}(\text{contextualize}(A)) \\ \text{primeFactors}'(A', \lambda B.\,\text{contextualize}(c(B)), \Delta) & \text{for } A = c(A') \\ \text{primeFactors}'(A_2, \lambda B.\,\text{contextualize}(A_1 \to B), \Delta) & \text{for } A = A_1 \to A_2 \\ \text{primeFactors}'(A_2, \lambda B.\,\text{contextualize}(\omega \star B), \\ \quad \text{primeFactors}'(A_1, \lambda B.\,\text{contextualize}(B \star \omega), \Delta)) & \text{for } A = A_1 \star A_2 \\ \text{primeFactors}'(A_2, \text{contextualize}, \\ \quad \text{primeFactors}'(A_1, \text{contextualize}, \Delta)) & \text{for } A = A_1 \cap A_2 \end{cases}$$

$$\text{addAndFilter}(\Delta, A) = \begin{cases} [::A] & \text{for } \Delta = [::] \\ [::B\&\Delta'] & \text{for } \Delta = [::B\&\Delta'] \text{ and } B \leq A \\ \text{addAndFilter}(\Delta', A) & \text{for } \Delta = [::B\&\Delta'] \text{ and not } B \leq A \text{ and } A \leq B \\ [::B\&\text{addAndFilter}(\Delta', A)] & \text{otherwise} \end{cases}$$

□

Function primeFactors is implemented using primeFactors′, which keeps track of the accumulated list of prime factors Δ and the context of the currently analyzed type with regard to the initial type. This context is represented as a function mapping the current type into the structure of the parent type. The context representation is inspired by Zippers, a functional programming technique [104; 101]. Recursion in primeFactors′ finishes when ω is reached. If ω is less or equal than the computed prime factor at this point, it is redundant, just like 1 is redundant as a prime factor in a natural number. Otherwise it is inserted into the list of prime factors using addAndFilter. Function addAndFilter appends a prime factor A to a list of prime factors Δ if that list contains no subtype related element. Otherwise, if there is such an element, and it is smaller than A, Δ remains constant. If the existing element is greater than A, it will be replaced. The result of addAndFilter is a minimal list of prime factors without duplication of subtype equal types. In any case which is not ω, function primeFactors′ recursively proceeds, adapting the context with the operation to map the next considered argument back into place. Products have to be split recontextualizing their left and right prime components separately, setting the other component to ω (c.f. the restricted shape of products in $\mathbb{T}_\pi$). The following lemma is necessary to establish correctness and minimality of primeFactors.

Lemma 9 (Prime factorization verification properties)
Let $\Delta, \Delta_1, \Delta_2, \Delta_3, \Delta'$ be universally quantified lists of types and $A, B, C, D \in \mathbb{T}$, then:

1. *There exists some B in* $\text{addAndFilter}(\Delta, A)$*, s.t. $B \leq A$*
2. *There exists some A' in* $\text{addAndFilter}(\Delta, A)$*, s.t. $A' \leq C$ iff $A \leq C$ or there exists B' in Δ s.t. $B' \leq C$.*
3. *If $\Delta \sqsubseteq \Delta'$ and there exists A' in* $\text{addAndFilter}(\Delta, A)$ *s.t. $A' \leq B$ then there exists A'' in* $\text{addAndFilter}(\Delta', A)$ *s.t. $A'' \leq B$*
4. *If B is in* $\text{addAndFilter}(\Delta, A)$ *then $B = A$ or B is in Δ*
5. *If A is in Δ and $A \leq B$ then $\bigcap_{A_i \in \Delta} A_i \leq B$*
6. $\bigcap_{A_i \in \text{addAndFilter}(\Delta, A)} A_i \leq \bigcap_{A_i \in \Delta} A_i$
7. *Let $P : \mathbb{T} \to \mathbb{B}$ then for all A in Δ: P(A) implies for all B in* $\text{addAndFilter}(\Delta)$*: P(B)*
8. *If for all A in Δ: $A \in \mathbb{T}_\pi$ and for all $B \in \mathbb{T}_\pi$:* $\text{contextualize}(B) \in \mathbb{T}_\pi$ *then for all D in* $\text{primeFactors}'(C, \text{contextualize}, \Delta)$*: $D \in \mathbb{T}_\pi$*
9. $\bigcap_{A_i \in \text{primeFactors}'(A, \text{contextualize}, \Delta)} A_i \leq \bigcap_{A_i \in \Delta} A_i$
10. *If for all $B, C \in \mathbb{T}$: $B \leq C$ implies* $\text{contextualize}(B) \leq \text{contextualize}(C)$ *and* $\text{contextualize}(B) \cap \text{contextualize}(C) \leq \text{contextualize}(B \cap C)$ *then* $\bigcap_{A_i \in \text{primeFactors}'(A, \text{contextualize}, \Delta)} A_i \leq A$
11. *If B is in* $\text{addAndFilter}(\Delta, A)$ *then there exists C in* $[:: A \& \Delta]$ *s.t. $C \leq B$*
12. *If for all A in Δ_2 there exists B in Δ_1 s.t. $B \leq A$ then $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$*
13. $\bigcap_{A_i \in [::A \& \Delta]} A_i \leq \bigcap_{A_i \in \text{addAndFilter}(\Delta, A)} A_i$
14. *If for all $B, C \in \mathbb{T}$: $B \leq C$ implies* $\text{contextualize}(B) \leq \text{contextualize}(C)$ *then* $\bigcap_{A_i \in [::A \& \Delta]} A_i \leq \bigcap_{A_i \in \text{primeFactors}'(A, \text{contextualize}, \Delta)} A_i$

Define lists free of subtype equal duplicates

$$\text{nosubdup}(\Delta) = \begin{cases} \textit{true} & \textit{for } \Delta = [::] \\ \text{nosubdup}(\Delta') \textit{ and} & \textit{for } \Delta = [:: A \& \Delta'] \\ \textit{for all } B \textit{ in } \Delta : \textit{not } A \leq B \textit{ and not } B \leq A & \end{cases}$$

15. $\text{nosubdup}(\Delta)$ *implies* $\text{nosubdup}(\text{addAndFilter}(\Delta, A))$
16. $\text{nosubdup}(\Delta)$ *implies* $\text{nosubdup}(\text{primeFactors}'(A, \text{contextualize}, \Delta))$

17. *If for all A in Δ: not $\omega \leq A$ then for all B in* $\text{primeFactors}'(A, \text{contextualize}, \Delta)$*: not $\omega \leq A$*

Define deduplication modulo subtyping

$$\text{desubdup}(\Delta) = \begin{cases} [::] & \textit{for } \Delta = [::] \\ \text{addAndFilter}(\text{desubdup}(\Delta'), A) & \textit{for } \Delta = [::\ A \& \Delta'] \textit{ and not } \text{isOmega}(A) \\ \text{desubdup}(\Delta') & \textit{for } \Delta = [::\ A \& \Delta'] \textit{ and } \text{isOmega}(A) \end{cases}$$

18. $\text{nosubdup}(\text{desubdup}(\Delta))$

19. *Let $P : \mathbb{T} \to \mathbb{B}$ then for all A in Δ: P(A) implies for all B in* $\text{desubdup}(\Delta)$*: P(B)*

20. $\bigcap_{A_i \in \text{desubdup}(\Delta)} A_i \leq \bigcap_{A_i \in \Delta} A_i$

21. $\bigcap_{A_i \in \Delta} A_i \leq \bigcap_{A_i \in \text{desubdup}(\Delta)} A_i$

22. $\text{lsize}(\text{addAndFilter}(\Delta)) \leq \text{lsize}(\Delta)$

23. $\text{lsize}(\text{desubdup}(\Delta)) \leq \text{lsize}(\Delta)$

24. *If A in* $\text{desubdup}(\Delta)$ *then not* $\text{isOmega}(A)$

25. *If $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$ and*
    for all A in Δ_2: $A \in \mathbb{T}_\pi$ and not $\text{isOmega}(A)$ *then*
    for all A in Δ_2: exists B in Δ_1, s.t. $B \leq A$

26. *If $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$ and*
    for all A in Δ_2: $A \in \mathbb{T}_\pi$ and not $\text{isOmega}(A)$ *then*
    for $\Delta = \text{filter}(\lambda A.$ *exists B in $\Delta_2 : A \leq B$*$)\Delta_1$
    $\bigcap_{A_i \in \Delta} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$

27. *If* $\text{nosubdup}(\Delta)$ *and A, B in Δ then $A \leq B$ implies $A = B$*

28. *If $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$ and*
    for all A in Δ_2: $A \in \mathbb{T}_\pi$ and not $\text{isOmega}(A)$ *and*
    $\text{nosubdup}(\Delta_1)$ *and* $\text{nosubdup}(\Delta_2)$ *and*
    A_1, A_2 are in Δ_2 and B is in Δ_1 and $A_1 \leq B$ and $A_2 \leq B$ then $A_1 = A_2$

29. *If $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$ and $\bigcap_{A_i \in \Delta_2} A_i \leq \bigcap_{A_i \in \Delta_1} A_i$ and*
    for all A in Δ_2: $A \in \mathbb{T}_\pi$ and not $\text{isOmega}(A)$ *and*
    $\text{nosubdup}(\Delta_1)$ *and* $\text{nosubdup}(\Delta_2)$ *and*
    A is in Δ_1 and B is in Δ_2 and $B \leq A$ then $A \leq B$

30. *If $\Delta_1 \sqsubseteq \Delta_2$ and* $\text{nosubdup}(\Delta_2)$ *then* $\text{nosubdup}(\Delta_1)$

Define the permutations of a list of types up to subtyping $\mathrm{Perm}_{\leq}$ *as the least set closed under the following rules:*

$$\frac{A \leq B \qquad B \leq A \qquad \Delta_1 \in \mathrm{Perm}_{\leq}(\Delta_2 +\!\!+ \Delta_3)}{[::A\&\Delta_1] \in \mathrm{Perm}_{\leq}(\Delta_2 +\!\!+ [::B\&\Delta_3])} \qquad \frac{}{[::] \in \mathrm{Perm}_{\leq}[::]}$$

31. *If* $\bigcap_{A_i \in \Delta_1 +\!\!+ [::A\&\Delta_2]} A_i \leq \bigcap_{A_i \in \Delta_3} A_i$ *and*
    for all B *in* Δ_3*:* $B \in \mathbb{T}_\pi$ *and not* $A \leq B$ *then*
    $\bigcap_{A_i \in \Delta_1 +\!\!+ \Delta_2} A_i \leq \bigcap_{A_i \in \Delta_3} A_i$

32. *If* $\mathrm{nosubdup}(\Delta_1 +\!\!+ [::A\&\Delta_2])$ *then*
    forall B *in* $\Delta_1 +\!\!+ \Delta_2$*: not* $B \leq A$ *and not* $A \leq B$

33. *If* $\bigcap_{A_i \in \Delta_1} A_i \leq \bigcap_{A_i \in \Delta_2} A_i$ *and* $\bigcap_{A_i \in \Delta_2} A_i \leq \bigcap_{A_i \in \Delta_1} A_i$ *and*
    $\mathrm{nosubdup}(\Delta_1)$ *and* $\mathrm{nosubdup}(\Delta_2)$ *and*
    for all A *in* Δ_1*:* $A \in \mathbb{T}_\pi$ *and not* $\mathrm{isOmega}(A)$ *and*
    for all A *in* Δ_2*:* $A \in \mathbb{T}_\pi$ *and not* $\mathrm{isOmega}(A)$ *then*
    $\Delta_1 \in \mathrm{Perm}_{\leq}(\Delta_2)$

34. *If* $\Delta_1 \in \mathrm{Perm}_{\leq}(\Delta_2)$ *then* $\mathrm{lsize}(\Delta_1) = \mathrm{lsize}(\Delta_2)$

PROOF The above statements should be proven in the order they were listed. Then they follow from easy induction, case-analysis, and application of previous statements. ■

Theorem 2 (Prime factorization correctness and minimality)
For all $A \in \mathbb{T}$, $\mathrm{primeFactors}(A)$ *is a subtype equal minimal list of prime factors of* A*:*

1. $\bigcap_{A_i \in \mathrm{primeFactors}(A)} A_i \leq A$
2. $A \leq \bigcap_{A_i \in \mathrm{primeFactors}(A)} A_i$
3. *For all* B *in* $\mathrm{primeFactors}(B)$*:* $B \in \mathbb{T}_\pi$
4. *For all* Δ*, if* $\bigcap_{A_i \in \Delta} A_i \leq A$ *and* $A \leq \bigcap_{A_i \in \Delta} A_i$ *and for all* B *in* Δ*:* $B \in \mathbb{T}_\pi$ *then*
   $\mathrm{lsize}(\mathrm{primeFactors}(A)) \leq \mathrm{lsize}(\Delta)$

PROOF

1. Direct consequence of Lemma 9.10.
2. Direct consequence of Lemma 9.14.
3. Direct consequence of Lemma 9.8.
4. By Lemma 9.23, Δ can be deduplicated with decreasing size. By Lemma 9.34 it is sufficient to show that $\mathrm{primeFactors}(A) \in \mathrm{Perm}_{\leq}(\mathrm{desubdup}(\Delta))$. The result is then obtained using Lemma 9.33 with Theorem 2.1, Theorem 2.2, Lemma 9.16, Lemma 9.18, the premise and Lemma 9.24, as well as Theorem 2.3, Lemma 9.19 and Lemma 9.17. ■

2.4 Finite Combinatory Logic with Intersection Types (FCL)

2.4.1 Basic Properties and Decidability of Type Checking

Finite Combinatory Logic with Intersection Types [165] is the main formalism driving synthesis in this work. In this section some of its basic properties are discussed and the type checking question for FCL is proven to be decidable.

Definition 12 (Finite Combinatory Logic with Intersection Types) Let $\mathbf{B}$ be a finite countable base of combinators. The set of applicative terms is defined by

$$\mathbb{A} \ni M, N, O ::= c \mid @(M, N)$$

where $c \in \mathbf{B}$ is a combinator and $@(M, N)$ is the application of term M to argument N. Term application is also abbreviated to (MN) using the convention that application associates to the left and omitting outermost parentheses: $MNO = ((MN)O) = @(@(M, N), O)$.

Given a function $\Gamma : \mathbf{B} \to \mathbb{T}$, called context, the type relation of Finite Combinatory Logic with Intersection Types is the least relation $\vdash \subset (\mathbf{B} \to \mathbb{T}) \times \mathbb{A} \times \mathbb{T}$ closed under the following rules:

$$\frac{}{\Gamma \vdash c : \Gamma(c)}\ (\textsc{Var}) \quad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ (\to \mathrm{E}) \quad \frac{\Gamma \vdash M : A \qquad A \leq B}{\Gamma \vdash M : B}\ (\leq) \qquad \square$$

In $\Gamma \vdash M : A$ Term M has type A in context Γ. Context Γ assigns types to combinators. This assignment is available in type judgments via rule (VAR). Terms are combined into new terms using application and the arrow elimination rule ($\to$ E) controls which terms are safe to combine. An intuitive understanding reads $\Gamma \vdash M : A \to B$ as a judgment proving that M is a function taking inputs of type A to outputs of type B. The only extensions to Combinatory Logic with Simple Types [176] are the richer type syntax and rule ($\leq$) that changes type judgments according to the rules of BCD Subtyping.

Proofs about type judgments use the following case-analysis and induction schemes.

Lemma 10 (Induction on and Inversion of FCL Type Judgments) *Let $\Gamma : \mathbf{B} \to \mathbb{T}$ and P be a proposition over terms and types, then:*

1. *Structural induction proofs use:*
   If for all $c \in \mathbf{B}$: $P(c, \Gamma(c))$
   and for all $M_1, M_2 \in \mathbb{A}$, $A_1, A_2 \in \mathbb{T}$:
   $\Gamma \vdash M_1 : A_1 \to A_2$ *and* $P(M_1, A_1 \to A_2)$ *and*
   $\Gamma \vdash M_2 : A_1$ *and* $P(M_2, A_1)$ *implies*
   $P(M_1 M_2, A_2)$
   and for all $N \in \mathbb{A}$, $A_1, A_2 \in \mathbb{T}$:
   $\Gamma \vdash N : A_1$ *and* $P(N, A_1)$ *and* $A_1 \leq A_2$ *implies* $P(N, A_2)$
   then for all $M \in \mathbb{A}$, $A \in \mathbb{T}$: $\Gamma \vdash M : A$ implies $P(M, A)$

2. *Combinator judgments can be reduced to Subtype judgments:*
   for all $c \in \mathbf{B}$, $A \in \mathbb{T}$*: If* $\Gamma \vdash c : A$ *then* $\Gamma(c) \leq A$

3. *Arrow elimination can be inverted:*
   for all $M, N \in \mathbb{A}$, $B \in \mathbb{T}$*:*
   If $\Gamma \vdash MN : B$ *then there exists* A*, s.t.* $\Gamma \vdash M : A \to B$ *and* $\Gamma \vdash N : A$

4. *Normalized induction proofs use:*
   If for all $c \in \mathbf{B}$*:* $P(c, \Gamma(c))$
   and for all $c \in \mathbf{B}$, $A \in \mathbb{T}$*:*
   $P(c, \Gamma(c))$ *and* $\Gamma(c) \leq A$ *implies* $P(c, A)$
   and for all $M_1, M_2 \in \mathbb{A}$, $A_1, A_2 \in \mathbb{T}$*:*
   $\Gamma \vdash M_1 : A_1 \to A_2$ *and* $P(M_1, A_1 \to A_2)$ *and*
   $\Gamma \vdash M_2 : A_1$ *and* $P(M_2, A_1)$ *implies*
   $P(M_1 M_2, A_2)$
   then for all $M \in \mathbb{A}$, $A \in \mathbb{T}$*:* $\Gamma \vdash M : A$ *implies* $P(M, A)$

PROOF

1. Recursion using the fact that trees are finite objects and children get smaller in each step.

2. Structural induction using the scheme from Lemma 10.1.

3. Structural induction using the scheme from Lemma 10.1.

4. Structural induction using the scheme from Lemma 10.1 and then Lemma 10.2 for the subtyping case, and Lemma 10.3 arrow elimination case. ■

The formalization is set up such that two more rules are derivable, which are sometimes explicitly included into the system [50; 165].

Lemma 11 (Rules derivable in FCL) *The following two additional rules are derivable in FCL:*

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

PROOF Both by structural induction on the derivation. For ($\cap$I) additionally Lemma 10.2 and Lemma 10.3 are used. The proof for rule ($\mathbb{A}\omega$) uses the fact that Γ is represented as a function and has to assign some type A to every combinator, which then can be converted into an arrow in case of application using BCD rules (ω) and ($\to \omega$): $A \leq \omega \leq \omega \to \omega$. ■

The system also allows weakening by making subtype assumptions smaller.

Lemma 12 (Weakening Subtypes) *For all* $\Gamma_1, \Gamma_2 : \mathbf{B} \to \mathbb{T}$*: if for all* $c \in \mathbf{B}$*:* $\Gamma_2(c) \leq \Gamma_1(c)$ *then for all* $M \in \mathbb{A}$, $A \in \mathbb{T}$*: if* $\Gamma_1 \vdash M : A$ *then* $\Gamma_2 \vdash M : A$.

PROOF By normalized induction (Lemma 10.4) on the derivation $\Gamma_1 \vdash M : A$. ■

Many other properties of FCL are best expressed using the construct of multi arrows.

Definition 13 (Multi Arrows) A multi arrow m is a pair $\mathbb{T}^* \times \mathbb{T}$, where the first component, srcs, is a list of source types and the second component, tgt, is a target type. Projections $m.1$ and $m.2$ select these components.

Multi arrows are converted into types using:

$$\text{mkArrow}(m) = \text{foldl}(\lambda t s.s \to t)(m.2)(m.1)$$ □

Lemma 13 (Properties of mkArrow) *For* $\text{src}, \text{tgt}, \text{tgt}_1, \text{tgt}_2 \in \mathbb{T}$*, and* $\text{srcs}, \text{srcs}_1, \text{srcs}_2 \in \mathbb{T}^*$ *function* mkArrow *has the following formal properties:*

1. $\text{mkArrow}(([::\text{src}\&\text{srcs}], \text{tgt})) = \text{last}(\text{src}, \text{srcs}) \to \text{mkArrow}((\text{srcs}, \text{tgt}))$
2. $\text{mkArrow}((\text{rcons}(\text{srcs}, \text{src}), \text{tgt})) = \text{mkArrow}((\text{srcs}, \text{src} \to \text{tgt}))$
3. $\text{arity}(\text{mkArrow}((\text{rcons}(\text{srcs}, \text{src}), \text{tgt}))) = \mathbb{T} \times \mathbb{T}$
4. *If* $\text{tgt}_1 \leq \text{tgt}_2$ *then* $\text{mkArrow}((\text{srcs}, \text{tgt}_1)) \leq \text{mkArrow}((\text{srcs}, \text{tgt}_2))$
5. *If* $\text{isOmega}(\text{tgt})$ *then* $\text{isOmega}(\text{mkArrow}((\text{srcs}, \text{tgt})))$
6. *If* $\text{tgt} \in \mathbb{T}_\pi$ *then* $\text{mkArrow}((\text{srcs}, \text{tgt})) \in \mathbb{T}_\pi$
7. $\text{mkArrow}((\text{srcs}, \text{tgt}_1 \cap \text{tgt}_2)) \leq \text{mkArrow}((\text{srcs}, \text{tgt}_1)) \cap \text{mkArrow}((\text{srcs}, \text{tgt}_2))$
8. *If* $\text{lsize}(\text{srcs}_1) = \text{lsize}(\text{srcs}_2)$ *and for all* $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}_1, \text{srcs}_2)$*:* $\text{src}_1 \leq \text{src}_2$ *then* $\text{mkArrow}((\text{srcs}_1, \text{tgt})) \leq \text{mkArrow}((\text{srcs}_2, \text{tgt}))$

PROOF In the order presented and by inserting the definition of mkArrow and induction or induction in reverse order on the source types. ■

There is an analogous operation to mkArrow on the term level.

Definition 14 (Reverse Application) Terms can be constructed from other terms $M \in \mathbb{A}$, $Ns \in \mathbb{A}^*$ using reverse application:

$$\text{revApply}((M, Ns)) = \text{foldr}(\lambda NM.@(M, N))(M)(Ns).$$

The inverse operation is defined by:

$$\text{unapply}(M) = \begin{cases} (c, [::]) & \text{for } M = c \\ (c, [::N\&Ns]) & \text{for } M = M'N \text{ and } \text{unapply}(M') = (c, Ns) \end{cases}$$

Lemma 14 (Properties of revApply and unapply) *For $M, N \in \mathbb{A}$, $c \in \mathbf{B}$, and $Ns \in \mathbb{A}^*$ functions* revApply *and* unapply *have the following formal properties:*

1. $\text{revApply}((M, \text{rcons}(Ns, N))) = \text{revApply}((@(M, N), Ns))$
2. $\text{revApply}((M, [::])) = M$
3. $\text{revApply}(\text{unapply}(M))) = M$
4. $\text{unapply}(\text{revApply}((c, Ns)) = (c, Ns)$

PROOF Inserting definitions and induction. ■

Being able to construct arrows and terms from lists generalizes (inverse) arrow elimination.

Lemma 15 (Generalized (inverse) Arrow Elimination)
Let $M \in \mathbb{A}$, $Ns \in \mathbb{A}^*$, $\text{tgt} \in \mathbb{T}$, $\text{srcs} \in \mathbb{T}^*$ and m be a multi arrow, then judgments can be (de-)constructed using:

1. *If* $\text{lsize}(Ns) = \text{lsize}(m.1)$ *and for all* $n \in \mathbb{N}$*:* $\Gamma \vdash \text{nth}(M, Ns, n) : \text{nth}(\text{mkArrow}(m), m.1, n)$ *then* $\Gamma \vdash \text{revApply}((M, Ns)) : m.2$
2. *If* $\Gamma \vdash \text{revApply}(M, Ns) : \text{tgt}$ *then there exists* $\text{srcs} \in \mathbb{T}^*$*, s.t.* $\text{lsize}(Ns) = \text{lsize}(\text{srcs})$ *and for all* $n \in \mathbb{N}$*:* $\Gamma \vdash \text{nth}(M, Ns, n) : \text{nth}(\text{mkArrow}((\text{srcs}, \text{tgt})), \text{srcs}, n)$

PROOF By induction on Ns, using Lemma 10.3 for the second statement. ■

Moreover, minimal types can now be assigned to any given term M.

Definition 15 (Minimal Types) The minimal type of $M \in \mathbb{A}$ in context $\Gamma : \mathbf{B} \to \mathbb{T}$ is found by:

$$\text{minimalType}(\Gamma, M) =$$
$$\begin{cases} \Gamma(c) & \text{for } M = c \\ \bigcap_{A_i \in \Delta} A_i & \\ \text{for } M = M'N \text{ and} & \\ \quad [\ \texttt{tgt\_for\_srcs\_gte}\ \ \text{minimalType}(\Gamma, N)\ \ \texttt{in}\ \ \text{cast}_{\omega \to \omega \star \omega}\, \text{minimalType}(\Gamma, M')] & \\ \quad \rightsquigarrow [\ \texttt{check\_tgt}\ \ \Delta] & \end{cases}$$

where Δ is computed using Definition 9 with a termination certificate obtained by Lemma 2.□

Lemma 16 (Minimal Type Correctness) *Let $M \in \mathbb{A}$, $A, B, C \in \mathbb{T}$ and $\Gamma : \mathbf{B} \to \mathbb{T}$ then function* minimalType *computes the correct minimal type:*

1. *For all Δ in* $[\ \texttt{tgt\_for\_srcs\_gte}\ \ B\ \ \texttt{in}\ \ \text{cast}_{\omega \to \omega \star \omega}\, A] \rightsquigarrow [\ \texttt{check\_tgt}\ \ \Delta]$*:* $A \leq B \to \bigcap_{A_i \in \Delta} A_i$

2. $\Gamma \vdash M : \text{minimalType}(\Gamma, M)$

3. *If* $A \leq B \to C$ *then*
   for all Δ *in* [`tgt_for_srcs_gte` B `in` $\text{cast}_{\omega\to\omega\star\omega}\, A$] $\rightsquigarrow$ [`check_tgt` Δ]*:*
   $\bigcap_{A_i \in \Delta} A_i \leq C$

4. *If* $\Gamma \vdash M : A$ *then* $\text{minimalType}(\Gamma, M) \leq A$

PROOF The intermediate statements 1 and 3 are proven by analysis of the last step of the subtype machine, and each time two separate cases for the result of isOmega(B), showing that $\text{cast}_{\omega\to\omega\star\omega}$ will find appropriate targets. Soundness in statement 2 is shown by structural induction using previous case 1 for ($\to$ E). Minimality in statement 4 is shown by normalized induction using the previous statements 3 for ($\to$ E). ■

Minimal type inference and decidability of the BCD relation are enough to provide a type-checking algorithm for FCL.

Theorem 3 (Type Checking in FCL) *Let* $M \in \mathbb{A}$, $A \in \mathbb{T}$ *and* $\Gamma : \mathbf{B} \to \mathbb{T}$ *then*

$$\Gamma \vdash M : A \textit{ iff } \text{minimalType}(\Gamma, M) \leq A$$

PROOF The forward direction of the iff is minimality from Lemma 16.4, the other direction follows from soundness Lemma 16.2 and rule ($\leq$). ■

The rest of this section is dedicated to the study of type inhabitation. The algorithm to infer minimal types solves the problem: Given a context Γ, and a term M, find the minimal type, such that $\Gamma \vdash M : A$ is derivable. Type inhabitation solves the same problem for terms: Given a context Γ and type A, find all terms M such that $\Gamma \vdash M : A$ is derivable.

2.4.2 Combining Separate Domains of Discourse

Finite Combinatory Logic with Intersection Types has the power to reason about multiple domains of discourse simultaneously. Sometimes these domains are hard to describe in a technically precise way. Steffen et al. [181] have introduced taxonomic modeling with semantic type descriptors in the setting of synthesis. This was later adopted for intersection types and inhabitation based synthesis [163]. Successful applications have been demonstrated multiple times in various different scenarios [57; 59; 20; 21; 23; 96; 25; 171; 201]. Yet, with the notable exception of [59] where the type system was modified to separate semantic domains, no formal study has been conducted on mathematical conditions that make a separation of domains of discourse possible. At this point, enough theory has been established to illustrate semantic types and domain separation by example.

Example 1 (Semantic Types) Suppose synthesis is used to create sheets for the music running in background to motivate reading or writing this text. The sheet is typeset by LilyPond [143], which can process a textual markup language. One combinator, `addLyrics`, inserts lyrics text into a LilyPond markup file. Another combinator, `motivationSong`, provides a markup file of "No Surrender" by Bruce Springsteen. Imagine that an automated web search has downloaded lyrics including the text "surrender" and two combinators have been created. The first combinator, `noSurrender`, provides the correct lyrics. The second combinator, `ISurrender`, provides the lyrics of "I Surrender" by Celine Dion.

In a technical domain of discourse these combinators can be typed as follows:

$$\begin{aligned}\Gamma_1 = \{\, & \texttt{addLyrics} : \text{LilyPond(Music)} \to \text{String} \to \text{LilyPond(Song)} \\ & \texttt{motivationSong} : \text{LilyPond(Music)} \\ & \texttt{noSurrender} : \text{String} \\ & \texttt{ISurrender} : \text{String} \,\}\end{aligned}$$

Synthesis in the technical domain uses type inhabitation to create all terms for which type LilyPond(Song) is derivable, i.e. the set

$$\begin{aligned}\{\, & \texttt{addLyrics}\ \texttt{motivationSong}\ \texttt{noSurrender}, \\ & \texttt{addLyrics}\ \texttt{motivationSong}\ \texttt{ISurrender} \,\}.\end{aligned}$$

The first term represents the correct version with the advice "No retreat baby, no surrender" [177]. However the second term not only has a text out of tune, but also includes the disastrous advice "I surrender. Every night's getting longer" [51].

A different, nontechnical, domain of discourse describes artist information:

$$\begin{aligned}\Gamma_2 = \{\, & \texttt{addLyrics} : (\text{Springsteen} \to \text{Springsteen} \to \text{Springsteen}) \cap \\ & \qquad\qquad (\text{Dion} \to \text{Dion} \to \text{Dion}) \\ & \texttt{motivationSong} : \text{Springsteen} \\ & \texttt{noSurrender} : \text{Springsteen} \\ & \texttt{ISurrender} : \text{Dion} \,\}.\end{aligned}$$

In this domain the type of combinator `addLyrics` forces the music and lyrics artist of its arguments to match.

The inhabitants of type Springsteen are

```
{motivationSong,
 noSurrender,
 addLyrics motivationSong noSurrender,
 addLyrics noSurrender motivationSong,
 addLyrics (addLyrics motivationSong noSurrender) motivationSong,
 addLyrics motivationSong (addLyrics motivationSong noSurrender), ...}.
```

The problematic use of advice from Celine Dion is avoided, but an infinite set of solutions is found. The output of `addLyrics` is fed back as input. Moreover, compared to the technical domain, the arguments of `addLyrics` are in the wrong order in some terms.

This section discusses why it is correct (sound and complete) to combine Γ_1 and Γ_2 into $\Gamma_3(c) = \Gamma_1(c) \cap \Gamma_2(c)$, for which inhabitation of LilyPond(Song) $\cap$ Springsteen exactly yields the correct answer

```
{addLyrics motivationSong noSurrender}.
```

□

Types in combined contexts need to be classified according to their provenance. The following definitions make this classification mathematically precise.

Definition 16 (Split Type Universes) A Boolean predicate inPartition : $\mathbb{T} \to \mathbb{B}$ is called proper classifier, if for all $A, B \in \mathbb{T}$ the following conditions are satisfied:

1. inPartition$(A \to B)$ iff inPartition(A) and inPartition(B)
2. inPartition$(A \cap B)$ iff inPartition(A) and inPartition(B)
3. inPartition(ω) = true

Two classifiers inPartition$_1$, inPartition$_2$: $\mathbb{T} \to \mathbb{B}$ are disjoint, if for all $A, B, C \in \mathbb{T}$:

1. If inPartition$_1(A)$, inPartition$_2(B)$, inPartition$_1(C)$, and $A \cap B \leq C$ then $A \leq C$
2. If inPartition$_2(A)$, inPartition$_1(B)$, inPartition$_2(C)$, and $A \cap B \leq C$ then $A \leq C$

A Split Type Universe is a pair of proper disjoint classifiers. □

Definition 17 (Split Context Pairs) Let (inPartition$_1$, inPartition$_2$) be a Split Type Universe. The pair of contexts $\Gamma_1, \Gamma_2 : \mathbf{B} \to \mathbb{T}$ is a Split Context Pair, if for all $c \in \mathbf{B}$:

$$\text{inPartition}_1(\Gamma_1(c)) \text{ and } \text{inPartition}_2(\Gamma_2(c))$$ □

Lemma 17 (Properties of Split Context Pairs) *For all Split Context Pairs* (Γ_1, Γ_2) *over a Split Type Universe* $(\text{inPartition}_1, \text{inPartition}_2)$*, and* $A, B \in \mathbb{T}$*,* $\Delta \in \mathbb{T}^*$*, and* $M, N \in \mathbb{A}$*, the following statements are true:*

1. $\text{inPartition}_1(\bigcap_{A_i \in \Delta} A_i)$ *iff for all* A *in* Δ*:* $\text{inPartition}_1(A)$
2. $\text{inPartition}_2(\bigcap_{A_i \in \Delta} A_i)$ *iff for all* A *in* Δ*:* $\text{inPartition}_2(A)$
3. $\text{inPartition}_1(\text{minimalType}(\Gamma_1, M))$
4. $\text{inPartition}_2(\text{minimalType}(\Gamma_2, M))$
5. *There exist* $\Delta_1, \Delta_2 \in \mathbb{T}^*$*, s.t.*
   $\text{minimalType}(\Gamma_1, M) = \bigcap_{A_i \in \Delta_1} A_i$ *and* $\text{minimalType}(\Gamma_2, M) = \bigcap_{A_i \in \Delta_2} A_i$ *and*
   for $\Gamma(c) = \Gamma_1(c) \cap \Gamma_2(c)$*:* $\text{minimalType}(\Gamma, M) = \bigcap_{A_i \in \Delta_1 + \Delta_2} A_i$

PROOF Statements 1 and 2 are proven by induction on Δ. Statements 3 and 4 require induction on M, and in the case for applications $M = M'N$ additional induction on $\text{minimalType}(\Gamma_i, M')$ with index i selecting the context mentioned in the statement to show. The last statement is shown by induction on M and Δ_1, Δ_2 are $[::\Gamma_1(c)], [::\Gamma_2(c)]$ for $M = c$ or the type lists Δ used in the minimalType algorithm for the case of $M = M'N$. In the latter case a detailed analysis of subtype machine behavior is required. ■

Theorem 4 (Split Context Pair Correctness) *For all Split Context Pairs* (Γ_1, Γ_2) *over a Split Type Universe* $(\text{inPartition}_1, \text{inPartition}_2)$*, and* $A, B \in \mathbb{T}$*,* $M \in \mathbb{A}$*, and* $\Gamma(c) = \Gamma_1(c) \cap \Gamma_2(c)$*:*

$$\begin{array}{l} \textit{If } \text{inPartition}_1(A) \textit{ and } \text{inPartition}_2(B) \textit{ then} \\ \Gamma \vdash M : A \cap B \textit{ iff } \Gamma_1 \vdash M : A \textit{ and } \Gamma_2 \vdash M : B. \end{array}$$

PROOF Combining proofs $\Gamma_1 \vdash M : A$ and $\Gamma_2 \vdash M : B$ into $\Gamma \vdash M : A \cap B$ is possible because of Lemma 11 rule $(\cap)$ and the context weakening property from Lemma 12. The other direction follows from the minimality and soundness in Lemma 16, Lemma 17.5, disjointness of universe classifiers, as well as Lemma 17.3 and Lemma 17.4. ■

In fact, any context pair (Γ_1, Γ_2) can be transformed into a Split Context Pair. The transformation requires a theory of intersection homomorphisms and their effect as context transformations in FCL.

Definition 18 (Intersection Type Homomorphism) Let $\mathbf{C}_1, \mathbf{C}_2$ be preordered countable sets of constructor symbols and $\mathbb{T}_1, \mathbb{T}_2$ types formed over each of these sets. Function $f : \mathbb{T}_1 \to \mathbb{T}_2$ is an Intersection Type Homomorphism if it satisfies the following conditions for arbitrary types $A, B \in \mathbb{T}_1$:

1. $A \leq B$ iff $f(A) \leq f(B)$

2. $f(A \to B) = f(A) \to f(B)$

3. $f(A \cap B) = f(A) \cap f(B)$

4. $f(\omega) = \omega$

5. If there exist $A_2, B_2 \in \mathbb{T}_2$, s.t. $f(A) = A_2 \to B_2$ then $A = A_1 \to B_1$ for some $A_1, B_1 \in \mathbb{T}_1$ □

Lemma 18 (Intersection Type Homomorphisms as FCL Context Transformations)
Let $f : \mathbb{T}_1 \to \mathbb{T}_2$ be an Intersection Type Homomorphism, then for all $\Delta \in \mathbb{T}_1^*$, $A \in \mathbb{T}_1$, $M \in \mathbb{A}$, and $\Gamma_1 : \mathbf{B} \to \mathbb{T}_1$, as well as $\Gamma_2(c) = f(\Gamma_1(c))$:

1. $f(\bigcap_{A_i \in \Delta} A_i) = \bigcap_{A_i \in \Delta} f(A_i)$

2. $\text{cast}_{\omega \to \omega \star \omega}(f(A)) = \text{map}(\lambda AB.(f(AB.1), f(AB.2)))(\text{cast}_{\omega \to \omega \star \omega}(A))$

3. $\text{minimalType}(\Gamma_2, M) = f(\text{minimalType}(\Gamma_1, M))$

4. $\Gamma_1 \vdash M : A$ *iff* $\Gamma_2 \vdash M : f(A)$

PROOF Statements 1 and 2 follow by induction on Δ and A and using the properties of f. Statement 3 is proven using induction on M and case analysis on the subtype machine behavior for the context constructed in $\text{minimalType}(\Gamma_2, M)$. Both directions of statement 4 require normalized induction on the derivations. The transformation from Γ_1 to Γ_2 only needs the properties of f, while the other direction requires previous statement. ■

The canonical way to combine two contexts is coproduct lifting, which is defined next.

Definition 19 (Coproduct Lifting) Let $\mathbf{C}_1, \mathbf{C}_2$ be preordered countable sets of constructor symbols and $\mathbb{T}_1, \mathbb{T}_2$ types formed over each of these sets. Additionally, set $\mathbf{C}_3 = \mathbb{B}$ is inserted. Set $\mathbf{C} = \bigoplus_{i \in \{1,2,3\}} \mathbf{C}_i$ is ordered by the least preorder closed under the following rules:

$$\frac{c \leq_{\mathbf{C}_1} d}{(1, c) \leq_{\mathbf{C}} (1, d)} \quad \frac{c \leq_{\mathbf{C}_2} d}{(2, c) \leq_{\mathbf{C}} (2, d)} \quad \frac{}{(3, b) \leq_{\mathbf{C}} (3, b)}$$

Coproduct lifting into types $\mathbb{T}$ formed over $\mathbf{C}$ is perfomed using:

$$
\begin{aligned}
&\mathrm{lift}_1(A) = \mathrm{lift}(\mathrm{liftCtor}_1, \mathrm{true}, A)\\
&\mathrm{lift}_2(A) = \mathrm{lift}(\mathrm{liftCtor}_2, \mathrm{false}, A)\\
&\mathrm{liftCtor}_1(A) = (1, A)\\
&\mathrm{liftCtor}_2(A) = (2, A)\\
&\mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A) =\\
&\begin{cases}
\omega & \text{for } A = \omega\\
\mathrm{liftCtor}(c)(\mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, B)) & \text{for } A = c(B)\\
\mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_1) \to \mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_2) & \text{for } A = A_1 \to A_2\\
(3, \mathrm{isLeft})(\mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_1) \star \mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_2)) & \text{for } A = A_1 \star A_2\\
\mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_1) \cap \mathrm{lift}(\mathrm{liftCtor}, \mathrm{isLeft}, A_2) & \text{for } A = A_1 \cap A_2
\end{cases}
\end{aligned}
$$

Coproduct lifting has partially defined right inverses:

$$
\begin{aligned}
&\mathrm{unlift}(\mathrm{unliftCtor}, A) =\\
&\begin{cases}
\omega & \text{for } A = \omega\\
A'_1 \star A'_2 & \text{for } A = (3, b)(A_1, A_2) \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_1) = A'_1 \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_2) = A'_2\\
c'(B') & \text{for } A = (i, c)(B) \text{ and } i \in \{1, 2\} \text{ and}\\
& \quad \mathrm{unliftCtor}((i, c)) = c' \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, B) = B'\\
A'_1 \to A'_2 & \text{for } A = A_1 \to A_2 \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_1) = A'_1 \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_2) = A'_2\\
A'_1 \cap A'_2 & \text{for } A = A_1 \cap A_2 \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_1) = A'_1 \text{ and}\\
& \quad \mathrm{unlift}(\mathrm{unliftCtor}, A_2) = A'_2
\end{cases}\\
&\mathrm{unliftCtor}_1(c) = c' \text{for } c = (1, c')\\
&\mathrm{unliftCtor}_2(c) = c' \text{for } c = (2, c')\\
&\mathrm{unlift}_1(A) = \mathrm{unlift}(\mathrm{unliftCtor}_1, A)\\
&\mathrm{unlift}_2(A) = \mathrm{unlift}(\mathrm{unliftCtor}_2, A)
\end{aligned}
$$

There is an easy to define pair of $\mathbb{T}$-classifiers inPartition_j for $j \in \{1,2\}$:

$$\text{inPartition}_j(A) = \begin{cases} \text{true} & \text{for } A = \omega \\ \text{true} & \text{for } A = (i,c)(B) \text{ and } i = j \text{ and } \text{inPartition}_j(B) \\ \text{true} & \text{for } A = (3,\text{true})(A_1 \star A_2) \text{ and } j = 1 \text{ and } \text{inPartition}_j(A_1) \text{ and } \text{inPartition}_j(A_2) \\ \text{true} & \text{for } A = A_1 \to A_2 \text{ and } \text{inPartition}_j(A_1) \text{ and } \text{inPartition}_j(A_2) \\ \text{true} & \text{for } A = A_1 \cap A_2 \text{ and } \text{inPartition}_j(A_1) \text{ and } \text{inPartition}_j(A_2) \\ \text{false} & \text{otherwise} \end{cases}$$

□

The lifting is mostly straight-forward, except for the subtle encoding of products. The latter is necessary to prevent information leakage between type universes from nested products. Type $\omega \star \omega$ illustrates this point: it contains information because $\omega \leq \omega \star \omega$ is false, but without the product encoding it cannot be uniquely classified as belonging to any type universe.

Lemma 19 (Properties of Coproduct Lifting) *Again let* $\mathbf{C}_1, \mathbf{C}_2$ *be preordered countable sets of constructor symbols,* $\mathbb{T}_1, \mathbb{T}_2$ *types formed over each of these sets and* $\mathbb{T}$ *be formed over* $\mathbf{C}$ *as constructed in Definition 19. For all* $k \in \{1,2\}$, $(c_k, d_k \in \mathbf{C}_k)$, *and* $A_k, A_k^1, A_k^2 \in \mathbb{T}_k$, $\Delta_k \in \mathbb{T}_k^*$, *and* $B, B_1, B_2 \in \mathbb{T}$, $c \in \mathbf{C}$, $b_1 = \textit{true}$, $b_2 = \textit{false}$, *coproduct lifting has the following properties:*

1. $\text{lift}_k(A_k^1 \to A_k^2) = \text{lift}_k(A_k^1) \to \text{lift}_k(A_k^2)$
2. $\text{lift}_k(A_k^1 \cap A_k^2) = \text{lift}_k(A_k^1) \cap \text{lift}_k(A_k^2)$
3. *If there exists* A_1', A_2', *s.t.* $\text{lift}_k(A_k) = A_1' \to A_2'$ *then there exists* $A^1, A^2 \in \mathbb{T}_k$, *s.t.* $A_k = A^1 \to A^2$
4. $\text{lift}_k(\omega) = \omega$
5. $\text{unlift}_k(\text{lift}_k(A_k)) = A_k$
6. $\text{lift}_k(\bigcap_{A_i \in \Delta_k} A_i) = \bigcap_{A_i \in \text{map}(\lambda A.\text{lift}_k(A))(\Delta_k)} A_i$
7. $\text{cast}_{\text{lift}_k(c_k(B_k))}(\text{lift}_k(A_k)) = \text{map}(\lambda A.\text{lift}_k(A))(\text{cast}_{c_k(B_k)}(A_k))$
8. $\text{isOmega}(A_k) = \text{isOmega}(\text{lift}_k(A_k))$
9. $\text{cast}_{\text{lift}_k(A_k^1 \to A_k^2)}(\text{lift}_k(A_k)) = \text{map}(\lambda AB.(\text{lift}_k(AB.1), \text{lift}_k(AB.2)))(\text{cast}_{A_k^1 \to A_k^2}(A_k))$
10. $\text{cast}_{\text{lift}_k(A_k^1 \star A_k^2)}(\text{lift}_k(A_k)) = \text{map}(\lambda AB.(\text{lift}_k(AB.1), \text{lift}_k(AB.2)))(\text{cast}_{A_k^1 \star A_k^2}(A_k))$
11. *If* $B = \text{lift}_k(A_k^1) \star \text{lift}_k(A_k^2)$ *and* $\Delta = \text{cast}_{(3,b_k)(B)}(\text{lift}_k(A_k))$ *then* $\text{cast}_B(\bigcap_{A_i \in \Delta} A_i) = \text{map}(\lambda AB.(\text{lift}_k(AB.1), \text{lift}_k(AB.2)))(\text{cast}_{A_k^1 \star A_k^2} A_k)$

12. $A_k^1 \leq A_k^2$ *iff* $\text{lift}_k(A_k^1) \leq \text{lift}_k(A_k^2)$
13. $\text{inPartition}_k(\text{lift}_k(A_k)) = \textit{true}$
14. $\text{inPartition}_k(B_1 \to B_2) = \text{inPartition}_k(B_1)$ *and* $\text{inPartition}_k(B_2)$
15. $\text{inPartition}_k(B_1 \cap B_2) = \text{inPartition}_k(B_1)$ *and* $\text{inPartition}_k(B_2)$
16. $\text{inPartition}_k(\omega) = \textit{true}$
17. *If* $\text{inPartition}_1(B_1)$ *and* $\text{inPartition}_2(c(B_2))$ *then* $\text{cast}_{c(B_2)}(B_1) = [::]$
    If $\text{inPartition}_2(B_1)$ *and* $\text{inPartition}_1(c(B_2))$ *then* $\text{cast}_{c(B_2)}(B_1) = [::]$
18. *If* $\text{inPartition}_1(B_1)$ *and* $\text{inPartition}_2(B_2)$ *and* $B_1 \leq B_2$ *then* $\text{isOmega}(B_2)$
    If $\text{inPartition}_2(B_1)$ *and* $\text{inPartition}_1(B_2)$ *and* $B_1 \leq B_2$ *then* $\text{isOmega}(B_2)$
19. *If* $\text{inPartition}_k(B)$ *then for all* B' *in* $\text{primeFactors}(B)$*:* $\text{inPartition}_k(B')$
20. *If* $\text{inPartition}_1(B_1)$ *and* $\text{inPartition}_2(B_2)$ *and* $\text{inPartition}_1(B)$ *and* $B_1 \cap B_2 \leq B$ *then* $B_1 \leq B$
    If $\text{inPartition}_1(B_1)$ *and* $\text{inPartition}_2(B_2)$ *and* $\text{inPartition}_2(B)$ *and* $B_1 \cap B_2 \leq B$ *then* $B_2 \leq B$

PROOF The statements are proven in the order presented. Most follow from simple induction, previous lemmas and unfolding of definitions. Statements 12 and 16 require analyzing the operational semantics of the subtype machine and induction on the termination certificates for [`subty` A_k^1 `of` A_k^2] and [`subty` B_1 `of` B_2]. This is necessary because of transitivity and the change of arrow variance. For similar reasons, statement 19 is proven by induction in the size of B. Statement 20 is proven using statement 19 and correctness of prime factorization (Theorem 2). ■

Theorem 5 (Coproduct Lifting Context Pairs to Split Context Pairs) *Any pair of contexts* $\Gamma_1 : \mathbf{B} \to \mathbb{T}_1$ *and* $\Gamma_2 : \mathbf{B} \to \mathbb{T}_2$ *can be lifted into* $\Gamma_1'(c) = \text{lift}_1(\Gamma_1(c))$ *and* $\Gamma_2'(c) = \text{lift}_2(\Gamma_2(c))$ *and combined to* $\Gamma(c) = \Gamma_1'(c) \cap \Gamma_2'(c)$ *such that for all* $A \in \mathbb{T}_1$, $B \in \mathbb{T}_2$, *and* $M \in \mathbb{A}$*:*

$$\Gamma \vdash M : \text{lift}_1(A) \cap \text{lift}_2(B) \textit{ iff } \Gamma_1 \vdash M : A \textit{ and } \Gamma_2 \vdash M : B.$$

PROOF By Lemma 19, inPartition_1 and inPartition_2 are proper disjoint classifiers and (Γ_1', Γ_2') is a Split Context Pair. Also by Lemma 19, lift_1 and lift_2 are Intersection Type Homomorphisms. Hence type judgments of the lifted contexts can be transformed back and forth between the original and lifted contexts by Lemma 18. ■

2.5 Verified Enumerative Type Inhabitation in FCL

This section discusses an algorithm for enumerative type inhabitation (enumerate all inhabitants of a given type) in FCL. Rehof and Urzyczyn [165] have studied the decision problem for type inhabitation in FCL and characterized it as EXPTIME-complete. The decision problem becomes undecidable if the system is extended with schematism and $(1+k)$-EXPTIME-complete[1] if the schematism is restricted to variable instantiations of maximal depth k [57]. The decision problem (does there exist an inhabitant of a given type?) is informative for complexity theory and [165] also provides decision procedures for the problems of ambiguity (do there exist multiple inhabitants of a given type?) and infinity (do there exist infinitely many inhabitants of a given type?). Decidability and the alternating tree-automaton based representation of FCL without subtyping [165] inform the design of an enumerative type inhabitation algorithm, which can construct inhabitants instead of just answering questions about their existence. The enumerative approach is most valuable from the practical perspective of software synthesis and has been at the heart of multiple implementations of the (CL)S Framework [56; 59; 55; 20; 61]. Yet, it has never been formalized – a gap that has existed for almost 10 years of development and is finally closed in this section.

2.5.1 The Cover Machine

Combinators are the building blocks of FCL and the most fundamental task in any type inhabitation algorithm is: given a combinator and a target type, find (all) possible types of arguments, such that application of the combinator produces (covers) the desired target. The cover machine solves this goal. For its specification some auxiliary functions are necessary.

Definition 20 (Auxiliary functions for the cover machine) Subtype duplicate free intersection is defined by:

$$A \pitchfork B = \begin{cases} A & \text{for } A \leq B \\ B & \text{for not } A \leq B \text{ and } B \leq A \\ A \cap B & \text{otherwise} \end{cases}$$

Function partitionCover partitions a list of types according to membership in another list:

$$\text{partitionCover}(\Delta_1, \Delta_2) = \begin{cases} ([::], [::]) & \text{for } \Delta_2 = [::] \\ ([::\ A\&\Delta_2^1], \Delta_2^2) & \text{for } \Delta_2 = [::\ A\&\Delta] \text{ and } A \text{ in } \Delta_1 \text{ and} \\ & \quad \text{partitionCover}(\Delta_1, \Delta) = (\Delta_2^1, \Delta_2^2) \\ (\Delta_2^1, [::\ A\&\Delta_2^2]) & \text{for } \Delta_2 = [::\ A\&\Delta] \text{ and not } A \text{ in } \Delta_1 \text{ and} \\ & \quad \text{partitionCover}(\Delta_1, \Delta) = (\Delta_2^1, \Delta_2^2) \end{cases}$$

[1] The definition of depth(A) in this text coincides with level$(A)+1$ in [57].

Predicate stillPossible : $(((\mathbb{T}^* \times \mathbb{T}) \times \mathbb{T}^*)^* \times \mathbb{T}^*) \to \mathbb{B}$ tests for a list of multi arrows with target components they cover and a list of target components, which are yet to cover, if these still can be covered:

$$\begin{aligned}\text{stillPossible}(\text{splits}, \text{toCover}) = \ & \text{for all } A \text{ in toCover}: \\ & \quad \text{exists covered} \in \text{map}(\lambda \text{mCovered.mCovered.2})(\text{splits}): \\ & \qquad A \in \text{covered}\end{aligned}$$

Function mergeMultiArrow distributes duplication free intersection over two multi arrows:

$$\text{mergeMultiArrow}(m_1, m_2) = (\text{map}(\lambda \text{srcs.srcs.1} \pitchfork \text{srcs.2})(\text{zip}(m_1.1, m_2.1)), m_1.2 \pitchfork m_2.2)$$

Definition 21 (Cover Machine) The instructions of the cover machine are given by

$$\begin{aligned}\mathbb{I}_{\text{Cover}} \ni i ::= \ & \texttt{Cover} \text{ splits toCover} \\ \mid \ & \texttt{CheckCover} \text{ splits toCover} \\ \mid \ & \texttt{ContinueCover} \text{ splits toCover currentResult} \\ \mid \ & \texttt{CheckContinueCover} \text{ splits toCover currentResult}\end{aligned}$$

where

- splits $\in ((\mathbb{T}^* \times \mathbb{T}) \times \mathbb{T}^*)^*$ is a list of multi arrows with parts of the target they cover
- toCover $\in \mathbb{T}^*$ is a list of target components still to cover
- currentResult $\in \mathbb{T}^* \times \mathbb{T}$ is an intermediate multi arrow, possibly added to the result

The output $\mathbb{O}_{\text{Cover}} = (\mathbb{T}^* \times \mathbb{T})^*$ is a list of multi arrows.

Behavior of the cover machine is specified by the following step relation:

$$\frac{\text{not stillPossible}(\text{splits}, \text{toCover})}{(s, [::\ \texttt{CheckCover} \text{ splits toCover}\&p]) \rightsquigarrow (s, p)} \ (\text{STEP}_{\text{CHECKPRUNE}})$$

$$\frac{\text{not stillPossible}(\text{splits}, \text{toCover})}{(s, [::\ \texttt{CheckContinueCover} \text{ splits toCover currentResult}\&p]) \rightsquigarrow (s, p)} \ (\text{STEP}_{\text{CHECKCONTINUEPRUNE}})$$

$$\frac{\text{stillPossible}(\text{splits}, \text{toCover})}{(s, [::\ \texttt{CheckCover} \text{ splits toCover}\&p]) \rightsquigarrow (s, [::\ \texttt{Cover} \text{ splits toCover}\&p])} \ (\text{STEP}_{\text{CHECKOK}})$$

$$\frac{\text{stillPossible}(\text{splits}, \text{toCover})}{\begin{aligned}&(s, [::\ \texttt{CheckContinueCover} \text{ splits toCover currentResult}\&p]) \rightsquigarrow \\ &(s, [::\ \texttt{ContinueCover} \text{ splits toCover currentResult}\&p])\end{aligned}} \ (\text{STEP}_{\text{CHECKCONTINUEOK}})$$

$$\frac{}{(s, [::\ \texttt{Cover}\ [::]\ \text{toCover}\&p]) \rightsquigarrow (s, p)} \ (\text{STEP}_{\text{DONE}})$$

$$\frac{}{(s, [::\ \texttt{ContinueCover}\ [::]\ \text{toCover currentResult}\&p]) \rightsquigarrow (s, p)}\ (\text{STEP}_{\text{DONECONTINUE}})$$

$$\frac{\text{partitionCover}(\text{covered}, \text{toCover}) = ([::], \Delta)}{(s, [::\ \texttt{Cover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover}\&p]) \rightsquigarrow (s, [::\ \texttt{Cover}\ \text{splits toCover}\&p])}\ (\text{STEP}_{\text{SKIP}})$$

$$\frac{\text{partitionCover}(\text{covered}, \text{toCover}) = ([::], \Delta)}{\begin{array}{l}(s, [::\ \texttt{ContinueCover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover currentResult}\&p]) \rightsquigarrow \\ (s, [::\ \texttt{ContinueCover}\ \text{splits toCover currentResult}\&p])\end{array}}\ (\text{STEP}_{\text{SKIPCONTINUE}})$$

$$\frac{\text{partitionCover}(\text{covered}, \text{toCover}) = ([::\ A\&\Delta], [::])}{\begin{array}{l}(s, [::\ \texttt{Cover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover}\&p]) \rightsquigarrow \\ ([::\ m\&s], [::\ \texttt{CheckCover}\ \text{splits toCover}\&p])\end{array}}\ (\text{STEP}_{\text{ADDDONE}})$$

$$\frac{\text{partitionCover}(\text{covered}, \text{toCover}) = ([::\ A\&\Delta], [::])}{\begin{array}{l}(s, [::\ \texttt{ContinueCover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover currentResult}\&p]) \rightsquigarrow \\ ([::\ \text{mergeMultiArrow}(\text{currentResult}, m)\&s], \\ \ [::\ \texttt{CheckContinueCover}\ \text{splits toCover currentResult}\&p])\end{array}}\ (\text{STEP}_{\text{MERGEDONE}})$$

$$\frac{\text{partitionCover}(\text{covered}, \text{toCover}) = ([::\ A\&\Delta_1], [::\ B\&\Delta_2])}{\begin{array}{l}(s, [::\ \texttt{Cover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover}\&p]) \rightsquigarrow \\ (s, [::\ \texttt{ContinueCover}\ \text{splits}\ [::\ B\&\Delta_2]\ m\&[::\ \texttt{CheckCover}\ \text{splits toCover}\&p]])\end{array}}\ (\text{STEP}_{\text{CONTINUE}})$$

$$\frac{\begin{array}{l}\text{partitionCover}(\text{covered}, \text{toCover}) = ([::\ A\&\Delta_1], [::\ B\&\Delta_2]) \\ \text{mergeMultiArrow}(\text{currentResult}, m).1 = \text{currentResult}.1\end{array}}{\begin{array}{l}(s, [::\ \texttt{ContinueCover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover currentResult}\&p]) \rightsquigarrow \\ (s, [::\ \texttt{ContinueCover}\ \text{splits}\ [::\ B\&\Delta_2]\ \text{mergeMultiArrow}(\text{currentResult}, m)\&p])\end{array}}\ (\text{STEP}_{\text{CONTINUE-MERGEALWAYS}})$$

$$\frac{\begin{array}{l}\text{partitionCover}(\text{covered}, \text{toCover}) = ([::\ A\&\Delta_1], [::\ B\&\Delta_2]) \\ \text{not mergeMultiArrow}(\text{currentResult}, m).1 = \text{currentResult}.1\end{array}}{\begin{array}{l}(s, [::\ \texttt{ContinueCover}\ [::\ (m, \text{covered})\&\text{splits}]\ \text{toCover currentResult}\&p]) \rightsquigarrow \\ (s, [::\ \texttt{ContinueCover}\ \text{splits}\ [::\ B\&\Delta_2]\ \text{mergeMultiArrow}(\text{currentResult}, m)\& \\ \quad [::\ \texttt{CheckContinueCover}\ \text{splits toCover currentResult}\&p]])\end{array}}\ (\text{STEP}_{\text{CONTINUE-MERGE-OPTIONS}})$$

The n-step closure of $\rightsquigarrow$ is the least relation closed under the rules:

$$\frac{}{(s, p) \rightsquigarrow_0 (s, p)} \qquad \frac{\boxed{1}(s_1, p_1) \rightsquigarrow (s_2, p_2) \qquad \boxed{2}(s_2, p_2) \rightsquigarrow_n (s_3, p_3)}{(s_1, p_1) \rightsquigarrow_{n+1} (s_3, p_3)}$$

By definition the n-step closure coincides with transtive reflexive closure: $(s, p) \rightsquigarrow_* (s', p')$ iff there exists n, s.t. $(s, p) \rightsquigarrow_n (s', p')$. In proof trees t for $(s, p) \rightsquigarrow_{n+1} (s', p')$ the first and second premise are selected using $t.1$ and $t.2$. □

The cover machine processes a stack of instructions. Results are pushed onto an output stack. Each of the instructions has lists splits, and toCover. For the initial instructions toCover will be chosen as the prime factors of the target type. List splits will include multi arrows computed from the type of a combinator. Each of these multi arrows will have the same number of arguments and be paired with the target prime factors that are greater or equal than the multi arrow target. For example, if a combinator x has type $\Gamma(x) = (a \to b \to c \cap d) \cap (d \to e) \cap (f \to g \to h)$ and the target type is $h \cap c$, then one of the initial instructions will be `CheckCover` $[::(([::b\&[::a]], c \cap d), [::c])\&[::(([::g\&[::f]], h), [::h])]]$ $[::h\&[::c]]$. This instruction checks the multi arrows with two targets, which are $([::b\&[::a]], c \cap d)$ for $a \to b \to c \cap d$ and $([::g\&[::f]], h)$ for $f \to g \to h$. The target prime factors are c and h, so the first multi arrow is only paired with c, because $c \cap d \leq c$ and not $c \cap d \leq h$. The second multi arrow is paired with h because $h \leq h$ and not $h \leq c$. Two of the four cover machine instructions are prefixed with CHECK. They try to prune its exponentially large search space. If at some point all of the covered components in splits are no longer enough to cover all components in toCover, the search is aborted and the instruction can be dropped. This shortcut, or if not viable the resumption of normal operations, is implemented by the first four rules ($\text{STEP}_{\text{CHECKPRUNE}}$), ($\text{STEP}_{\text{CHECKCONTINUEPRUNE}}$), ($\text{STEP}_{\text{CHECKOK}}$), and ($\text{STEP}_{\text{CHECKCONTINUEOK}}$). Rules ($\text{STEP}_{\text{DONE}}$) and ($\text{STEP}_{\text{DONECONTINUE}}$) implement failure modes and skip the current instruction. They are inserted to ensure left totality of the step relation and invariants established for proper initialization will later prevent, that their precondition (empty choices for splits) ever occurs. If the first multi arrow in splits does not cover any new elements of toCover, it is removed by ($\text{STEP}_{\text{SKIP}}$) or ($\text{STEP}_{\text{DONECONTINUE}}$). The opposite condition, if the first multi arrow covers all remaining targets, is implemented in rules ($\text{STEP}_{\text{ADDDONE}}$) and ($\text{STEP}_{\text{MERGEDONE}}$). In each case the multi arrow from splits is pushed to the output stack. In ($\text{STEP}_{\text{MERGEDONE}}$) it is also merged with the current result. When merging two multi arrows, all of their targets and all of their sources are intersected, avoiding redundancy whenever possible: $\text{mergeMultiArrow}(([::b\&[::a]], c), ([::d\&[::a \cap e]], f)) = ([::b \cap d\&[::a \cap e]], c \cap f)$ where redundancy in the second argument has been removed, i.e. $a \cap e$ instead of $a \cap (a \cap e)$. The DONE-rules continue with the rest of the multi arrows in splits. Intuitively in $(a \to c) \cap (b \to c)$, type $a \to c$ is enough to cover target c, but $b \to c$ is also viable, and recursive targets a and b can have different inhabitants. Possibilities to cover the target are lost when resuming with the rest of splits, so prune checks are scheduled. The CONTINUE-rules are used for the remaining cases, where the first multi arrow in splits covers some, but not all, of the targets in toCover. This multi arrow is then included in the current result, either by copying it if there was no prior result (rule ($\text{STEP}_{\text{CONTINUE}}$)) or by merging it with the previous result. In rule ($\text{STEP}_{\text{CONTINUE}}$) merging with the previous result does not change any of the sources of the current result. Unlike the other CONTINUE-rules and the DONE-rules, which need to consider inclusion and exclusion of the first multi arrow, the multi arrow can always be included in this case. For example in $(a \to b) \cap (a \to c) \cap (d \to e \cap c)$ with target $b \cap c \cap e$ choosing $a \to c$ after $b \to c$ does not change any sources $((\text{mergeMultiArrow}(([::a], b), ([::a], c))).1 = ([::a], b).1)$. Unchanged sources imply that no inhabitants are ruled out for recursive targets compared to the scenario without that particular choice. The choice is always safe, again avoiding exploration of an entire search

space area.

The first fact to establish about the specification in Definition 21 is functionality (c.f. Lemma 1).

Lemma 20 (Functionality) *For all $n \in \mathbb{N}$, and $s, s_1, s_2 \in \mathbb{O}_{\text{Cover}}$ and $p, p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$ the cover machine step relation is functional:*

1. *If $(s, p) \rightsquigarrow (s_1, p_1)$ and $(s, p) \rightsquigarrow (s_2, p_2)$ then $(s_1, p_1) = (s_2, p_2)$*
2. *If $(s, p) \rightsquigarrow_n (s_1, p_1)$ and $(s, p) \rightsquigarrow_n (s_2, p_2)$ then $(s_1, p_1) = (s_2, p_2)$*

PROOF The precondition of all rules in Definition 21 are mutually exclusive. ■

Definition 22 (Cover Machine Step Function) The result (s', p') of steps $(s, p) \rightsquigarrow_n (s', p')$ for $n \leq 1$ is computed by:

$$\text{step}(s, p) : \{(s', p') \in (\mathbb{O}_{\text{Cover}} \times \mathbb{I}^*_{\text{Cover}}) \mid \text{ if } p = [::] \text{ then } (s', p') = (s, p) \text{ else } (s, p) \rightsquigarrow (s', p')\}$$

$$\text{step}(s, p) =$$

$$\begin{cases} (s, [::]) & \text{for } p = [::] \\ (s', p') & \text{for } \dfrac{P(s, p)}{(s, p) \rightsquigarrow (s', p')} \text{ in the rules of } \rightsquigarrow \text{ and } P(s, p) = \text{true} \end{cases}$$

Similar to the implementation of the subtype machine, totality of the implementation of the cover machine is guaranteed by a termination certificate.

Definition 23 (Cover Machine Termination Certificate) For any input $(s, p) \in \mathbb{O}_{\text{Cover}} \times \mathbb{I}^*_{\text{Cover}}$ termination certificate $\text{Dom}(s, p)$ is a proof tree constructed from the rules of $(s, p) \rightsquigarrow_n (s', [::])$ for some $n \in \mathbb{N}$ and $s' \in \mathbb{O}_{\text{Cover}}$. □

A tail recursive procedure now implements the full n-step behavior of the cover machine.

Definition 24 (Procedure implementing the Cover Machine)

$$\text{coverMachine}(s, p) : \text{Dom}(s, p) \to \{s' \in \mathbb{O}_{\text{Cover}} \mid (s, p) \rightsquigarrow_* (s', [::])\}$$

$$\text{coverMachine}(s, p)(d) =$$

$$\begin{cases} s' & \text{for step}(s, p) = (s', [::]) \\ \text{coverMachine}(s', [:: i\&p'])(d.2) & \text{for step}(s, p) = (s, [:: i\&p']) \end{cases}$$

□

It remains to show that the cover machine is total, i.e. that a termination certificate can be provided for all inputs.

Lemma 21 (Totality of the Cover Machine) *For $i \in \mathbb{I}_{\text{Cover}}$ define:*

$$\text{splitsOf}(i) = \begin{cases} \text{splits} & \textit{for } i = \texttt{Cover} \text{ splits toCover} \\ & \textit{or } i = \texttt{CheckCover} \text{ splits toCover} \\ & \textit{or } i = \texttt{ContinueCover} \text{ splits toCover currentResult} \\ & \textit{or } i = \texttt{CheckCover} \text{ splits toCover currentResult} \end{cases}$$

$$\text{isChecked}(i) = \begin{cases} \textit{true} & \textit{for } i = \texttt{Cover} \text{ splits toCover} \\ & \textit{or } i = \texttt{ContinueCover} \text{ splits toCover currentResult} \\ \textit{false} & \textit{otherwise} \end{cases}$$

$$\text{measure}(i) = \begin{cases} 2^{2 \cdot \text{lsize}(\text{splitsOf}(i))} & \textit{for } \text{isChecked}(i) = \textit{true} \\ 2^{2 \cdot \text{lsize}(\text{splitsOf}(i))} + 1 & \textit{otherwise} \end{cases}$$

For all $n \in \mathbb{N}$, $s_1, s_2 \in \mathbb{O}_{\text{Cover}}$ and $p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$ the following statements are true:

1. *If $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ then $\sum_{i \text{ in } p_1} \text{measure(i)} < \sum_{i \text{ in } p_2} \text{measure(i)}$*
2. *If $(s_1, p_1) \rightsquigarrow_n (s_2, p_2)$ then $n \leq \sum_{i \text{ in } p_1} \text{measure(i)}$*
3. *Proof tree $\text{Dom}(s_1, p_1)$ exists*

PROOF The first statement is proven by case analysis on the possibilities for $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ and application of some simple arithmetic. The second statement then follows by induction on n, the first statement, and transitivity of $\leq$ on natural numbers. Finally the proof tree for $\text{Dom}(s_1, p_1)$ is constructed by iterating the single step function up to $\sum_{i \text{ in } p_1} \text{measure(i)}$ times. The second statement proves that more iterations cannot result in additional steps and thereby the program instruction stack p must be empty as is required for the leaves of the Dom proof tree. ■

Correctness of the cover machine is a multi faceted issue and its proof requires to study multiple properties and invariants of the machine behavior first.

Lemma 22 (Stack Behavior) *For all $s_1, s_2 \in \mathbb{O}_{\text{Cover}}$, and $p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$ the cover machine satisfies the following stack invariants for its state and list of instructions*

1. *If $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ then $\text{suffix}(\text{behead}(p1), p2)$*
2. *If $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ then $\text{suffix}(s1, s2)$*
3. *If $(s_1, p_1) \rightsquigarrow_* (s_2, p_2)$ then $\text{suffix}(s1, s2)$*

PROOF Case analysis and in the last case induction on the steps. ■

More auxiliary functions for multi arrows are used in the subsequent specifications.

Definition 25 (Auxiliary Functions for Multi Arrows) Function mergeMultiArrows merges sequences of multi arrows:

$$\text{mergeMultiArrows}(ms) = \begin{cases} ([::], \omega) & \text{for } ms = [::] \\ \text{foldl}(\lambda m_1 m_2.\,\text{mergeMultiArrow}(m_1, m_2))(m)(ms') & \text{for } ms = [::\, m \& ms'] \end{cases}$$

Function filterMergeMultiArrows flattens sequences of sequences of multi arrows by merging subsequences or removing them if they are empty:

$$\text{filterMergeMultiArrows}(mss) = \begin{cases} [::] & \text{for } mss = [::] \\ \text{filterMergeMultiArrows}(mss') & \text{for } mss = [::\, [::] \& mss'] \\ [::\, \text{mergeMultiArrows}([::\, m \& ms]) \& \\ \quad \text{filterMergeMultiArrows}(mss')] & \text{for } mss = [::\, [::\, m \& ms] \& mss'] \end{cases}$$

Components that might be subject to merging in a run of the cover machine are collected from its instructions using:

$$\text{mergeComponentsOf}(i) = \begin{cases} \text{map}(\lambda m.m.1)(\text{splits}) \\ \quad \text{for } i = \texttt{Cover} \text{ splits toCover or} \\ \quad\quad i = \texttt{CheckCover} \text{ splits toCover} \\ [::\, \text{currentResult} \& \text{map}(\lambda m.m.1)(\text{splits})] \\ \quad \text{for } i = \texttt{ContinueCover} \text{ splits toCover currentResult or} \\ \quad\quad i = \texttt{CheckContinueCover} \text{ splits toCover currentResult} \end{cases}$$

□

The first soundness property captures the fact that the cover machine does not add multi arrows to its results which were not present in its input instructions:

Lemma 23 (Cover Machine Input Merge Component Soundness)
For $mss_1, mss_2 \in ((\mathbb{T}^* \times \mathbb{T})^*)^*$, *and* $m_1, m_2 \in (\mathbb{T}^* \times \mathbb{T})$, $i \in \mathbb{I}_{\text{Cover}}$, *and* $p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$, *and* $s_1, s_2 \in \mathbb{O}_{\text{Cover}}$, *and* splits $\in ((\mathbb{T}^* \times \mathbb{T}) \times \mathbb{T}^*)^*$ *the following statements are true:*

1. $\text{filterMergeMultiArrows}(mss_1 +\!\!+ mss_2) =$
   $\text{filterMergeMultiArrows}(mss_1) +\!\!+ \text{filterMergeMultiArrows}(mss_2)$
2. *If* $mss_1 \sqsubseteq mss_2$ *then* $\text{filterMergeMultiArrows}(mss_1) \sqsubseteq \text{filterMergeMultiArrows}(mss_2)$

3. $\text{filterMergeMultiArrows}(\text{map}(\lambda ms.[::\text{mergeMultiArrow}(m_1, m_2)\&ms])(mss_1)) =$
   $\text{filterMergeMultiArrows}(\text{map}(\lambda ms.[::m_1\&[::m_2\&ms]])(mss_1))$

4. *If* $(s_1, [::i\&p_1]) \rightsquigarrow (s_2, p_2)$ *then*
   for all m *in* $\text{take}(\text{lsize}(s_2) - \text{lsize}(s_1), s_2)$:
   m *is in* $\text{filterMergeMultiArrows}(\text{subseqs}(\text{mergeComponentsOf}(i)))$

Define soundness as a Boolean predicate on $\mathbb{O}_{\text{Cover}} \times \mathbb{I}^*_{\text{Cover}}$ *by:*
$\text{sound}(s, p) =$ *for all* m *in* s :
m *is in*
$\text{flatten}(\text{map}(\lambda i.\text{filterMergeMultiArrows}(\text{subseqs}(\text{mergeComponentsOf}(i))))(p))$

5. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *then* $\text{sound}(\text{take}(\text{lsize}(s_2) - \text{lsize}(s_1), s_2), p_1)$

6. *If* $(s_1, [::i\&p_1]) \rightsquigarrow (s_2, p_2)$ *then*
   for all i_2 *in* $\text{take}(\text{lsize}(p_2) - \text{lsize}(p_1), p_2)$:
   if $\text{isChecked}(i)$ *then* $\text{splitsOf}(i_2) = \text{behead}(\text{splitsOf(i)})$

7. *If* $(s_1, [::i\&p_1]) \rightsquigarrow (s_2, p_2)$ *then*
   for all i_2 *in* $\text{take}(\text{lsize}(p_2) - \text{lsize}(p_1), p_2)$:
   if $i_2 =$ `ContinueCover` splits toCover currentResult or
   $i_2 =$ `CheckContinueCover` splits toCover currentResult then
   currentResult *is in*
   $\text{filterMergeMultiArrows}(\text{subseqs}(\text{mergeComponentsOf}(i)))$

8. *If* $(s_1, [::i\&p_1]) \rightsquigarrow (s_2, p_2)$ *and* $\text{sound}(\text{splits}, p_2)$ *then* $\text{sound}(\text{splits}, [::i_1\&p_1])$

9. *If* $(s_1, p_1) \rightsquigarrow_* (s_2, p_2)$ *then* $\text{sound}(\text{take}(\text{lsize}(s_2) - \text{lsize}(s_1), s_2), p_1)$

PROOF Statements 1, 2, and 3 are proven by induction on mss_1, mss_2, and mss_1. They are used to decompose properties of output states in multi step reductions. Statement 4 is the interesting part of single step soundness in statement 5. The idea is to show that freshly pushed outputs (those in $\text{take}(\text{lsize}(s_2) - \text{lsize}(s_1), s_2)$) are the result of merging multi arrows in the instructions before the step was performed. Lemma 22 facilitates this kind of reasoning by decomposing instructions and outputs. Statement 4 requires Lemma 22.2 followed by case analysis on i. Statement 5 is then proven by simple case analysis on p_1. Reductions with multiple steps require reasoning backwards: the output state is related to the input instructions. Statements 6, 7, and 8 perform backwards reasoning for single steps. They follow from the decomposition properties and case analysis on the first instruction i. Finally, statement 9 is proven by induction on the steps, decomposition of the first step with Lemma 22.2 and the later steps with Lemma 22.3, finishing by conjunction of the properties obtained by statement 5 for the first step with the properties obtained by statement 8 with the induction hypothesis for the last steps. ■

Completeness requires a more subtle and technically challenging argument. Statements and definitions of the following lemma will be illustrated in its proof and the puzzled reader is advised to read it first and then interactively step through the accompanying Coq formalization.

Lemma 24 (Cover Machine Input Merge Component Completeness)
Define

$$\text{toCoverOf}(i) = \begin{cases} \text{toCover} & \textit{for } i = \texttt{Cover} \text{ splits toCover or} \\ & \quad i = \texttt{CheckCover} \text{ splits toCover or} \\ & \quad i = \texttt{ContinueCover} \text{ splits toCover currentResult or} \\ & \quad i = \texttt{CheckContinueCover} \text{ splits toCover currentResult} \end{cases}$$

and some Boolean predicates to encode invariants:

- $\text{complete}(s, i) =$

  $\textit{for all } m_1 \textit{ in } \text{filterMergeMultiArrows}(\text{subseqs}(\text{mergeComponentsOf}(i))) :$

  $\textit{if } m_1.2 \leq \bigcap_{A_i \in \text{toCoverOf}(i)} A_i :$

  $\textit{exists } m_2 \textit{ in } s :$

  $\textit{for } (\text{srcs}, \text{tgt}) =$

  $$\begin{cases} ((\text{mergeMultiArrow}(\text{currentResult}, m_1)).1, \\ \quad \text{currentResult}.2 \cap \bigcap_{A_i \in \text{toCover}} A_i) \\ \quad \textit{for } i = \texttt{ContinueCover} \text{ splits toCover currentResult or} \\ \qquad i = \texttt{CheckContinueCover} \text{ splits toCover currentResult} \\ (m_1.1, \bigcap_{A_i \in \text{toCoverOf}(i)} A_i) \quad \textit{otherwise} \end{cases}$$

  $\text{lsize}(m_2.1) = \text{lsize}(\text{srcs}) \textit{ and}$

  $\textit{for all } (\text{src}_1, \text{src}_2) \textit{ in } \text{zip}(\text{srcs}, m_2.1) : \text{src}_1 \leq \text{src}_2 \textit{ and}$

  $m_2.2 \leq \text{tgt}$

- $\text{instruction\_covered}(i) =$

  $\textit{for all } ((\text{srcs}, \text{tgt}), \text{covered}) \textit{ in } \text{splitsOf}(i) :$

  $\text{tgt} \leq \bigcap_{A_i \in \text{covered}} A_i \textit{ and}$

  $\textit{for all } A \textit{ in } \text{toCoverOf}(i) : \textit{ If } \text{tgt} \leq A \textit{ then } A \textit{ is in } \text{covered}$

- $\text{not\_omega\_covered}(i) =$

  $\textit{not } \text{toCoverOf}(i) = [::] \textit{ and for all } A \textit{ in } \text{toCoverOf}(i) : \textit{not } \omega \leq A$

- $\text{arity\_equal}(i) =$

  $\textit{for all } (\text{srcs}_1, \text{tgt}_1) \textit{ in } \text{mergeComponentsOf}(i) :$

  $\textit{for all } (\text{srcs}_2, \text{tgt}_2) \textit{ in } \text{mergeComponentsOf}(i) : \text{lsize}(\text{srcs}_1) = \text{lsize}(\text{srcs}_2)$

- toCover_prime(i) = *for all* A *in* toCoverOf(i) : $A \in \mathbb{T}_\pi$
- currentResultNotDone(i) =

$$\begin{cases} \textit{If } A \textit{ in } \text{toCover } \textit{then not } \text{currentResult.2} \leq A \\ \qquad \textit{for } i = \texttt{ContinueCover} \text{ splits toCover currentResult or} \\ \qquad\quad i = \texttt{CheckContinueCover} \text{ splits toCover currentResult} \\ \textit{true} \qquad \textit{otherwise} \end{cases}$$

For $n \in \mathbb{N}$, $mss \in ((\mathbb{T}^* \times \mathbb{T})^*)^*$, $ms \in (\mathbb{T}^* \times \mathbb{T})^*$, $\text{srcs} \in \mathbb{T}^*$, *and* $A, B, \text{tgt} \in \mathbb{T}$, *and* $m, m_1, m_2 \in (\mathbb{T}^* \times \mathbb{T})$, $i \in \mathbb{I}_{\text{Cover}}$, *and* $p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$, *and* $s, s_1, s_2 \in \mathbb{O}_{\text{Cover}}$, $\text{splits} \in ((\mathbb{T}^* \times \mathbb{T}) \times \mathbb{T}^*)^*$, *and* $\text{covered}, \text{toCover} \in \mathbb{T}^*$ *the following statements are true:*

1. *for all* A *in* (partitionCover(covered, toCover)).1*:* A *is in* covered
2. *for all* A *in* (partitionCover(covered, toCover)).2*:* A *is not in* covered
3. (partitionCover(covered, toCover)).1 $\sqsubseteq$ toCover
4. (partitionCover(covered, toCover)).2 $\sqsubseteq$ toCover
5. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and for all* i_1 *in* p_1*:* instruction_covered(i_1) *then* *for all* i_2 *in* p_2*:* instruction_covered(i_2)
6. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and for all* i_1 *in* p_1*:* not_omega_instruction(i_1) *then* *for all* i_2 *in* p_2*:* not_omega_instruction(i_2)
7. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and for all* i_1 *in* p_1*:* arity_equal(i_1) *then* *for all* i_2 *in* p_2*:* arity_equal(i_2)
8. *If for all* m_1 *in* ms*: for all* m_2 *in* ms*:* $\text{lsize}(m_1.1) = \text{lsize}(m_2).1$ *then* *for all* m *in* ms*:* $\text{lsize}(m.1) = \text{lsize}((\text{mergeMultiArrows}(ms)).1)$
9. $(\text{mergeMultiArrow}(m_1, m_2)).2 \leq m_1.2 \cap m_2.2$
10. $m_1.2 \cap m_2.2 \leq (\text{mergeMultiArrow}(m_1, m_2)).2$
11. *for all* $(\text{src}_1, (\text{src}_2, \text{src}_3))$ *in* $\text{zip}(\text{mergeMultiArrow}(m_1, m_2).1, \text{zip}(m_1.1, m_2.1))$*:* $\text{src}_1 \leq \text{src}_2 \cap \text{src}_3$
12. *for all* $(\text{src}_1, (\text{src}_2, \text{src}_3))$ *in* $\text{zip}(\text{mergeMultiArrow}(m_1, m_2).1, \text{zip}(m_1.1, m_2.1))$*:* $\text{src}_2 \cap \text{src}_3 \leq \text{src}_1$
13. $(\text{mergeMultiArrows}(ms)).2 \leq \bigcap_{m_i \in ms} m_i.2$
14. $\bigcap_{m_i \in ms} m_i.2 \leq (\text{mergeMultiArrows}(ms)).2$
15. *If for all* m_1 *in* ms*: for all* m_2 *in* ms*:* $\text{lsize}(m_1) = \text{lsize}(m_2)$ *then* $\text{nth}(\omega, (\text{mergeMultiArrows}(ms)).1, n) \leq \bigcap_{m_i \in ms} \text{nth}(\omega, m_i.1, n)$

16. *If for all* m_1 *in* ms*: for all* m_2 *in* ms*:* $\text{lsize}(m_1) = \text{lsize}(m_2)$ *then*
$\bigcap_{m_i \in ms} \text{nth}(\omega, m_i.1, n) \leq \text{nth}(\omega, (\text{mergeMultiArrows}(ms)).1, n)$

17. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and for all* i_1 *in* p_1*:* toCover_prime(i_1) *then*
for all i_2 *in* p_2*:* toCover_prime(i_2)

18. *If for all* A *in* toCover*:* isPrimeComponent(A) *and*
for all A *in* toCover*:* $m.2 \leq A$ *implies* A *is in* covered *and*
$m.2 \leq \bigcap_{A_i \in \text{covered}} A_i$ *and*
$\bigcap_{m_i \in [::m\&ms]} m_i.2 \leq \bigcap_{A_i \in \text{toCover}} A_i$ *then*
$m.2 \leq \bigcap_{A_i \in (\text{partitionCover(covered,toCover)}).1} A_i$ *and*
$\bigcap_{m_i \in ms} m_i.2 \leq \bigcap_{A_i \in (\text{partitionCover(covered,toCover)}).2} A_i$

19. *If* m *is in* filterMergeMultiArrows(map($\lambda ms.[::m_1\&ms], mss$)) *then*
$m = m_1$ *or*
exists ms *in* mss *s.t.* mergeMultiArrows(ms) *is in* filterMergeMultiArrows(mss)

20. *If for all* $((\text{srcs}_i, \text{tgt}_i), \text{covered}_i)$ *in* $[::((\text{srcs}, \text{tgt}), \text{covered})\&\text{splits}]$:

$$\text{tgt}_i \leq \bigcap_{A_i \in \text{covered}_i} A_i \text{ and}$$

for all A *in* toCover : $A \leq \text{tgt}_i$

then $\text{tgt} \leq \bigcap_{A_i \in} \text{toCover}$ *and*

for all A *in* toCover : $\text{tgt} \leq A$ *implies* A *is in* covered

21. *If* (partitionCover(covered, toCover)).1 = [::] *then*
(partitionCover(covered, toCover)).2 = toCover

22. *If* (partitionCover(covered, toCover)).2 = [::] *then*
(partitionCover(covered, toCover)).1 = toCover

23. *If for all* m_1 *in* $[::m\&ms]$*: for all* m_2 *in* $[::m\&ms]$*:* $\text{lsize}(m_1) = \text{lsize}(m_2)$ *and*
not $ms = [::]$ *then*
$\text{lsize}((\text{mergeMultiArrows}([::m\&ms])).1) = \text{lsize}((\text{mergeMultiArrows}(ms)).1)$

24. *If* A *is in* toCover *then*
A *is in* (partitionCover(covered, toCover)).1 ++ (partitionCover(covered, toCover)).2

25. $\bigcap_{A_i \in (\text{partitionCover(covered,toCover)}).1 + (\text{partitionCover(covered,toCover)}).2} A_i \leq \bigcap_{A_i \in \text{toCover}} A_i$

26. *If for all* A_i *in* toCover*:* $A \leq A_i$ *implies* B *is in* covered *and*
$A \leq \bigcap_{A_i \in \text{toCover}} A_i$ *then*
(partitionCover(covered, toCover)).1 = [::]

27. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and*
for all i *in* p_1*:*
instruction_covered(i) *and*
toCover_prime(i) *and*
currentResultNotDone(i)
then for all i *in* p_2*:* currentResultNotDone(i)

28. *If not* toCover = [::] *and for all* A *in* toCover*: not* tgt $\leq A$ *then*
not tgt $\leq \bigcap_{A_i \in \text{toCover}} A_i$

29. filterMergeMultiArrows(map($\lambda ms.$[:: m_1&ms], map($\lambda ms.$[:: m_2&ms], mss))) =
filterMergeMultiArrows(map($\lambda ms.$[:: mergeMultiArrow($m1, m2$)&ms], mss))

30. *for all* $(\text{src}_1, \text{src}_2)$ *in* zip(srcs, (mergeMultiArrow(m, (srcs, tgt))).1)*:* $\text{src}_2 \leq \text{src}_1$

31. $A \cap B \leq A \pitchfork B$

32. $A \pitchfork B \leq A \cap B$

33. (mergeMultiArrow(m, (srcs, tgt))).1 = map(λsrcs.srcs.1 $\pitchfork$ srcs.2, zip(srcs, m.1))

34. *If for all* ((srcs, tgt), covered) *in* splits*:*
$\text{tgt} \leq \bigcap_{A_i \in \text{covered}} A_i$ *and*
for all A *in* toCover : tgt $\leq A$ *implies* A *is in* covered
and for all A *in* toCover*:* isPrimeComponent(A) *and*
not stillPossible(splits, toCover) *and*
m *is in* filterMergeMultiArrows(subseqs(map($\lambda mc.mc$.1, splits))) *then*
not $m.2 \leq \bigcap_{A_i \in \text{toCover}} A_i$

35. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and*
suffix(s_2, s) *and*
for all i *in* p_1*:*
arity_equal(i) *and*
not_omega_instruction(i) *and*
instruction_covered(i) *and*
toCover_prime(i) *and*
currentResultNotDone(i)
and for all i *in* p_2*:* complete(s, i) *then*
for all i *in* p_1*:* complete(s, i)

36. *If* $(s_1, p_1) \rightsquigarrow_* (s_2, [::])$ *and*
for all i *in* p_1*:*
arity_equal(i) *and*
not_omega_instruction(i) *and*
instruction_covered(i) *and*
toCover_prime(i) *and*
currentResultNotDone(i)
then for all i *in* p_1*:* complete(s_2, i)

PROOF Statement 36 is the interesting property to show. The complete-predicate is applied to all the initial instructions and the final state. For each instruction it ensures that the state includes a multi arrow sufficient to represent all possible merged solution subsets of multi arrows specified in the instruction, including the current result if present. Merged multi arrow subsets are only possible solutions if their merged target meets the goal, i.e., it has to be a subtype of all the types to cover by the instruction. A multi arrow is sufficient to represent a merged subset multi arrow, if its sources are greater or equal and the target is enough to meet the goal. For example multi arrow $([::a], b)$ is sufficient to represent $([::a \cap c], b \cap d)$ if only b is to cover: sources of resulting multi arrows will eventually become recursive inhabitation targets and the terms inhabiting a include all the terms inhabiting $a \cap b$. Just like the proof for soundness, the proof for statement 36 requires induction, decomposition of the first step and then reasoning backwards. Statements 5, 6, 7, 17, and 27 make it possible to apply the induction hypothesis by showing that invariants still hold after the first step. Each of them is proven by case-analysis on the derivation. Backwards reasoning is provided by statement 35. It requires a detailed and careful case analysis on the many possibilities to derive $(s_1, p_1) \rightsquigarrow (s_2, p_2)$.

During this analysis all the invariants have to be taken into account. Invariant arity_equal ensures, that all multi arrows in an instruction have the same number of sources: type $(a \to b \to c) \cap (d \to e)$ can be used to inhabit c and $(b \to c) \cap e$ but not $c \cap e$ which would require the multi arrows $([::\ b\&[::a]], c)$ and $([::d], e)$ with different numbers of sources. Late termination is prevented by not_omega_instruction and currentResultNotDone. Late termination would occur if the target of the current result in an instruction is smaller or equal than all of the types to cover (invariant currentResultNotDone). Alternatively, if there is no current target, there could be no types to cover. Both situations would lead to unnecessarily specialized sources, restricting recursive inhabitation targets to types too small. Invariant not_omega_instruction asserts that normal operation of the machine will stop before the targets to cover by an instruction are empty. Also targets greater than ω have to be sorted out before starting the machine. This is because subtype rule $(\to \omega)$ converts any such target to an arrow of arbitrary length, which is incompatible with invariant arity_equal. Machine operations also rely on the possibility to discharge targets in toCover by adding multi arrows one-by-one. This is possible if the targets are prime (asserted by invariant toCover_prime): for a prime target tgt and multi arrows m_1 and m_2, $m_1.2 \cap m_2.2 \leq$ tgt if $m_1.2 \leq$ tgt or $m_2.2 \leq$ tgt and no distribution across m_1

and m_2 has to be taken into account. When processing a single multi arrow, its covered targets are classified by partitionCover according to their relevance for the remaining targets to cover. In splits each multi arrow is paired with its covered targets, allowing partitionCover to test for equality instead of performing costly subtype checks. Invariant instruction_covered ensures multi arrows are exactly paired with targets they cover. Inclusion of too few targets would render CHECK and SKIP-steps incomplete by triggering premature exclusion of possibilities. For the other steps, too many covered targets for a single multi arrow exclude future choices by removing too many elements from the targets yet to cover.

The rest of the statements of Lemma 24 are technical necessities that simplify backward reasoning in statement 35 and maintaining the invariants across steps. None of them are particularly difficult if they are proven in the order presented. ■

There is another important soundness property, which will be proven next.

Lemma 25 (Target Soundness) *Target soundness is defined as a Boolean predicate on states* $s \in (\mathbb{T}^* \times \mathbb{T})^*$ *and programs* $p \in \mathbb{I}^*_{\text{Cover}}$*:*

$$
\begin{aligned}
&\text{tgt\_sound}(s, p) = \\
&\quad \textit{for all } (\text{srcs}, \text{tgt}) \textit{ in } s: \\
&\qquad \textit{there exists } i \textit{ in } p \textit{ s.t. } \text{tgt} \leq \bigcap_{A_i \in \Delta} A_i \\
&\qquad\qquad \textit{for } \Delta = \begin{cases} [::\text{currentResult.2\&toCover}] \\ \qquad \textit{for } i = \texttt{ContinueCover} \text{ splits toCover currentResult or} \\ \qquad\quad i = \texttt{CheckContinueCover} \text{ splits toCover currentResult} \\ \text{toCover} \\ \qquad \textit{for } i = \texttt{Cover} \text{ splits toCover or} \\ \qquad\quad i = \texttt{CheckCover} \text{ splits toCover} \end{cases}
\end{aligned}
$$

For $i \in \mathbb{I}_{\text{Cover}}$*, and* $p_1, p_2 \in \mathbb{I}^*_{\text{Cover}}$*, and* $s_1, s_2 \in \mathbb{O}_{\text{Cover}}$ *the following statements are true:*

1. *If* $(s_1, p_1) \rightsquigarrow (s_2, p_2)$ *and for all* i *in* p_1*:* instruction_covered(i) *then* tgt_sound(take(lsize(s_2) − lsize$(s_1), s_2), p_1$)

2. *If* $(s_1, [::i]) \rightsquigarrow (s_2, p_2)$ *and* instruction_covered(i) *and* tgt_sound(s, p) *then* tgt_sound$(s, [::i])$

3. *If* $(s_1, p_1) \rightsquigarrow_* (s_2, p_2)$ *and for all* i *in* p_1*:* instruction_covered(i) *then* tgt_sound(take(lsize(s_2) − lsize$(s_1), s_2), p_1$)

PROOF Target soundness ensures that the target of every multi arrow in the result state s covers at least one of the sets specified in an instruction (and the associated intermediate result if it exists). The proof is similar to the proofs of Lemma 24.36 and Lemma 23.5. It again

combines forwards and backwards reasoning after an induction on the machine steps and decomposition of state and programs using the machine stack behavior established in Lemma 22. Forward and backward steps are proven by case analysis on the derivation, where the backwards reasoning in statement 2 has been further simplified to account for just one instruction (decomposition with Lemma 22 is strong enough for this). Invariant instruction_covered again prevents the final state from becoming too small because of premature result exclusion. ■

The five preconditions for completeness in Lemma 24.36 make proper initialization mandatory. Additionally types in context Γ have to be preprocessed into multi arrows to be accepted in instructions. This preprocessing is performed by function splitTy defined next.

Definition 26 (Preprocessing Types into Multi Arrows) The list of lists of all multi arrows of equal length for a given type A is computed by $\text{splitTy} : \mathbb{T} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$

$$\text{splitTy}(A) = \begin{cases} [::] & \text{for isOmega}(A) \\ [::\ ([::], A) \& \text{splitRec}(A, [::], [::])] & \text{otherwise} \end{cases}$$

$$\text{splitRec}(A, \text{srcs}, \Delta) = \begin{cases} [::\ [::\ ([::\ A_1 \& \text{srcs}], A_2) \& \Delta_1] \& \text{splitRec}(A_2, [::\ A_1 \& \text{srcs}], \Delta_2)] \\ \qquad \text{for } A = A_1 \to A_2 \text{ and } (\Delta_1, \Delta_2) = \text{safeSplit}(\Delta) \\ \text{splitRec}(A_2, \text{srcs}, \Delta) \qquad \text{for } A = A_1 \cap A_2 \text{ and isOmega}(A_1) \\ \text{splitRec}(A_1, \text{srcs}, \Delta) \\ \qquad \text{for } A = A_1 \cap A_2 \text{ and isOmega}(A_2) \text{ and not isOmega}(A_2) \\ \text{splitRec}(A_1, \text{srcs}, \text{splitRec}(A_2, \text{srcs}, \Delta)) \\ \qquad \text{for } A = A_1 \cap A_2 \text{ and not isOmega}(A_2) \text{ and not isOmega}(A_2) \\ \Delta \qquad \text{otherwise} \end{cases}$$

$$\text{safeSplit}(\Delta) = \begin{cases} ([::], [::]) & \text{for } \Delta = [::] \\ (\Delta', [::]) & \text{for } \Delta = [::\Delta'] \\ (\Delta_1, \Delta_2) & \text{for } \Delta = [::\ \Delta_1 \& \Delta_2] \end{cases}$$

□

Function splitTy first sorts out types equal to ω and then proceeds recursively on the type structure. During recursion the sources of the multi arrow currently under construction and a list Δ of computed results are maintained. For constructor symbols and products recursion stops, returning the computed results. These already include the current type as target from a previous step. Intersections are processed component-wise, again filtering out all types equal to ω. For arrows the result list is split into a head and tail list. The head list contains all results with $\text{lsize}(\text{srcs}) + 1$ sources and a new multi arrow constructed from the current arrow is

inserted into it. The remaining arrows of the current arrow target are recursively constructed from the tail list, which contains the results with more sources. The current source is added to the sources for arrows under construction.

Lemma 26 (Properties of Type Preprocessing) *Define the recursive Boolean predicate*

$$\text{arity\_increasing}(n, \Delta) = \begin{cases} \textit{true} & \textit{for } \Delta = [::] \\ \text{arity\_increasing}(n+1, \Delta_2) \textit{ and} & \textit{for } \Delta = [::\ \Delta_1 \& \Delta_2] \\ \quad \textit{for all } (\text{srcs}, \text{tgt}) \textit{ in } \Delta_1 : n = \text{lsize}(\text{srcs}) & \end{cases}$$

For $c \in \mathbf{C}$, $n \in \mathbb{N}$, *and* $\text{srcs}, \text{srcs}_1, \text{srcs}_2 \in \mathbb{T}^*$, *and* $A, B \in Types$, *and* $\Delta, \Delta_1, \Delta_2, \Delta_3 \in ((\mathbb{T}^* \times \mathbb{T})^*)^*$, *and* $f : (\mathbb{T}^* \times \mathbb{T})^* \to (\mathbb{T}^* \times \mathbb{T})^*$ *the following statements are true:*

1. $\text{arity\_increasing}(n, \Delta_1 +\!\!+ \Delta_2)$ *iff*
   $\text{arity\_increasing}(n, \Delta_1)$ *and* $\text{arity\_increasing}(n + \text{lsize}(\Delta_1), \Delta_2)$

2. *If* $\text{arity\_increasing}(\text{lsize}(\text{srcs}) + 1, \Delta)$ *then*
   $\text{arity\_increasing}(\text{lsize}(\text{srcs}) + 1, \text{splitRec}(A, \text{srcs}, \Delta))$

3. $\text{arity\_increasing}(0, \text{splitTy}(A))$

4. *For all* ms *in* Δ*: for all* $(\text{srcs}_1, \text{tgt}_1)$ *in* ms*: for all* $(\text{srcs}_2, \text{tgt}_2)$ *in* ms*:*
   $\text{lsize}(\text{srcs}_1) = \text{lsize}(\text{srcs}_2)$

5. $\bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \Delta), n)} \text{mkArrow}(m_i) \leq \bigcap_{m_i \in \text{nth}([::], \Delta, n)} \text{mkArrow}(m_i)$

6. *If* $\text{lsize}(\Delta_1) = \text{lsize}(\Delta_2)$ *then* $\text{lsize}(\text{splitRec}(A, \text{srcs}, \Delta_1)) = \text{lsize}(\text{splitRec}(A, \text{srcs}, \Delta_2))$

7. *If* $\text{lsize}(\Delta_1) = \text{lsize}(\Delta_2)$ *and*
   $\bigcap_{m_i \in \text{nth}([::], \Delta_1, n)} \text{mkArrow}(m_i) \leq \bigcap_{m_i \in \text{nth}([::], \Delta_2, n)} \text{mkArrow}(m_i)$ *then*
   $\bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \Delta_1), n)} \text{mkArrow}(m_i) \leq \bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \Delta_2), n)} \text{mkArrow}(m_i)$

8. *If* $B \leq \bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \text{nseq}(\text{lsize}(\Delta), [::])), n)} \text{mkArrow}(m_i)$ *and*
   $B \leq \bigcap_{m_i \in \text{nth}([::], \Delta, n)} \text{mkArrow}(m_i)$ *then*
   $B \leq \bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \Delta), n)} \text{mkArrow}(m_i)$

9. *If* $\text{mkArrow}(\text{srcs}, A) \leq \bigcap_{m_i \in \text{nth}([::], \Delta, n)} \text{mkArrow}(m_i)$ *then*
   $\text{mkArrow}(\text{srcs}, A) \leq \bigcap_{m_i \in \text{nth}([::], \text{splitRec}(A, \text{srcs}, \Delta), n)} \text{mkArrow}(m_i)$

10. $A \leq \bigcap_{m_i \in \text{nth}([::], \text{splitTy}(A), n)} \text{mkArrow}(m_i)$

Define

- $\text{merge}(\Delta_1, \Delta_2) =$

$$\begin{cases} [::\, ms_1 +\!\!+ ms_2 \& \text{merge}(\Delta_1', \Delta_2')] & \textit{for } \Delta_1 = [::\, ms_1 \& \Delta_1'] \textit{ and } \Delta_2 = [::\, ms_2 \& \Delta_2'] \\ \Delta_2 & \textit{for } \Delta_1 = [::] \\ \Delta_1 & \textit{for } \Delta_2 = [::] \end{cases}$$

- $\text{splitTy\_slow}(A) =$

$$\begin{cases} [::] \quad \textit{for } \text{isOmega}(A) \\ [::\, [::([::], A_1 \to A_2)] \& \text{map}(\lambda ms.\, \text{map}(\lambda m.(\text{rcons}(m.1, A_1), m.2), ms), \text{splitTy\_slow}(A_2)] \\ \qquad \textit{for } A = A_1 \to A_2 \textit{ and not } \text{isOmega}(A) \\ [::\, [::([::], A_1 \cap A_2)] \& \text{behead}(\text{merge}(\text{splitTy\_slow}(A_1), \text{splitTy\_slow}(A_2)))] \\ \qquad \textit{for } A = A_1 \cap A_2 \textit{ and not } \text{isOmega}(A) \\ [::[::([::], A)]] \quad \text{for not isOmega}(A) \end{cases}$$

then:

11. $\text{merge}(\text{merge}(\Delta_1, \Delta_2), \Delta_3) = \text{merge}(\Delta_1, \text{merge}(\Delta_2, \Delta_3))$

12. $\text{merge}(\Delta, [::]) = \Delta$

13. $\text{merge}([::], \Delta) = \Delta$

14. $\text{splitRec}(A, \text{srcs}, \Delta) = \text{merge}(\text{splitRec}(A, \text{srcs}, [::]), \Delta)$

15. *If for all* $ms_1, ms_2 \in (\mathbb{T}^* \times \mathbb{T})^*$: $f(ms_1 +\!\!+ ms_2) = f(ms_1) +\!\!+ f(ms_2)$ *then*
$\text{map}(f, \text{merge}(\Delta_1, \Delta_2)) = \text{merge}(\text{map}(f, \Delta_1), \text{map}(f, \Delta_2))$

16. $\text{splitRec}(A, \text{srcs}_1 +\!\!+ \text{srcs}_2, [::]) =$
$\text{map}(\lambda ms.\, \text{map}(\lambda m.(m.1 +\!\!+ \text{srcs}_2, m.2), ms), \text{splitRec}(A, \text{srcs}_1, [::]))$

17. $\text{splitTy}(A) = \text{splitTy\_slow}(A)$

18. *If* $\text{isOmega}(A)$ *then* $\text{splitTy\_slow}(A) = [::]$

19. $\text{nth}([::], \text{merge}(mss_1, mss_2), n) = \text{nth}([::], mss_1, n) +\!\!+ \text{nth}([::], mss_2, n)$

20. $\text{nth}([::], \text{splitTy\_slow}(A_2), n+1) \sqsubseteq \text{nth}([::], \text{splitTy\_slow}(A_1 \cap A_2), n+1)$

21. *If* $c(A) \leq \text{mkArrow}(\text{srcs}, B)$ *then*
$(\text{mergeMultiArrows}($
$\quad \text{filter}($
$\quad\quad \lambda m.\, \text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and for all* $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2,$
$\quad\quad \text{nth}([::], \text{splitTy}(c(A)), \text{lsize}(\text{srcs})))).2 \leq B$

22. *If* $A_1 \star A_2 \leq \text{mkArrow}(\text{srcs}, B)$ *then*
(mergeMultiArrows(
filter(
$\lambda m.\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and for all* $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$,
$\text{nth}([::], \text{splitTy}(A_1 \star A_2), \text{lsize}(\text{srcs})))))).2 \leq B$

23. *If* $\omega \leq \text{mkArrow}(\text{srcs}, B)$ *then*
(mergeMultiArrows(
filter(
$\lambda m.\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and for all* $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$,
$\text{nth}([::], \text{splitTy}(\omega), \text{lsize}(\text{srcs})))))).2 \leq B$

24. *If* $A \leq \text{mkArrow}(\text{srcs}, B)$ *then*
(mergeMultiArrows(
filter(
$\lambda m.\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and for all* $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$,
$\text{nth}([::], \text{splitTy}(A), \text{lsize}(\text{srcs})))))).2 \leq B$

25. *If* $A \leq \text{mkArrow}(\text{srcs}, B)$ *and not* $\text{isOmega}(\text{mkArrow}(\text{srcs}, B))$ *then*
exists m *in* $\text{filterMergeMultiArrows}(\text{subseqs}(\text{nth}([::], \text{splitTy}(A), \text{lsize}(\text{srcs}))))$, *s.t.*
$\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and*
for all $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$ *and*
$m.2 \leq B$

PROOF Statements 1-10 prove soundness of splitTy with respect to its specification. The resulting multi arrows are ordered increasing by their source count (statement 3). After recombination with mkArrow, type A is less than each of the multi arrows it includes (statement 10). When done in the order presented, proofs are unsurprising and follow by induction on the first context mentioned in statement 1 and 2, and structural induction on the type that is split in the other statements. Completeness is specified in statement 25 and requires a less straight-forward proof. The main difficulty is to find invariants for the accumulated result state Δ in the definition of splitRec. Instead the proof strategy is to replace splitTy by a slower version splitTy_slow with linear overhead for list concatenation instead of constant overhead for manipulating the head of the lists. Some easy proofs show that this slower version produces equal results (statements leading up to 17) and behaves well with ω, distribution of multi arrows, and intersections (statements 18 - 20). Finally, the completeness proof in statement 25 follows from its alternative representation in 24, which constructs the desired multi arrow selection to provide completeness instead of postulating its existence. The proof of statement 24 proceeds by structural induction on A. The easier cases for constructors, products and omega are managed by 21, 22, and 23. The case for ω, 23, is immediate, while for the other cases reverse induction on the list of sources has to be performed. Then splitTy is replaced by splitTy_slow, which can be evaluated one step before replacing it again with splitTy to be able

to use induction hypotheses which are otherwise not applicable to the evaluated recursive calls to splitTyRec. ■

Preprocessing types allows for proper calls to the cover machine.

Lemma 27 (Proper Cover Machine Calls) *For* $ms \in (\mathbb{T}^* \times \mathbb{T})^*$, $\text{srcs} \in \mathbb{T}^*$, *and* $A, B \in \mathbb{T}$, *and* $s \in \mathbb{O}_{\text{Cover}}$, *with* $Bs = \text{primeFactors}(B)$, *and* $mss = \text{splitTy}(A)$ *the following statements are true:*

1. *for all* i *in* $\text{map}(\lambda ms.\text{map}(\lambda m.(m, \text{filter}(\lambda B.m.2 \leq B, Bs)), ms), mss)$*:* $\text{instruction\_covered}(i)$
2. *If* $A \leq \text{mkArrow}(\text{srcs}, B)$ *and not* $\text{isOmega}(B)$ *and* $([::], \text{map}(\lambda ms.\texttt{Cover}\ (\text{map}(\lambda m.(m, \text{filter}(\lambda B.m.2 \leq B, Bs)), ms))\ Bs, mss)) \leadsto_* (s, [::])$ *then there exists* m *in* s*, s.t.*
   $\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and*
   for all $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$ *and*
   $m.2 \leq B$
3. *If for all* m_1 *in* ms*: for all* m_2 *in* ms*:* $\text{lsize}(m_1) = \text{lsize}(m_2)$ *then* $\bigcap_{m_i \in ms} \text{mkArrow}(m_i) \leq \text{mkArrow}(\text{mergeMultiArrows}(\text{ms}))$
4. *If not* $\text{isOmega}(B)$ *and* $([::], \text{map}(\lambda ms.\texttt{Cover}\ (\text{map}(\lambda m.(m, \text{filter}(\lambda B.m.2 \leq B, Bs)), ms))\ Bs, mss)) \leadsto_* (s, [::])$ *then for all* m *in* s*:* $A \leq \text{mkArrow}(m)$
5. *If* $([::], \text{map}(\lambda ms.\texttt{Cover}\ (\text{map}(\lambda m.(m, \text{filter}(\lambda B.m.2 \leq B, Bs)), ms))\ Bs, mss)) \leadsto_* (s, [::])$ *then for all* m *in* s*:* $m.2 \leq B$

PROOF Statement 1 establishes the instructionsCovered invariant later used for Lemma 24.36 in statement 2 and Lemma 25.3 in statement 5. It easily follows from the definitions of map and filter. Statement 2 proves completeness: for any multi arrow (srcs, B) greater or equal than A, a subtype compatible multi arrow with the same number of sources can be found in the final state of the cover machine. The proof uses Lemma 24.36. Its preconditions are provided using Lemma 26.3 with Lemma 26.4, Lemma 9.17, statement 1, Theorem 2.3, and observing that no intermediate results are present in the initial input. Statement 3 is proven by induction on ms. The soundness proof in statement 4 proceeds by instantiating Lemma 23.9 and Lemma 26.10, induction on s, and combining the instantiated lemmas with subtype transitivity and statement 3. Finally, target soundness in statement 5 follows from Theorem 2.1, Lemma 25.3 with statement 5, and transitivity of subtyping. ■

A final post processing step is applied to the results of the cover machine. Sometimes independent choices can lead to redundancy, e.g. in $(a \cap b \to c \cap d) \cap (a \to c) \cap (b \to d)$ with target $c \cap d$ the machine has to independently inspect possibilities for $(a \to c) \cap (b \to d)$ for an execution branch not choosing $a \cap b \to c \cap d$. This leads to duplicated inclusion of $a \cap b \to c \cap d$ in the results. Fortunately, these duplicates can easily be removed.

Definition 27 (Cover Machine Result Reduction) The result reduction function defined recursively on its input list:

$$
\begin{array}{l}
\text{reduceMultiArrows} : (\mathbb{T}^* \times \mathbb{T})^* \to (\mathbb{T}^* \times \mathbb{T})^* \\
\text{reduceMultiArrows}(ms) = \\
\left\{
\begin{array}{l}
ms_r \\
\quad \text{for } ms = [::\, m_1 \& ms'] \text{ and} \\
\qquad ms_r = \text{reduceMultiArrows}(ms') \text{ and} \\
\qquad \text{exists } (\text{srcs}, \text{tgt}) \text{ in } ms_r \text{ s.t.} \\
\qquad\quad \text{lsize}(\text{srcs}) = \text{lsize}(m_1.1) \text{ and} \\
\qquad\quad \text{for all } (\text{src}_1, \text{src}_2) \text{ in } \text{zip}(m_1.1, \text{srcs}) : \text{src}_1 \leq \text{src}_2 \\
[::\, m_1 \& ms'_r] \\
\quad \text{for } ms = [::\, m_1 \& ms'] \text{ and} \\
\qquad ms_r = \text{reduceMultiArrows}(ms') \text{ and} \\
\qquad \text{not exists } (\text{srcs}, \text{tgt}) \text{ in } ms_r \text{ s.t.} \\
\qquad\quad \text{lsize}(\text{srcs}) = \text{lsize}(m_1.1) \text{ and} \\
\qquad\quad \text{for all } (\text{src}_1, \text{src}_2) \text{ in } \text{zip}(m_1.1, \text{srcs}) : \text{src}_1 \leq \text{src}_2 \\
\qquad \text{and } ms'_r = \text{filter}(\lambda m_2.\text{not } (\text{lsize}(m_2.1) = \text{lsize}(m_1.1) \text{ and} \\
\qquad\qquad\qquad\qquad \text{for all } (\text{src}_1, \text{src}_2) \text{ in } \text{zip}(m_2.1, m_1.1) : \\
\qquad\qquad\qquad\qquad\quad \text{src}_1 \leq \text{src}_2), ms_r) \\
[::] \qquad \text{otherwise}
\end{array}
\right.
\end{array}
$$

Lemma 28 (Cover Machine Result Reduction Properties) *For* $ms \in (\mathbb{T}^* \times \mathbb{T})^*$, $\text{srcs} \in \mathbb{T}^*$, *and* $A, B, \text{tgt} \in \mathbb{T}$ *function* reduceMultiArrows *has the following properties:*

1. $\text{reduceMultiArrows}(ms) \sqsubseteq ms$
2. *If for all* m *in* ms*:* $A \leq \text{mkArrow}(m)$ *then*
   for all m *in* $\text{reduceMultiArrows}(ms)$*:* $A \leq \text{mkArrow}(m)$
3. *If for all* m *in* ms*:* $m.2 \leq B$ *then*
   for all m *in* $\text{reduceMultiArrows}(ms)$*:* $m.2 \leq B$
4. *If for all* m *in* ms*:* $m.2 \leq B$ *and*
   exists m *in* ms*, s.t.*
   $\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and*
   for all $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$ *and*
   $m.2 \leq B$
   then exists m *in* $\text{reduceMultiArrows}(ms)$*, s.t.*
   $\text{lsize}(m.1) = \text{lsize}(\text{srcs})$ *and*
   for all $(\text{src}_1, \text{src}_2)$ *in* $\text{zip}(\text{srcs}, m.1) : \text{src}_1 \leq \text{src}_2$ *and*
   $m.2 \leq B$

PROOF Statement 1 is shown by induction on ms and the (target) soundness preservation properties of statements 2 and 3 immediately follow from it. Completeness preservation, which is stated in 4, is proven by induction on ms and case analysis on the conditions checked by reduceMultiArrows. ■

2.5.2 Generation of Tree Grammars

Results of the cover machine are combined into tree grammars, which allow for easy enumeration. This section discusses an algorithm, which grows these tree grammars in a step-wise way. Its correctness properties are specified and proven. Specifically the set of inhabitants of a requested type coincides with the language of the resulting grammar.

Definition 28 (Regular Normalized Tree Grammar and their Language) In [42], chapter 2, regular normalized tree grammars are defined by $G = (S, N, \mathscr{F}, R)$, where

- $S \in N$ is an axiom (start symbol)
- N is a set of non terminal symbols
- $\mathscr{F}$ is a set of terminal symbols disjoint from N
- R is a set of rules of shape $A \mapsto f(A_1, A_2, \ldots, A_n)$ or $A \mapsto f$ where $f \in \mathscr{F}$, and $A, A_1, A_2, \ldots A_n \in N$
- Arities are used consistently, i.e. if $f \in \mathscr{F}$ occurs in $r_1, r_2 \in R$ then either there exists some $n \in \mathbb{N}$ and for $i \in 1,2$ and $1 \leq k \leq n$: $A^i, A^i_k \in N$ s.t. either $r_1 = A^1 \mapsto f(A^1_1, A^1_2, \ldots, A^1_n)$ and $r_2 = A^2 \mapsto f(A^2_1, A^2_2, \ldots, A^2_n)$ or there exist $A_1, A_2 \in N$ s.t. $r_1 = A^1 \mapsto f$ and $r_2 = A^2 \mapsto f$.

Language $L(G, A)$ for start symbol A is the least set closed under the following rules:

$$\frac{A \mapsto f \in R}{f \in L(G, A)} \qquad \frac{A \mapsto f(A_1, A_2, \ldots, A_n) \in R \qquad t_k \in L(G, A_k) \text{ for } k \in \{1, 2, \ldots, n\}}{f(t_1, t_2, \ldots, t_n) \in L(G, A)}$$

Additionally language $L(G)$ is defined to be $L(G, S)$. □

Tree Grammars are well-studied concepts with good formal properties. These include efficient decision procedures for emptiness, uniqueness, and infinity of their associated languages, as well as an equivalent machine model, non-deterministic finite tree-automata (NFTA) [42]. Note that the associated NFTA model can express the same languages but is algorithmically weaker than the alternating tree-automata constructed in [165]: transforming a given alternating automaton into an automaton without alternation introduces exponential blowup.

Rehof and Urzyczyn [165] state that this blowup occurring in addition to the already exponential worst-case size prevents the alternating automaton construction from being a good solution for deciding inhabitation with subtyping. The tree grammar approach does not introduce double exponential blowup: for input size n there exist polynomial functions p_1, p_2, p_3 such that there can be up to $2^{p_1(n)}$ calls to the cover machine, which by Lemma 21.2 takes at most $2^{p_2(n)}$ steps, adding at most $p_3(n)$ many recursive targets (multi arrow sources) per step. The resulting tree grammar only contains rules for those targets, so its size is bound by $2^{p_1(n)} \cdot 2^{p_2(n)} \cdot p_3(n) \leq 2^{p_1(n)+p_2(n)+\log(p_3(n))}$. Intuitively, the reason for the performance improvement compared to the alternating automaton approach is that alternation elimination is oblivious of the fact that the task of the cover machine is independent from the main algorithm loop. It computes the powerset of main loop and cover computation states, instead of their Cartesian product.

The main algorithm and its helper functions are defined next. The behavior is specified as a machine performing step-wise transformations of rule lists for tree grammars.

Definition 29 (Inhabitation Machine) For $\Gamma : \mathbf{B} \to \mathbb{T}$ define:

- The preprocessed multi arrow context $\Gamma' : \mathbf{B} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$

  $$\Gamma'(c) = \text{splitTy}(\Gamma(c))$$

- The set of all possible rules

  $$\mathbb{R} \ni r ::= A \mapsto \bot \mid A \mapsto c \mid A \mapsto @(B, C)$$

  for $c \in \mathbf{B}$, and $A, B, C \in \mathbb{T}$

- Function computeFailExisting $: \mathbb{R}^* \times \mathbb{T} \to \mathbb{B} \times \mathbb{B}$

  $$\text{computeFailExisting}(G, A) = \begin{cases} (\text{true}, \text{true}) & \text{for } G = [:: A \mapsto \bot \& G'] \\ (\text{true}, \text{ exists } A \mapsto \bot \text{ in } G') & \text{for } G = [:: B \mapsto \bot \& G'] \text{ and not } A = B \text{ and } A \leq B \\ \text{computeFailExisting}(G', A) & \text{for } G = [:: B \mapsto \bot \& G'] \text{ and not } A = B \text{ and not } A \leq B \\ \text{computeFailExisting}(G', A) & \text{for } G = [:: B \mapsto c \& G'] \\ (\text{false}, \text{true}) & \text{for } G = [:: B \mapsto @(C, A) \& G'] \\ \text{computeFailExisting}(G', A) & \text{for } G = [:: B \mapsto @(C, D) \& G'] \text{ and not } A = D \\ (\text{false}, \text{false}) & \text{otherwise} \end{cases}$$

- Function commitMultiArrow : $\mathbb{R}^* \times \mathbf{B} \times (\mathbb{T}^* \times \mathbb{T}) \to \mathbb{R}^*$

$$\text{commitMultiArrow}(G, c, (\text{srcs}, \text{tgt})) =$$
$$\begin{cases} \text{commitMultiArrow}([::\ \text{tgt} \mapsto @(\text{src} \to \text{tgt}, \text{src})\&G], c, (\text{srcs}', \text{src} \to \text{tgt})) \\ \qquad \text{for srcs} = [::\ \text{src}\&\text{srcs}'] \\ [::\ \text{tgt} \mapsto c\&G] \quad \text{otherwise} \end{cases}$$

- Function commitUpdates : $\mathbb{R}^* \times \mathbb{T} \times \mathbf{B} \times (\mathbb{T}^* \times \mathbb{T})^* \to \mathbb{R}^*$

$$\text{commitUpdates}(G, \text{tgt}, c, \text{covers}) =$$
$$\begin{cases} \text{commitUpdates}(\text{commitMultiArrow}(G, c, \text{srcs}, (\text{srcs}, \text{tgt})), \text{tgt}, c, \text{covers}') \\ \qquad \text{for covers} = [::\ (\text{srcs}, \text{tgt}')\&\text{covers}'] \\ G \quad \text{otherwise} \end{cases}$$

- Function dropTargets : $\mathbb{R}^* \to \mathbb{R}^*$

$$\text{dropTargets}(G) =$$
$$\begin{cases} \text{dropTargets}(G') & \text{for } G = [::\ A \mapsto @(B, C)\&G'] \text{ or } G = [::\ A \mapsto \bot\&G'] \\ G & \text{otherwise} \end{cases}$$

- Function accumulateCovers : $(\mathbf{B} \to (\mathbb{T}^* \times \mathbb{T})^*) \times \mathbb{T} \times \mathbb{T}^* \times (\mathbb{R}^* \times \mathbb{B}) \times \mathbf{B} \to \mathbb{R}^* \times \mathbb{B}$

$$\begin{aligned} &\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, (G, \text{failed}), c) = \\ &\quad (\text{commitUpdates}(G, \text{tgt}, c, \text{reduceMultiArrows}(\text{covers})),\ \text{failed and covers} = [::]) \\ &\qquad \text{for } p = \text{map}(\lambda ms.\texttt{Cover}\ (\text{map}(\lambda m.(m, \text{filter}(\lambda B.m.2 \leq B, \text{toCover})), ms))\ \text{toCover}, \\ &\qquad\qquad\qquad \Gamma'(c)) \text{ and} \\ &\qquad\quad ([::], p) \rightsquigarrow_* (\text{covers}, [::]) \end{aligned}$$

- Function inhabit_cover : $(\mathbf{B} \to (\mathbb{T}^* \times \mathbb{T})^*) \times \mathbb{R}^* \times \mathbb{T} \to \mathbb{B} \times \mathbb{R}^*$

$$\text{inhabit\_cover}(\Gamma', G, \text{tgt}) =$$
$$\begin{cases} (\text{false}, \text{G} +\!\!+ \text{G}') \\ \quad \text{for } (\text{false}, G') = \text{foldl}(\lambda s\ c.\text{accumulateCovers}(\Gamma', \text{tgt}, \text{primeFactors}(\text{tgt}), s, c), \\ \qquad\qquad\qquad\qquad ([::], \text{true}), \\ \qquad\qquad\qquad\qquad \text{enum}(\mathbf{B})) \\ (\text{true}, G) \qquad \text{otherwise} \end{cases}$$

- Function $G_\omega : \mathbb{T} \to \mathbb{R}^*$

$$G_\omega(A) = [::\, A \mapsto @(A, A) \& \operatorname{map}(\lambda c.A \mapsto c, \operatorname{enum}(\mathbf{B}))]$$

- The machine state transition function

$$\begin{aligned}
&\text{inhabitation_step} : (\mathbf{B} \to (\mathbb{T}^* \times \mathbb{T})^*) \times \mathbb{R}^* \times \mathbb{R}^* \to \mathbb{R}^* \times \mathbb{R}^* \\
&\text{inhabitation_step}(\Gamma', G_{\text{stable}}, G_{\text{targets}}) = \\
&\left\{\begin{array}{l}
① \left\{\begin{array}{l}
(G_{\text{stable}}, G'_{\text{targets}}) \\
\quad \text{for } A \mapsto c \text{ in } G_{\text{stable}} \\
([::\, A \mapsto c \& G_{\text{stable}}], G'_{\text{targets}}) \\
\quad \text{for not } A \mapsto c \text{ in } G_{\text{stable}}
\end{array}\right. \\
\quad \text{for } G_{\text{targets}} = [::\, A \mapsto c \& G'_{\text{targets}}] \\
② \left\{\begin{array}{l}
(G_{\text{stable}}, G'_{\text{targets}}) \\
\quad \text{for } A \mapsto @(B, \text{tgt}) \text{ in } G_{\text{stable}} \\
③ \left\{\begin{array}{l}
([::\, \text{tgt} \mapsto \bot \& G_{\text{stable}}], \text{dropTargets}(G'_{\text{targets}})) \\
\quad \text{for computeFailExisting}(G_{\text{stable}}, \text{tgt}) = (\text{true}, \text{false}) \\
(G_{\text{stable}}, \text{dropTargets}(G'_{\text{targets}})) \\
\quad \text{for computeFailExisting}(G_{\text{stable}}, \text{tgt}) = (\text{true}, \text{true}) \\
([::\, A \mapsto @(B, \text{tgt}) \& G_{\text{stable}}], G'_{\text{targets}}) \\
\quad \text{for computeFailExisting}(G_{\text{stable}}, \text{tgt}) = (\text{false}, \text{true}) \\
④ \left\{\begin{array}{l}
([::\, A \mapsto @(B, \text{tgt}) \& G_\omega(\text{tgt}) +\!\!+ G_{\text{stable}}], G'_{\text{targets}}) \\
\quad \text{for isOmega}(\text{tgt}) \\
⑤ \left\{\begin{array}{l}
([::\, \text{tgt} \mapsto \bot \& G_{\text{stable}}], \text{dropTargets}(G'_{\text{targets}})) \\
\quad \text{for inhabit_cover}(\Gamma', G'_{\text{targets}}, \text{tgt}) = (\text{true}, G''_{\text{targets}}) \\
([::\, A \mapsto @(B, \text{tgt}) \& G_{\text{stable}}], G''_{\text{targets}}) \\
\quad \text{for inhabit_cover}(\Gamma', G'_{\text{targets}}, \text{tgt}) = (\text{false}, G''_{\text{targets}})
\end{array}\right. \\
\quad \text{for not isOmega}(\text{tgt})
\end{array}\right. \\
\quad \text{for computeFailExisting}(G_{\text{stable}}, \text{tgt}) = (\text{false}, \text{false})
\end{array}\right. \\
\quad \text{for not } A \mapsto @(B, \text{tgt}) \text{ in } G_{\text{stable}}
\end{array}\right. \\
\quad \text{for } G_{\text{targets}} = [::\, A \mapsto @(B, \text{tgt}) \& G'_{\text{targets}}] \\
⑥ (G_{\text{stable}}, \text{dropTargets}(G'_{\text{targets}})) \\
\quad \text{for } G_{\text{targets}} = [::\, A \mapsto \bot \& G'_{\text{targets}}] \\
⑦ (G_{\text{stable}}, [::]) \\
\quad \text{for } G_{\text{targets}} = [::]
\end{array}\right.
\end{aligned}$$

Within preprocessed context Γ' the inhabitation machine transforms lists of rules into lists of rules . Its behavior is specified by the step relation induced by inhabitation_step:

$$\frac{}{\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow \text{inhabitation\_step}(\Gamma', G_{\text{stable}}, G_{\text{targets}})}$$

The n-step closure of $\rightsquigarrow$ is the least relation closed under the rules:

$$\frac{}{\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_0 (G_{\text{stable}}, G_{\text{targets}})}$$

$$\frac{\boxed{1}\ \Gamma' \vdash (G^1_{\text{stable}}, G^1_{\text{targets}}) \rightsquigarrow (G^2_{\text{stable}}, G^2_{\text{targets}}) \qquad \boxed{2}\ \Gamma' \vdash (G^2_{\text{stable}}, G^2_{\text{targets}}) \rightsquigarrow_n (G^3_{\text{stable}}, G^3_{\text{targets}})}{\Gamma' \vdash (G^1_{\text{stable}}, G^1_{\text{targets}}) \rightsquigarrow_{n+1} (G^3_{\text{stable}}, G^3_{\text{targets}})}$$

By definition the n-step closure coincides with transtive reflexive closure:

$$\begin{gathered}\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_* (G'_{\text{stable}}, G'_{\text{targets}}) \text{ iff} \\ \text{there exists n, s.t. } \Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_n (G'_{\text{stable}}, G'_{\text{targets}}).\end{gathered}$$

In proof trees t for $\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_{n+1} (G'_{\text{stable}}, G'_{\text{targets}})$ the first and second premise are selected using $t.1$ and $t.2$. □

The inhabitation machine with all of its helper functions may seem daunting at first, but its core idea is simple. Each step transforms two rule lists. The first list G_{stable} represents rules of the tree grammar under construction. It is stable because it will only grow by addition of new rules, while its contents will never be deleted or modified. The second list G_{targets} is a queue of rules under consideration for inclusion into the stable list. When there are no more entries in G_{targets} inhabitation stops and G_{stable} is completed (case ⑦). Then the resulting grammar is $G = (A, \mathbb{T}, \{@\} \uplus \mathbf{B}, R)$ for rules $R = \{r \mid r \text{ is in } G_{\text{stable}} \text{ and not exists } A \text{ s.t. } r = A \mapsto \bot\}$ and the request type A from $\Gamma \vdash ? : A$. There are three options for rules $r \in \mathbb{R}$. Nonterminals are types $\mathbb{T}$ and used without subtyping. Terminals are combinators **B** or the apply symbol "@". Rules always use combinators without arguments and the apply symbol with two arguments. List G_{stable} can also include rules of form $A \mapsto \bot$, which are not considered for the resulting grammar. They encode the absence of words (inhabitants) for nonterminal (type) A and allow the algorithm to abort inhabitation early in certain situations. In each step, the machine makes a decision based on the first rule in G_{targets}. The cases grouped under ① describe how to add a combinator rule, which is just copied to the stable grammar if it is not yet present. The target rule queue G_{targets} has an additional internal order. If the cover machine finds multi arrow $m = ([::A_2\&[::A_1]], B)$ for target B and combinator c, then G_{targets} includes $[::A_1 \to A_2 \to B \mapsto c\&[::A_2 \to B \mapsto @(A_1 \to A_2 \to B, A_1)\&[::B \mapsto @(A_2 \to B, A_2)\&G'_{\text{targets}}]]]$ where G'_{targets} is either empty or starts with a combinator rule of shape $C \mapsto d$. Whenever the

machine detects that an intermediate nonterminal (type) such as A_2 is unproductive (has no inhabitants), all of the targets up to the next combinator are discarded using function dropTargets (case ⑦ and multiple subcases of ②). In the example, this aborts inhabitation of B using m and thereby avoids redundant work for A_1. If the first rule on the target queue has shape $A \mapsto @(B, \text{tgt})$ it is an application rule and processed by case ②. Again, rules which have already been processed are skipped. Otherwise (case ③) computeFailExisting(G_{stable}, tgt) analyzes if enough information has been collected about the next target tgt in G_{stable} to continue without extra effort. Function computeFailExisting(G_{stable}, tgt) returns a tuple of Booleans. The first Boolean indicates if type tgt or a greater or equal type has been marked as uninhabited. By FCL rule ($\leq$) type tgt cannot have inhabitants in this case, because they would also be inhabitants of the marked type. The second Boolean is true if tgt has previously been used in target position or been directly marked as unproductive. In case ③ unproductive target types are discarded with the rest of the entries from their multi arrows, and they are marked unproductive in G_{stable} unless they have been directly marked before. If tgt has been previously used and not marked, the current rule $A \mapsto @(B, \text{tgt})$ is added to G_{stable} and no further targets are generated. Otherwise, in case ④ the machine checks if ω is a subtype of tgt. Remember that the cover machine from Definition 21 only works correctly with inputs not greater or equal than ω (Lemma 27.2, Lemma 27.4). Every combinator c can inhabit $\Gamma(c) \leq \omega$ and therefore by rule ($\leq$) also tgt. Additionally, the subtype rules can derive $\text{tgt} \leq \omega \leq \omega \to \omega$ resulting in inhabitants of tgt being applicable to other inhabitants of type tgt. The rules in $G_\omega(\text{tgt})$ express both of these facts, which is why they and the current rule are added to G_{stable}. Every applicative term is derivable by $G_\omega(\text{tgt})$ and no additional targets are required. Finally, in case ⑤ the cover machine is invoked to compute additional inhabitation targets. Functions inhabit_cover and accumulateCovers generate proper cover machine calls (c.f. Lemma 27) for the split multi arrows of every combinator, which are precomputed in Γ'. Results of the cover machine are postprocessed (c.f. Definition 27) and converted for future addition to G_{targets} using commitUpdates and commitMultiArrow. Function commitUpdates adds all cover machine results for one combinator and commitMultiArrow adds a single result multi arrow in the order described above. Both inhabit_cover, and accumulateCovers also keep track of a Boolean to indicate if no cover machine call yielded any results for any combinator. If this is the case, no inhabitants for tgt can be found. The inhabitation machine then marks tgt as unproductive and again discards entries from the target queue. Otherwise, the current rule is added to G_{stable} and the newly computed targets are enqueued. Note that using a stack instead of a queue would lead to depth first instead of breadth first search.

Correctness of the inhabitation machine requires a fairly complex proof presented next. There are two main sources of complexity. The first is the requirement to reuse information from G_{stable} to avoid loops caused by cyclic grammar entries (e.g. from type $A \to A$). The other complex part is reasoning about the partially constructed list G_{stable}, taking into account potential future inhabitants from G_{targets} which can grow or shrink in any step, potentially even discarding multiple scheduled targets at once. The easiest part, soundness, is proven first. Again, the relational machine specification enables reasoning about soundness, completeness,

and termination independently. Also, the initial call to the machine can be constructed after preconditions for its correct execution have been established.

For the following proofs, membership in the language of the grammar computed by the inhabitation machine is replaced by a Boolean predicate. This avoids the clumsy notation to distinguish between the grammar and its rules.

Definition 30 (Boolean Predicate deciding Language Membership) Let $G \in \mathbb{R}^*$ be rules computed by the subtype machine for some requested type $B \in \mathbb{T}$. Then for all types $A \in \mathbb{T}$, and terms $M \in \mathbb{A}$ the Boolean predicate $\text{word} : \mathbb{R}^* \times \mathbb{T} \times \mathbb{A} \to \mathbb{B}$ is by definition the same as membership in the computed grammar, i.e. $\text{word}(G, A, M)$ iff $M \in L((B, \mathbb{T}, \{@\} \uplus \mathbf{B}, \{r \mid r \text{ is in } G \text{ and not exists } C \text{ s.t. } r = C \to \bot\}, A))$:

$$\text{word}(G, A, M) = \begin{cases} \text{true} & \text{for } M = c \text{ and } A \mapsto c \text{ is in } G \\ \text{true} & \text{for } M = @(M_1, M2) \text{ and exists } A \mapsto @(B, C) \text{ in } G \text{ s.t.} \\ & \qquad \text{word}(G, B, M_1) \text{ and } \text{word}(G, C, M_2) \\ \text{false} & \text{otherwise} \end{cases}$$

Lemma 29 (Inhabitation Machine Soundness) *For context* $\Gamma : \mathbf{B} \to \mathbb{T}$ *define the Boolean predicate* $\text{FCL\_sound} : \mathbb{R}^* \to \mathbb{B}$ *by*

$$\text{FCL\_sound}(G) = \textit{for all } r \textit{ in } G: \begin{cases} A \leq \Gamma(c) & \textit{for } r = A \mapsto c \\ B \leq C \to A & \textit{for } r = A \mapsto @(B, C) \\ \textit{true} & \textit{otherwise} \end{cases}$$

then the following statements are true for all $G, G_1, G_2, G_{\text{stable}}, G_{\text{targets}}, G'_{\text{stable}}, G'_{\text{targets}} \in \mathbb{R}^*$*, and* $A, B, C, \text{tgt} \in \mathbb{T}$, $\text{srcs} \in \mathbb{T}^*$, $ms \in (\mathbb{T}^* \times \mathbb{T})^*$, $c \in \mathbf{B}$, $M \in \mathbb{A}$, *and* $b \in \mathbb{B}$*:*

1. *If* $\text{FCL\_sound}(G)$ *and* $\text{word}(G, A, M)$ *then* $\Gamma \vdash M : A$
2. $\text{suffix}(\text{dropTargets}(G), G)$
3. *If* $\text{suffix}(G_1, G_2)$ *and* $\text{word}(G_1, A, M)$ *then* $\text{word}(G_2, A, M)$
4. *If* $\text{suffix}(G_1, G_2)$ *and* $\text{FCL\_sound}(G_2)$ *then* $\text{FCL\_sound}(G_1)$
5. *If* $\text{FCL\_sound}(G_1)$ *and* $\text{FCL\_sound}(G_2)$ *then* $\text{FCL\_sound}(G_1 +\!\!+ G_2)$
6. *If* $\text{FCL\_sound}(G)$ *and* $\Gamma(c) \leq \text{mkArrow}((\text{srcs}, \text{tgt}))$ *and* $\text{tgt} \leq A$ *then* $\text{FCL\_sound}(\text{commitMultiArrow}(G, c, \text{srcs}, A)$

7. *If* FCL_sound(G) *and*
   for all m *in* ms:
   $\Gamma(c) \leq \text{mkArrow}(m)$ *and*
   $m.2 \leq A$ *then*
   FCL_sound(commitUpdates(G, A, c, ms)

Additionally in the preprocessed context $\Gamma' : \mathbf{B} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$ *from Definition 29:*

8. *If* FCL_sound(G) *and not* isOmega(A) *then*
   FCL_sound(accumulateCovers($\Gamma', A,$ primeFactors(A), $(G, b), c$).1)

9. *If* FCL_sound(G) *and not* isOmega(A) *then*
   FCL_sound(
   (foldl($\lambda\ s\ c.$ accumulateCovers($\Gamma', A,$ primeFactors(A), s, c), (G, b), enum($\mathbf{B}$))).1)

10. *If* FCL_sound(G) *and not* isOmega(A) *then*
    FCL_sound((inhabit_cover(Γ', G, A)).2)

11. *If* isOmega(A) *then* FCL_sound($G_\omega(A)$)

12. *If* FCL_sound(G_{stable}) *and* FCL_sound(G_{targets}) *then*
    FCL_sound((inhabitation_step($\Gamma', G_{\text{stable}}, G_{\text{targets}}$)).1) *and*
    FCL_sound((inhabitation_step($\Gamma', G_{\text{stable}}, G_{\text{targets}}$)).2)

13. *If* FCL_sound(G_{stable}) *and* FCL_sound(G_{targets}) *and*
    $\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_* (G'_{\text{stable}}, G'_{\text{targets}})$ *then*
    FCL_sound((inhabitation_step($\Gamma', G'_{\text{stable}}, G'_{\text{targets}}$)).1) *and*
    FCL_sound((inhabitation_step($\Gamma', G'_{\text{stable}}, G'_{\text{targets}}$)).2)

PROOF The crucial trick is to define FCL_sound and to prove proposition 1 by induction on M. This predicate is term-independent and statements 2 - 5, which are easy to prove, enable compositional reasoning for all the intermediate rule lists created in the other statements. These cover the different decisions made by inhabitation_step and in the order presented their proofs follow by unsurprising case-analysis or induction. ■

A total function for the inhabitation machine step relation is constructed next.

Lemma 30 (Functionality of the Inhabitation Machine)
For all preprocessed contexts $\Gamma' : \mathbf{B} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$ *and rule lists* $G^1_{\text{stable}}, G^1_{\text{targets}}, G^2_{\text{stable}}, G^2_{\text{targets}}, G^3_{\text{stable}}, G^3_{\text{targets}} \in \mathbb{R}^*$ *the step relation is functional:*

If $(G^1_{\text{stable}}, G^1_{\text{targets}}) \rightsquigarrow (G^2_{\text{stable}}, G^2_{\text{targets}})$ *and* $(G^1_{\text{stable}}, G^1_{\text{targets}}) \rightsquigarrow (G^3_{\text{stable}}, G^3_{\text{targets}})$ *then*
$(G^2_{\text{stable}}, G^2_{\text{targets}}) = (G^3_{\text{stable}}, G^3_{\text{targets}})$

PROOF Direct consequence of the definition via inhabitation_step. ■

Definition 31 (Inhabitation Machine Termination Certificate)
For all preprocessed contexts $\Gamma' : \mathbf{B} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$ and input rule lists $G_{\text{stable}}, G_{\text{targets}} \in \mathbb{R}^*$ termination certificate $\text{Dom}_{\Gamma'}(G_{\text{stable}}, G_{\text{targets}})$ is a proof tree constructed from the rules of $\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_* (G'_{\text{stable}}, [::])$ for some $G'_{\text{stable}} \in \mathbb{R}^*$. □

Just as for the cover machine (c.f. Definition 24) a tail recursive procedure can now be stated as implementation of the inhabitation machine.

Definition 32 (Procedure implementing the Inhabitation Machine)

$$\begin{aligned}
&\text{inhabitationMachine}(\Gamma', G_{\text{stable}}, G_{\text{targets}}) : \\
&\quad \text{Dom}_{\Gamma'}(G_{\text{stable}}, G_{\text{targets}}) \to \{G \in \mathbb{R}^* \mid \Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_* (G, [::])\} \\
&\text{inhabitationMachine}(\Gamma', G_{\text{stable}}, G_{\text{targets}})(d) = \\
&\begin{cases} G & \text{for } G_{\text{targets}} = [::] \\ \text{inhabitationMachine}(\Gamma', G'_{\text{stable}}, G'_{\text{targets}})(d.2) & \\ \quad \text{for inhabitation_step}(\Gamma', G_{\text{stable}}, G_{\text{targets}}) = (G'_{\text{stable}}, G'_{\text{targets}}) & \end{cases}
\end{aligned}$$

□

Termination certificates $d \in \text{Dom}_{\Gamma'}(G_{\text{stable}}, G_{\text{targets}})$ always exist because the growth of G_{stable} is limited.

Lemma 31 (Totality of the Inhabitation Machine) *For context* $\Gamma : \mathbf{B} \to \mathbb{T}$ *and the preprocessed context* $\Gamma' : \mathbf{B} \to ((\mathbb{T}^* \times \mathbb{T})^*)^*$ *define*

- *A projection to the parameter types of rules by function* parameterTypes : $\mathbb{R}^* \to \mathbb{T}^*$

$$\text{parameterTypes}(G) = \text{pmap}\left(\lambda r. \begin{cases} \text{Some}(C) & \textit{for } r = A \mapsto @(B, C) \\ \text{None} & \textit{otherwise} \end{cases}, G\right)$$

- *All types in rules generated for a multi arrow via* grammarTypes : $\mathbb{T}^* \times \mathbb{T} \to \mathbb{T}^*$

$$\begin{aligned}
&\text{grammarTypes}(\text{srcs}, \text{tgt}) = \\
&\begin{cases} [::\ \text{src}\&[::\ \text{tgt}\&\text{grammarTypes}(\text{srcs}', \text{src} \to \text{tgt})]] & \textit{for } \text{srcs} = [::\ \text{src}\&\text{srcs}'] \\ \text{tgt} & \textit{otherwise} \end{cases}
\end{aligned}$$

- *All possible parameter types for an initial target by* maxParameterTypes : $\mathbb{T} \to \mathbb{T}^*$

$$\begin{aligned}
&\text{maxParameterTypes}(A) = \\
&\quad [::\ A\&\text{flatten}(\text{map}(\lambda c.\text{flatten}(\text{map}(\lambda ms.\text{flatten}(\text{map}(\lambda m. \\
&\qquad \text{grammarTypes}(m) +\!\!+ \text{grammarTypes}(m.1, A), \\
&\qquad \text{filterMergeMultiArrows}(\text{subseqs}(ms)))), \Gamma'(c))), \text{enum}(\mathbf{B})))]
\end{aligned}$$

Then for all $r \in \mathbb{R}$*, and* $G, G_{\text{stable}}, G_{\text{targets}} \in \mathbb{R}^*$*,* $c \in \mathbf{B}$*, and* $A, \text{src}, \text{tgt}, \text{tgt}' \in \mathbb{T}$*, and* $\text{srcs}, \Delta, \text{toCover} \in \mathbb{T}^*$*,* $ms \in (\mathbb{T}^* \times \mathbb{T})^*$*,* $cs \in \mathbf{B}^*$*,* $b \in \mathbb{B}$*, and* $n \in \mathbb{N}$ *the following statements are true:*

1. *If* src *is in* srcs *then* src *is in* grammarTypes(srcs, tgt)

2. mkArrow(take(n, srcs), tgt) *is in* grammarTypes(srcs, tgt)

3. *If for all* A *in* undup(parameterTypes(G))*:* A *is in* Δ *and*
   for all A *in* srcs*:* A *is in* Δ *then*
   for all A *in* undup(parameterTypes(commitMultiArrow(G, c, srcs, tgt)))*:* A *is in* Δ

4. *If for all* A *in* undup(parameterTypes(G))*:* A *is in* Δ *and*
   for all A *in* srcs*:* A *is in* Δ *and*
   for all m *in* ms*: for all* src *in* m.1*:* src *is in* Δ *then*
   for all A *in* undup(parameterTypes(commitUpdates(G, tgt, c, ms)))*:* A *is in* Δ

5. *If for all* A *in* undup(parameterTypes(G))*:* A *is in* maxParameterTypes(tgt) *then*
   for all A *in* (accumulateCovers(Γ', tgt, primeFactors(tgt), $(G, b), c$)).1*:*
   A *is in* maxParameterTypes(tgt)

6. tgt *is in* maxParameterTypes(tgt)

7. *If for all* A *in* undup(parameterTypes(G))*:* A *is in* maxParameterTypes(tgt) *then*
   for all A *in*
   undup(parameterTypes(
   (foldl($\lambda\ s\ c$.accumulateCovers(Γ', tgt$'$, primeFactors(tgt$'$), s, c), (G, b), enum($\mathbf{B}$))).1)) :
   A *is in* maxParameterTypes(tgt)

8. *If for all* A *in* undup(parameterTypes(G))*:* A *is in* maxParameterTypes(tgt) *then*
   for all A *in* undup(parameterTypes((inhabitCover(Γ', G, tgt)).2))*:*
   A *is in* maxParameterTypes(tgt)

9. *For all* A *in* parameterTypes(G_ω(tgt))*:* A = tgt

10. *If* tgt$'$ *is in* maxParameterTypes(tgt) *then*
    for all A *in* undup(parameterTypes(G_ω(tgt$'$)))*:* A *is in* maxParameterTypes(tgt)

11. *If for all* A *in* undup(parameterTypes(G_{targets}))*:* A *is in* maxParameterTypes(tgt) *and*
    for all A *in* undup(parameterTypes(G_{stable}))*:* A *is in* maxParameterTypes(tgt) *then*
    for all A *in* undup(parameterTypes((inhabitation_step($\Gamma', G_{\text{stable}}, G_{\text{targets}}$)).1)) :
    A *is in* maxParameterTypes(tgt)
    and for all A *in* undup(parameterTypes((inhabitation_step($\Gamma', G_{\text{stable}}, G_{\text{targets}}$)).2)) :
    A *is in* maxParameterTypes(tgt)

Define the measured lexicographic product $\text{inhabit\_step\_rel}_{\text{tgt}} \subset (\mathbb{R}^* \times \mathbb{R}^*) \times (\mathbb{R}^* \times \mathbb{R}^*)$ *as the least relation closed under the rules:*

$$\frac{\begin{array}{c}\text{lsize}(\text{maxParameterTypes}(\text{tgt})) + 1 - \text{lsize}(\text{undup}(\text{parameterTypes}(G_{\text{stable}}))) < \\ \text{lsize}(\text{maxParameterTypes}(\text{tgt})) + 1 - \text{lsize}(\text{undup}(\text{parameterTypes}(G'_{\text{stable}})))\end{array}}{\text{inhabit\_step\_rel}_{\text{tgt}}((G_{\text{stable}}, G_{\text{targets}}), (G'_{\text{stable}}, G'_{\text{targets}}))}$$

$$\frac{\begin{array}{c}\text{undup}(\text{parameterTypes}(G_{\text{stable}})) = \text{undup}(\text{parameterTypes}(G'_{\text{stable}})) \\ \text{lsize}(\text{undup}(\text{parameterTypes}(G_{\text{targets}}))) < \text{lsize}(\text{undup}(\text{parameterTypes}(G'_{\text{targets}})))\end{array}}{\text{inhabit\_step\_rel}_{\text{tgt}}((G_{\text{stable}}, G_{\text{targets}}), (G'_{\text{stable}}, G'_{\text{targets}}))}$$

Also assert that relation $\text{R}(x, y)$ *is well-founded [154], written* $\text{WF}(\text{R})$*, if all* x *are accessible, written* $\text{Acc}(x)$*, where* $\text{Acc}(x)$ *is the least set closed under the rule:*

$$\frac{\textit{for all } y\textit{: if } \text{R}(y, x) \textit{ then } \text{Acc}(y)}{\text{Acc}(x)}$$

Continue with the proof of the following statements:

12. $\text{WF}(\text{inhabit\_step\_rel}_{\text{tgt}})$
13. *If* $\text{inhabit\_step\_rel}_{\text{tgt}}((G_{\text{stable}}, G_{\text{targets}}), (G'_{\text{stable}}, G'_{\text{targets}}))$ *and for all* A *in* $\text{undup}(\text{parameterTypes}(G_{\text{stable}}))$*:* A *is in* $\text{maxParameterTypes}(\text{tgt})$ *then* $\text{inhabit\_step\_rel}_{\text{tgt}}(([::\, r \& G_{\text{stable}}], G_{\text{targets}}), (G'_{\text{stable}}, G'_{\text{targets}}))$
14. $\text{lsize}(\text{dropTargets}(G_{\text{targets}})) \leq \text{lsize}(G_{\text{targets}})$
15. *If* $ms = [::]$ *then* $G_{\text{targets}} = \text{commitUpdates}(G_{\text{targets}}, \text{tgt}, c, ms)$
16. *If* $ms = [::]$ *then* $\text{reduceMultiArrows}(ms) = [::]$
17. *If* $(\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, (G, \textit{true}), c)).2$ *then* $G = (\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, (G, \textit{true}), c)).1$
18. *If* $(\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, (G, b), c)).2$ *then* $b = \textit{true}$
19. *If* $(\text{foldl}(\lambda s\ c.\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, s, c), (G, b), cs)).2$ *then* $b = \textit{true}$
20. *If* $(\text{foldl}(\lambda s\ c.\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, s, c), (G, b), cs)).2$ *then* $G = (\text{foldl}(\lambda s\ c.\text{accumulateCovers}(\Gamma', \text{tgt}, \text{toCover}, s, c), (G, b), cs)).1$
21. *If* $(\text{inhabit\_cover}(\Gamma', G_{\text{targets}}, \text{tgt})).1$ *then* $(\text{inhabit\_cover}(\Gamma', G_{\text{targets}}, \text{tgt})).2 = G_{\text{targets}}$
22. *If not* $(\text{computeFailExisting}(G, A)).1$ *and not* $(\text{computeFailExisting}(G, A)).2$ *then* A *is not in* $\text{parameterTypes}(G)$

23. *If for all A in* undup(parameterTypes(G_{stable}))*: A is in* maxParameterTypes(tgt) *and for all A in* undup(parameterTypes(G_{targets}))*: A is in* maxParameterTypes(tgt) *and not* $G_{\text{targets}} = [::]$ *then*
inhabit_step_rel$_{\text{tgt}}$(inhabitation_step($\Gamma', G_{\text{stable}}, G_{\text{targets}}$), ($G_{\text{stable}}, G_{\text{targets}}$))

24. *A finite proof tree* $\text{Dom}_{\Gamma'}([::], (\text{inhabit\_cover}(\Gamma', [::], \text{tgt})).2)$ *can always be constructed*

PROOF The first part establishes the maximal list of parameter types that will ever occur in position C in rules of shape $A \mapsto @(B, C)$. These will always stem from the filtered and merged combination of some multi arrows drawn from Γ', possibly with their targets replaced by the initial request type. Function grammarTypes allows all sources and targets of these multi arrows in the generated set. Statements 1–11 establish the properties of the auxiliary inhabitation machine functions with respect to the maximal parameters. Their proofs are by straight-forward induction and/or case analysis. Statement 11 is the most important part and shown by case analysis on the options considered by the inhabitation machine transition function inhabitation_step.
Well-foundedness from statement 11 is used to construct the proof tree in statement 24. With the rule for Acc well-foundedness generates the induction principle

If for all x:
  if for all $y : R(y, x)$ implies $P(y)$
  then $P(x)$
then forall $x : P(x)$

where P is chosen to be $\text{Dom}_{\Gamma'}$ and statement 23 satisfies the precondition $R(y, x)$ for constructing the tree via the induction hypothesis. Statement 11 uses that inhabit_step_rel is the lexicographic product of two well founded relations (less than on natural numbers), which is again well-founded [154]. Statements 13–22 break down the proof of 11, which is by case-analysis on the options of inhabitation_step, into easy to show steps. ■

The initial remark about the upper bound on the number of machine steps and thereby the size of the grammar can also be clarified further at this point. It is easy to show that the number of multi arrows in $\Gamma'(c) = \text{splitTy}(\Gamma(c))$ is bound by $\text{size}(A) \cdot \text{length}(A)$. Function subseqs computes exponentially many subsequences. The subsequences are passed to filterMergeMultiArrows, which invokes mergeMultiArrows for each subsequence. Function mergeMultiArrows calls mergeMultiArrow linear many times, which then combines multi arrows in polynomial time of their size using the quadratic subtyping algorithm. Results are passed to grammarTypes and processed in linear time. Therefore maxParameterTypes(A) is overall in $2^{p(\text{size}(A) \cdot \text{length}(A))}$ for some polynomial p. Relation inhabit_step_rel$_{\text{tgt}}$ bounds the number of possible steps by $(\text{lsize}(\text{maxParameterTypes}(A)) + 1)^2$ because Lemma 31.11 places all parameters in G_{stable} and G_{targets} within maxParameterTypes(A). There is only one call to the cover machine per step and its arguments are polynomial in the input size, hence the overall single-exponential running time of the algorithm.

The final part of the correctness proof, completeness, requires two additional invariants.

Lemma 32 (Inhabitation Machine Invariants) *For context* $\Gamma : \mathbf{B} \to \mathbb{T}$*, a list of rules* $G \in \mathbb{R}^*$

- *is soundly marked for failures,* $\mathrm{FailSound}_\Gamma(G)$*, if for all* $A \in \mathbb{T}$ *s.t. there exists* $A \mapsto \bot$ *in* G *there does not exist* $M \in \mathbb{A}$ *s.t.* $\Gamma \vdash M : A$
- *is a target list without fail rules,* $\mathrm{noTargetFailures}(G)$*, if there does not exist* $A \in \mathbb{T}$ *s.t. there exists* $A \mapsto \bot$ *in* G

Additionally for the preprocessed context $\Gamma'(c) = \mathrm{splitTy}(\Gamma(c))$ *as well as all* $G_1, G_2, G_{\mathrm{stable}}, G_{\mathrm{targets}}, G'_{\mathrm{targets}} \in \mathbb{R}^*$, $c \in \mathbf{B}$, $\mathrm{srcs} \in \mathbb{T}^*$, *and* $A, \mathrm{tgt} \in \mathbb{T}$ *the following statements are true:*

1. *If* $\mathrm{FailSound}_\Gamma(G_1)$ *and* $\mathrm{FailSound}_\Gamma(G_2)$ *then* $\mathrm{FailSound}_\Gamma(G_1 +\!\!+ G_2)$
2. *If* $\mathrm{FailSound}_\Gamma(G_1 +\!\!+ G_2)$ *then* $\mathrm{FailSound}_\Gamma(G_1)$ *and* $\mathrm{FailSound}_\Gamma(G_2)$
3. *If* $\mathrm{FailSound}_\Gamma(G)$ *and* $(\mathrm{computeFailExisting}(G, A)).1 = \mathit{true}$ *then* $\mathrm{FailSound}_\Gamma([:: A \mapsto \bot \& G])$
4. *If* $(\mathrm{accumulateCovers}(\Gamma', \mathrm{tgt}, \mathrm{primeFactors}(\mathrm{tgt}), (G, \mathit{true}), c)).2 = \mathit{true}$ *and* *not* $\mathrm{isOmega}(\mathrm{tgt})$ *then* *not* $\Gamma \vdash c : \mathrm{mkArrow}(\mathrm{srcs}, \mathrm{tgt})$
5. *If* $(\mathrm{inhabit\_cover}(\Gamma', G, \mathrm{tgt})).1 = \mathit{true}$ *and not* $\mathrm{isOmega}(\mathrm{tgt})$ *then* *not exists* $M \in \mathbb{A}$ *s.t.* $\Gamma \vdash M : \mathrm{tgt}$
6. $\mathrm{FailSound}_\Gamma(G_\omega(A))$
7. *If* $\mathrm{FailSound}_\Gamma(G_{\mathrm{stable}})$ *then* $\mathrm{FailSound}_\Gamma((\mathrm{inhabitation\_step}(\Gamma', G_{\mathrm{stable}}, G_{\mathrm{targets}})).1)$
8. *If* $\mathrm{noTargetFailures}(G_{\mathrm{targets}})$ *and* $\mathrm{suffix}(G'_{\mathrm{targets}}, G_{\mathrm{targets}})$ *then* $\mathrm{noTargetFailures}(G'_{\mathrm{targets}})$
9. *If* $\mathrm{noTargetFailures}(G_{\mathrm{targets}})$ *then* $\mathrm{noTargetFailures}((\mathrm{inhabit\_cover}(\Gamma', G_{\mathrm{targets}}, \mathrm{tgt})).2)$
10. *If* $\mathrm{noTargetFailures}(G_{\mathrm{targets}})$ *then* $\mathrm{noTargetFailures}((\mathrm{inhabitation\_step}(\Gamma', G_{\mathrm{stable}}, G_{\mathrm{targets}})).2)$

PROOF Statements 1 and 2 follow from the definition of FailSound, while 3 is shown by induction on G and case analysis of the first rule. The proof for statement 4 uses that $\Gamma(c)$ is the minimal type of c (Lemma 16), and completeness of the cover machine (Lemma 24) after unfolding the definition of accumulateCovers (Definition 29). Statement 5 is shown by inversion of $\Gamma \vdash M : \mathrm{tgt}$ using Lemma 15.2 and induction on the finite combinator set $\mathbf{B}$. Statement 6 follows from the definition of $G_\omega(A)$. Statement 7, sound failure marking, follows by case analysis on the first rule in G_{targets} in combination with the previous statements. Statement 8 follows from the definition of noTargetFailures, statement 9 by unfolding the definition of inhabit_cover and induction on the finite combinator set $\mathbf{B}$, and then by use of statement 10 in the case analysis of the first rule of G_{targets} and the prior statements. ∎

Lemma 33 (Inhabitation Machine Completeness) *For context* $\Gamma : \mathbf{B} \to \mathbb{T}$ *define:*

- *Function* $\mathrm{updateGroups} : (\mathbb{R}^*)^* \times \mathbb{R} \to (\mathbb{R}^*)^*$ *to group a single rule together with the rules of the multi arrow it was generated from:*

$$\mathrm{updateGroups}(\mathrm{groups}, r) = \begin{cases} [::\, [::A \mapsto c] \& \mathrm{groups}] & \textit{for } r = A \mapsto c \\ [::\, \mathrm{rcons}(g, r) \& \mathrm{groups}'] & \textit{for } \mathrm{groups} = [::\, g \& \mathrm{groups}'] \textit{ and not } r = A \mapsto c \\ [::[::r]] & \textit{for } \mathrm{groups} = [::] \textit{ and not } r = A \mapsto c \end{cases}$$

- *Function* $\mathrm{group} : \mathbb{R}^* \to (\mathbb{R}^*)^*$ *to group rules by the multi arrow they were generated from:*

$$\mathrm{group}(G) = \mathrm{rev}(\mathrm{foldl}(\lambda\ s\ r.\, \mathrm{updateGroups}(s, r), [::], G))$$

- *A projection to the target (left hand side) of a rule* $\mathrm{lhs} : \mathbb{R} \to \mathbb{T}$

$$\mathrm{lhs}(r) = A \textit{ for } r = A \mapsto \bot \textit{ or } r = A \mapsto c \textit{ or } A \mapsto @(B, C)$$

- *A projection to the targets of a list of rules* $\mathrm{targetTypes} : \mathbb{R}^* \to \mathbb{T}^*$

$$\mathrm{targetTypes}(G) = \mathrm{map}(\mathrm{lhs}, G)$$

- *Potential future derivability of a term as the least relation* $\mathrm{future\_word} \subseteq \mathbb{R}^* \times \mathbb{R}^* \times \mathbb{T} \times \mathbb{A}$ *closed under the rules:*

$$\frac{A \mapsto c \textit{ is in } G_{\mathrm{stable}}}{\mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, A, c)}$$

$$\frac{\begin{array}{ll} g \textit{ is in } \mathrm{group}(G_{\mathrm{targets}}) & \textit{for all } B \textit{ in } \mathrm{parameterTypes}(g)\textit{:} \\ A \mapsto c \textit{ is in } g & \textit{exists } M \textit{ s.t. } \Gamma \vdash M : B \end{array}}{\mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, A, c)}$$

$$\frac{\begin{array}{ll} & \mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, B, M) \\ A \mapsto @(B, C) \textit{ is in } G_{\mathrm{stable}} & \mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, C, N) \end{array}}{\mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, A, @(M, N))}$$

$$\frac{\begin{array}{ll} & \mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, B, M) \\ & \Gamma \vdash N : C \\ g \textit{ is in } \mathrm{group}(G_{\mathrm{targets}}) & \textit{for all } B \textit{ in } \mathrm{parameterTypes}(g)\textit{:} \\ A \mapsto @(B, C) \textit{ is in } g & \textit{exists } M \textit{ s.t. } \Gamma \vdash M : B \end{array}}{\mathrm{future\_word}(G_{\mathrm{stable}}, G_{\mathrm{targets}}, A, @(M, N))}$$

- *An intermediate machine state* $(G_{\text{stable}}, G_{\text{targets}}) \in \mathbb{R}^* \times \mathbb{R}^*$ *to be partially complete for initial type* $\text{tgt} \in \mathbb{T}$*, written* $\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, G_{\text{targets}})$*, if:*

$$\begin{aligned}
&\textit{for all } A \in \mathbb{T} \textit{ s.t. } A = \text{tgt} \textit{ or} \\
&\qquad A \textit{ is in } \text{parameterTypes}(G_{\text{stable}}) \textit{ or} \\
&\qquad A \textit{ is in } \text{targetTypes}(\text{pmap}(\lambda g.\,\text{ohead}(\text{rev}(g)), \text{group}(G_{\text{targets}}))) : \\
&\textit{for all } M \in \mathbb{A} : \Gamma \vdash M : A \textit{ implies } \text{future_word}(G_{\text{stable}}, G_{\text{targets}}, A, M)
\end{aligned}$$

Then for the preprocessed context $\Gamma'(c) = \text{splitTy}(\Gamma(c))$ *and all* $G, G_{\text{stable}}, G_{\text{targets}}, G'_{\text{stable}}, G'_{\text{targets}}, g, \text{nextTargets} \in \mathbb{R}^*$, $\text{groups} \in (\mathbb{R}^*)^*$, $r \in \mathbb{R}$, *and* $A, B, C, D, \text{tgt} \in \mathbb{T}$, $\text{srcs} \in \mathbb{T}^*$, $\text{covers}, ms \in (\mathbb{T}^* \times \mathbb{T})^*$, $c \in \mathbf{B}$, $cs \in \mathbf{B}^*$, $M \in \mathbb{A}$, $b \in \mathbb{B}$, $n \in \mathbb{N}$ *the following statements are true:*

1. $\text{flatten}(\text{group}(G)) = G$
2. $\text{updateGroups}([::], r) = [::[::r]]$
3. *If for all* r *in* G*: not exists* $A \in \mathbb{T}$ *and* $c \in \mathbf{B}$ *s.t.* $r = A \mapsto c$ *then*
   $\text{foldl}(\lambda\ s\ r.\,\text{updateGroups}(s, r), [::\ g\&\text{groups}], G) = [::\ g +\!\!+ G\&\text{groups}]$
4. $\text{foldl}(\lambda\ s\ r.\,\text{updateGroups}(s, c), \text{groups}, [::\ A \mapsto c\&G]) =$
   $\text{foldl}(\lambda\ s\ r.\,\text{updateGroups}(s, c), [::], [::\ A \mapsto c\&G]) +\!\!+ \text{groups}$
5. *If* $G_{\text{targets}} = G +\!\!+ \text{dropTargets}(G_{\text{targets}})$ *then*
   for all r *in* G*: not exists* $A \in \mathbb{T}$ *and* $c \in \mathbf{B}$ *s.t.* $r = A \mapsto c$
6. *Either* $\text{dropTargets}(G) = [::\ A \mapsto c\&G']$ *for some* $A \in \mathbb{T}$, $c \in \mathbf{B}$, *and* $G' \in \mathbb{R}^*$ *or*
   $\text{dropTargets}(G) = [::]$
7. *If* $G_{\text{targets}} = G +\!\!+ \text{dropTargets}(G_{\text{targets}})$ *then*
   $\text{group}([::\ r\&G_{\text{targets}}]) = [::\ [::\ r\&G]\&\text{group}(\text{dropTargets}(G_{\text{targets}}))]$
8. *If* $A = \text{tgt}$ *or* A *is in* $\text{parameterTypes}(G_{\text{stable}})$ *and*
   $\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, [::])$ *and* $\Gamma \vdash M : A$ *then*
   $\text{word}(G_{\text{stable}}, A, M)$
9. *If* $\text{future\_word}(G_{\text{stable}}, [::], A, M)$ *then* $\text{word}(G_{\text{stable}}, A, M)$
10. *If* $\text{future\_word}(G_{\text{stable}}, G_{\text{targets}}, A, M)$ *and for all* r *in* G_{stable}*:* r *is in* G'_{stable} *then*
    $\text{future\_word}(G'_{\text{stable}}, G_{\text{targets}}, A, M)$
11. *If* $A \mapsto c$ *is in* G_{stable} *and* $\text{future\_word}(G_{\text{stable}}, [::\ A \mapsto c\&G_{\text{targets}}], B, M)$ *then*
    $\text{future\_word}(G_{\text{stable}}, G_{\text{targets}}, B, M)$
12. *If* $\text{future\_word}(G_{\text{stable}}, [::\ A \mapsto @(B, C)\&G_{\text{targets}}], D, M)$ *and*
    not exists $N \in \mathbb{A}$ *s.t.* $\Gamma \vdash N : C$ *then*
    $\text{future\_word}(G_{\text{stable}}, \text{dropTargets}(G_{\text{targets}}), D, M)$

13. *If* $\text{FailSound}(G_{\text{stable}})$ *and*
$(\text{computeFailExisting}(G_{\text{stable}}, C)).1 = \textit{true}$ *and*
$\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, [:: A \mapsto @(B,C) \& G_{\text{targets}}])$ *then*
$\text{FCL\_complete}(\text{tgt}, G'_{\text{stable}}, \text{dropTargets}(G_{\text{targets}}))$ *where*

$$G'_{\text{stable}} = \begin{cases} G_{\text{stable}} & \textit{for } (\text{computeFailExisting}(G_{\text{stable}}, C)).2 = \textit{true} \\ [:: C \mapsto \bot \& G_{\text{stable}}] & \textit{otherwise} \end{cases}$$

14. *If* $A \mapsto @(B,C)$ *is in* G_{stable} *and*
$\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, [:: A \mapsto @(B,C) \& G_{\text{targets}}])$ *and*
$\text{future\_word}(G_{\text{stable}}, [:: A \mapsto @(B,C) \& G_{\text{targets}}], D, M)$ *then*
$\text{future\_word}(G_{\text{stable}}, G_{\text{targets}}, D, M)$

15. *If* $\text{computeFailExisting}(G, A) = (\textit{false}, \textit{true})$ *then* A *is in* $\text{parameterTypes}(G)$

16. *If* $\text{isOmega}(C)$ *then* $\text{future\_word}(G_\omega(C) +\!\!+ G_{\text{stable}}, G_{\text{targets}}, C, M)$

17. *If* $\text{future\_word}(G_{\text{stable}}, [:: A \mapsto @(B,C) \& G_{\text{targets}}], D, M)$ *and*
for all $N \in \mathbb{A}$: $\Gamma \vdash N : C$ *implies* $\text{future\_word}([:: A \mapsto @(B,C) \& G_{\text{stable}}], G_{\text{targets}}, C, N)$ *then*
$\text{future\_word}([:: A \mapsto @(B,C) \& G_{\text{stable}}], G_{\text{targets}}, D, M)$

18. *If* $\text{isOmega}(C)$ *and* $\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, [:: A \mapsto @(B,C) \& G_{\text{targets}}])$ *then*
$\text{FCL\_complete}(\text{tgt}, [:: A \mapsto @(B,C) \& G_\omega(C) +\!\!+ G_{\text{stable}}], G_{\text{targets}})$

19. $(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), (G, b), c)).1 =$
$(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), ([::], b), c)).1 +\!\!+ G$

20. $(\text{foldl}(\lambda\ s\ c.\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), s, c), ([::], \textit{true}), \text{enum}(\mathbf{B}))).1 =$
$\text{flatten}(\text{map}($
$\lambda\ c.(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), ([::], \textit{true}), c)).1,$
$\text{rev}(\text{enum}(\mathbf{B}))))$

21. *If*
$(\text{foldl}(\lambda\ s\ c.\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), s, c), ([::], \textit{true}), \text{enum}(\mathbf{B}))).2 = \textit{true}$
then
$(\text{foldl}(\lambda\ s\ c.\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), s, c), ([::], \textit{true}), \text{enum}(\mathbf{B}))).1 = [::]$

22. $\text{commitMultiArrow}(G, c, (\text{srcs}, A)) = \text{commitMultiArrow}([::], c, (\text{srcs}, A)) +\!\!+ G$

23. $\text{commitUpdates}(G, A, c, \text{covers}) =$
$\text{rev}(\text{map}(\lambda m.\text{rev}(\text{commitMultiArrow}([::], c, (m.1, A))), \text{covers})) +\!\!+ G$

24. $\text{lsize}(\text{commitMultiArrow}([::], c, \text{srcs}, \text{tgt})) = 1 + \text{lsize}(\text{srcs})$

25. $\text{nth}(\text{tgt} \mapsto @(\text{mkArrow}(\text{take}(\text{lsize}(\text{srcs}) - (n-1), \text{srcs}), \text{tgt}), \text{nth}(\text{tgt}, \text{srcs}, \text{lsize}(\text{srcs}) - n)),$
$\quad \text{commitMultiArrow}([::], c, (\text{srcs}, \text{tgt})), n) =$
$$\begin{cases} \text{mkArrow}(\text{srcs}, \text{tgt}) \mapsto c & \textit{for } n = 0 \\ \text{mkArrow}(\text{take}(\text{lsize}(\text{srcs}) - n, \text{srcs}), \text{tgt}) \mapsto & \textit{otherwise} \\ \quad @(\text{mkArrow}(\text{take}(\text{lsize}(\text{srcs}) - (n-1), \text{srcs}), \text{tgt}), \text{nth}(\text{tgt}, \text{srcs}, \text{lsize}(\text{srcs}) - n)) & \end{cases}$$

26. *If* $G_{\text{nextTargets}} = \text{rev}(\text{flatten}(\text{map}(\lambda\, m.\text{commitMultiArrow}([::], c, (m.1, C)), ms)))$ *then*
$G_{\text{nextTargets}} = A \mapsto d$ *for some* $A \in \mathbb{T}$ *and* $d \in \mathbf{B}$ *or* $G_{\text{nextTargets}} = [::]$

27. *If* $G_{\text{nextTargets}} =$
$\quad \text{flatten}(\text{map}(\lambda\, c.(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), ([::], \textit{true}), c)).1, cs))$ *then*
$G_{\text{nextTargets}} = A \mapsto d$ *for some* $A \in \mathbb{T}$ *and* $d \in \mathbf{B}$ *or* $G_{\text{nextTargets}} = [::]$

28. $\text{group}(\text{commitMultiArrow}([::], c, \text{srcs}, \text{tgt})) = [::\text{commitMultiArrow}([::], c, \text{srcs}, \text{tgt})]$

29. *If* $\text{future\_word}([::], G_{\text{targets}}, D, M)$ *and exists* $N \in \mathbb{A}$ *s.t.* $\Gamma \vdash N : C$ *then*
$\text{future\_word}([::], G_{\text{targets}} +\!\!+ [::A \mapsto @(B, C)], D, M)$

30. $\text{parameterTypes}(\text{commitMultiArrow}([::], c, \text{srcs}, \text{tgt})) = \text{rev}(\text{srcs})$

31. *If* $\text{future\_word}(G_{\text{stable}}, G_{\text{targets}}, A, M)$ *and*
$G'_{\text{targets}} = [::A \mapsto c\&G]$ *or* $G'_{\text{targets}} = [::]$ *then*
$\text{future\_word}(G_{\text{stable}}, G_{\text{targets}} +\!\!+ G'_{\text{targets}}, A, M)$

32. *If* $\text{future\_word}(G_{\text{stable}}, G'_{\text{targets}}, A, M)$ *and*
$G'_{\text{targets}} = [::A \mapsto c\&G]$ *or* $G'_{\text{targets}} = [::]$ *then*
$\text{future\_word}(G_{\text{stable}}, G_{\text{targets}} +\!\!+ G'_{\text{targets}}, A, M)$

33. *If* $\Gamma \vdash M : C$ *and not* $\text{isOmega}(C)$ *then*
$\text{future\_word}([::],$
$\quad \text{flatten}(\text{map}(\lambda\, c.(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), ([::], \textit{true}), c)).1, \text{rev}(\text{enum}(\mathbf{B}))))$
$\quad C, M)$

34. *If* $\text{future\_word}(G_{\text{stable}}, [::A \mapsto @(B, C)\&G_{\text{targets}}], D, M)$ *and not* $\text{isOmega}(C)$ *then*
$\text{future\_word}([::A \mapsto @(B, C)\&G_{\text{stable}}],$
$\quad G_{\text{targets}} +\!\!+$
$\quad\quad \text{flatten}(\text{map}(\lambda\, c.(\text{accumulateCovers}(\Gamma', C, \text{primeFactors}(C), ([::], \textit{true}), c)).1, \text{rev}(\text{enum}(\mathbf{B}))),$
$\quad D, M)$

35. *If* B *is in* $\text{targetTypes}(\text{pmap}(\lambda\, g.\text{ohead}(\text{rev}(g)), \text{group}(\text{targets})))$ *then*
B *is in* $\text{targetTypes}(\text{pmap}(\lambda\, g.\text{ohead}(\text{rev}(g)), \text{group}([::r\&\text{targets}])))$

36. *If* $\text{reduceMultiArrows}(\text{covers}) = [::]$ *then* $\text{covers} = [::]$

37. *If* (foldl(accumulateCovers(Γ', C, primeFactors(C), ([::], *true*), enum(**B**)))).1 = [::] *then* (foldl(accumulateCovers(Γ', C, primeFactors(C), ([::], *true*), enum(**B**)))).2 = *true*

38. *For all* A *in* targetTypes(pmap(
    $\lambda\, g$.ohead(rev(g)),
    group(flatten(map($\lambda\, c$.(accumulateCovers(Γ', C, primeFactors(C), ([::], *true*), c)).1,
    rev(enum(**B**))))))) :
    $A = C$

39. *If not* isOmega(C) *and* FCL_complete(tgt, G_{stable}, [:: $A \mapsto @(B, C)\&G_{\text{targets}}$]) *and* $(G'_{\text{stable}}, G'_{\text{targets}}) =$
$$\begin{cases} ([::\, A \mapsto @(B,C)\&G_{\text{stable}}], G_{\text{nextTargets}}) \\ \qquad \textit{for}\ \text{inhabit\_cover}(\Gamma', G_{\text{targets}}, C) = (\textit{false}, G_{\text{nextTargets}}) \\ ([::\, C \mapsto \bot\&G_{\text{stable}}], \text{dropTargets}(G_{\text{targets}})) \quad \textit{otherwise} \end{cases}$$
then FCL_complete(tgt, G'_{stable}, G'_{targets})

40. *If* $\text{FailSound}_{\Gamma}(G_{\text{stable}})$ *and* noTargetFailures(G_{targets}) *and* FCL_complete(tgt, G_{stable}, G_{targets}) *then*
    for $(G'_{\text{stable}}, G'_{\text{targets}})$ = inhabitation_step(Γ', G_{stable}, G_{targets})*:*
    FCL_complete(tgt, G'_{stable}, G'_{targets})

41. *If* $\text{FailSound}_{\Gamma}(G_{\text{stable}})$ *and* noTargetFailures(G_{targets}) *and* FCL_complete(tgt, G_{stable}, G_{targets}) *and* $\Gamma' \vdash (G_{\text{stable}}, G_{\text{targets}}) \rightsquigarrow_* (G'_{\text{stable}}, G'_{\text{targets}})$ *then* FCL_complete(tgt, G'_{stable}, G'_{targets})

PROOF The most important trick is to define potential future derivability by future_word. It extends the language definition of Tree Grammars (Definition 29) to consider rules scheduled in G_{targets}. If at some point of the derivation of an applicative term list G_{stable} does not yet include enough rules, the derivation can be suspended. This suspension is only possible if G_{targets} includes a rule that will – in later machine steps – ensure G_{stable} is sufficient to continue. The rule is identified by regrouping subsequent G_{targets} entries according to the multi arrow they originated from. Remember that the cover machine uses [:: $A_1 \to A_2 \to B \mapsto c\&[::\, A_2 \to B \mapsto @(A_1 \to A_2 \to B, A_1)\&[::\, B \mapsto @(A_2 \to B, A_2)\&G'_{\text{targets}}]]]$ for multi arrow $m = ([::\, A_2\&[::A_1]], B)$, target B, and combinator c with a split type $\Gamma'(c)$ including m. Function group isolates multi arrow entries into separate lists, resulting in $[::\, [::\, A_1 \to A_2 \to B \mapsto c\&[::\, A_2 \to B \mapsto @(A_1 \to A_2 \to B, A_1)\&[::B \mapsto @(A_2 \to B, A_2)]]]\&\text{groups}']$. If all the types in parameter positions, i.e. A_1 and A_2, are inhabited, the group will not be discarded and its rules can be used to resume suspended derivations of $A_1 \to A_2 \to B$, $A_2 \to B$, and B. Property FCL_complete(tgt, G_{stable}, G_{targets}) characterizes machine states G_{stable} and G_{targets} which in future derive all terms M s.t. $\Gamma \vdash M$: tgt. For this additionally all terms with types in parameter positions of G_{stable} have to be derivable in future, guaranteeing that all choices for N will be considered when $\Gamma \vdash @(c, N)$: tgt for any c. Moreover,

terms for some of the target types in G_{targets} need to be future derivable. Suppose G_{stable} includes $A \mapsto @(B, C)$ and rules with target type C are not yet included in G_{stable}, then rules for all multi arrows with type C are required to be scheduled for future consideration in G_{targets}. However, simply requiring that all terms with target types of rules in G_{targets} are future derivable is too restrictive. Suppose type A_1 in the previous example is uninhabited. Then group $[:: A_1 \to A_2 \to B \mapsto c\&[:: A_2 \to B \mapsto @(A_1 \to A_2 \to B, A_1)\&[::B \mapsto @(A_2 \to B, A_2)]]]$ does not contribute to future derivability and it will be discarded after encountering rule $A_2 \to B \mapsto @(A_1 \to A_2 \to B, A_1)$. Yet, terms N with $\Gamma \vdash N : A_2 \to B$ might exist (e.g. using a combinator $d : A_2' \to B$ with $A_2 \leq A_2'$) but not be (future) derivable. For them, rules $[:: A_2' \to B \mapsto d\&[::B \mapsto @(A_2' \to B, A_2')]]$ would derive N from non-terminal $A_2' \to B$ instead. In fact, only terms of the last target type of a group have to be derivable. By the definition of future_word these terms only exist if all of the parameter types in the group are inhabited and therefore the group will not be discarded. In the definition of FCL_complete, the part $\text{pmap}(\lambda g.\,\text{ohead}(\text{rev}(g)), \text{group}(G_{\text{targets}}))$ projects all non-empty groups exactly to their last target types.

Keeping the illustration above in mind, statements of the lemma break down its proof into manageable components, individually proven by case analysis, induction and use of the previously established propositions. Statements 1–7 make it possible to reason about function group used in the definitions of future_word and FCL_complete. Statement 9 is evident from the definitions of word and future_word. It shows that future derivability is a sufficient condition for derivability in the final machine state where G_{targets} is empty. The direct consequence, proposition 8, justifies to prove $\text{FCL\_complete}(\text{tgt}, G_{\text{stable}}, [::])$ instead of $\Gamma \vdash M : \text{tgt}$ implies $\text{word}(G_{\text{stable}}, M, \text{tgt})$ to obtain completeness. Proposition 10 allows weakening of future derivability from G_{stable}, showing that future_word is closed under subset membership of stable rule lists. Statement 11 is the first of multiple absorption properties (c.f. 14, 17, 34) incorporating rules form G_{targets} into G_{stable}. Specifically, with statement 11 combinator rules can be removed from G_{targets} if they are included in G_{stable}, possibly because of application of the prior weakening property. This is in correspondence to case ① of inhabitation_step in Definition 29. Statements 12 and 13 justify to discard target rule groups because of uninhabited parameters identified by computeFailExisting. The previously established invariant FailSound (Lemma 32) is required to ensure completeness is not lost due to conflicting rules in G_{stable}. Absorption rule 14 can remove applicative rules from the targets. It has FCL_complete as precondition to ensure derivations of the removed rule $A \mapsto @(B, C)$, which previously succeeded in suspended state from G_{targets} without further derivation for C, do not get stuck when replaced by derivations from G_{stable} that do require C. Rule 14 is used for the last two situations of case ③, Definition 29. In the first situation the rule to move already existed in G_{stable} and this fact is checked by computeFailExisting. Statement 15 proves soundness of the decision made by computeFailExisting. The other situation is grouped together under ④ and again subdivided into two cases. Statements 16, 17, and their consequence statement 18 dispatch the first case of ④ in which ω is a subtype of the parameter type of the next applicative rule. For the other situation, ⑤, function inhabit_cover has to be studied in more depth. The

analysis proceeds with the auxiliary functions. Statements 19 and 20 allow separate reasoning about the new entries in G_{targets} for individual combinators. By proposition 21 the Boolean that detects if there are no updates to G_{targets} is computed soundly. Statements 22–30 help to reason about adding multi arrows to the target queue. Analogous to statement 10, the target queue can grow whilst weakening future_word to include at least the same derivable terms, which is shown in 31 and 32. This is then used to prove the absorption rule 34 in combination with future derivability of the terms for its parameter type C in statement 33. Propositions 35–38 provide more properties of grouping collected targets. They relate the target type of freshly collected grouped rules to the requested parameter type (statement 38) and establish (statement 37) the other part of correctness for the emptyness detection Boolean previously addressed in statement 21. Results for case ⑤ in Definition 29 are combined in proposition 39. The final statements 40, and 41 are the main properties to show. They establish FCL_complete as an invariant across machine steps. The poof of 40 is by case analysis on all of the branches of inhabitation_step, dispatching each branch with the facts proven above. The proof of 41 is by induction on the machine steps using 40, Lemma 32.7, and Lemma 32.10 for progress. ■

The proof of Lemma 33 is, together with that of Lemma 24, the most intricate of this text and the interested reader is encouraged to interactively step through the Coq formalization for all of its details (possibly starting in reverse from Lemma 33.40 to see where which part becomes necessary). Both proofs have revealed subtle mistakes in initial attempts to define the machines characterized by their lemmas. Spotting these mistakes would have been impossible or at least improbable without the aid of Coq as a formal verification system. Also tuning the delicate definitions of future_word and FCL_complete required multiple attempts, and being able to automatically re-check which parts were affected by the changes was of great value during development.

Finally the initial call to the inhabitation machine can be constructed and correctness of the type inhabitation algorithm can be proven.

Theorem 6 (Type Inhabitation Correctness)
For applicative terms $\mathbb{A}$ formed over any finite enumerable combinator base $\mathbf{B}$, and intersection types $\mathbb{T}$ formed over any $\leq_{\mathbf{C}}$-preordered countable set $\mathbf{C}$ of type constructor symbols define:

$$\text{inhabit} : (\mathbf{B} \to \mathbb{T}) \times \mathbb{T} \to \mathbb{R}^*$$

$$\text{inhabit}(\Gamma, A) =$$

$$\begin{cases} G_\omega(A) \quad \textit{for } \text{isOmega}(A) = \textit{true} \\ \text{inhabitationMachine}(\lambda c.\,\text{splitTy}(\Gamma(c)), [::], G_{\text{targets}})(d) \\ \qquad \textit{for } \text{isOmega}(A) = \textit{false and} \\ \qquad\quad \text{inhabit\_cover}(\lambda c.\,\text{splitTy}(\Gamma(c)), [::], A) = (\textit{false}, G_{\text{targets}}) \textit{ and} \\ \qquad\quad d \textit{ constructed by Lemma 31.24} \\ A \mapsto \bot \quad \textit{otherwise} \end{cases}$$

with the following properties for all $A \in \mathbb{T}$ *and* $M \in \mathbb{A}$*:*

1. *If* $\mathrm{word}(\mathrm{inhabit}(\Gamma, A), A, M)$ *then* $\Gamma \vdash M : A$
2. *If* $\Gamma \vdash M : A$ *then* $\mathrm{word}(\mathrm{inhabit}(\Gamma, A), A, M)$

PROOF By case distinction following the branches of inhabit and use of the invariants established in Lemma 29, Lemma 32, and Lemma 33. ■

2.6 Algebraic Interpretation of Results

Terms of the tree grammar computed in Theorem 6 are of little use by themselves. This section discusses a general algebraic framework to interpret them in different target languages.

2.6.1 Subsorted Σ-Algebra Families

Signatures are a common language for interfaces and connect combinatory logic with synthesis target languages. Families of subsorted signatures allow to model interfaces parametric and subtype polymorphism.

Definition 33 (Subsorted Signature Families) I-indexed $\leq_{\mathbb{S}}$-subsorted signature families Σ are defined by $\Sigma_{i\in I} = (\mathbb{S}, \mathbb{O}, \text{arity}_i, \text{dom}_i, \text{range}_i)$ where

- I is a set of indexes
- $\mathbb{S}$ is a set of sorts ordered by a preorder (reflexive, transitive) relation $\leq_{\mathbb{S}} \subseteq \mathbb{S} \times \mathbb{S}$
- $\mathbb{O}$ is a finite set of operations
- $\text{arity} : (\mathbb{O} \to \mathbb{N})_{i\in I}$ is a family of functions, assigning an arity to each operation
- $\text{dom} : \left(\left(\prod_{n=1}^{\text{arity}_i(o)} \mathbb{S}\right)_{o\in\mathbb{O}}\right)_{i\in I}$ is a family of families of vectors, assigning a domain, which is a vector of $\text{arity}_i(o)$ sorts, to each operation
- $\text{range} : ((\mathbb{S})_{o\in\mathbb{O}})_{i\in I}$ is a family of families of sorts, assigning a range to each operation. □

Restricting I to a singleton set yields traditional subsorted signatures described in [83]. Further, the restriction $\leq_{\mathbb{S}} = \{(s,s) \mid s \in \mathbb{S}\}$ results in many-sorted signatures [70; 71; 150]. Less trivial restrictions on the parameter set I and their implications on the equations that can be imposed on interpretations have been investigated in [148]. Signatures where $\mathbb{S}$ is a singleton have also been studied [109]. Avoiding any meta-theoretic issues and simplifying the Coq formalization, from now on all sort sets $\mathbb{S}$ are additionally assumed to be countable and $\leq_{\mathbb{S}}$ is assumed to be decidable. The following example illustrates the roles of the different components of a signature.

Example 2 (Signature family for even and odd natural numbers) Even and odd natural numbers can be constructed using

$$\Sigma^{\mathbb{N}} = (\mathbb{S}_{\mathbb{N}}, \mathbb{O}_{\mathbb{N}}, \text{arity}_i^{\mathbb{N}}, \text{dom}_i^{\mathbb{N}}, \text{range}_i^{\mathbb{N}})_{i \in \{0,1,2\}}$$

where:

- The sorts $\mathbb{S}_{\mathbb{N}} = \{\text{Nat}, \text{Even}, \text{Odd}\}$ are ordered by:

  $$\leq_{\mathbb{S}_{\mathbb{N}}} = \{(\text{Nat}, \text{Nat}), (\text{Even}, \text{Nat}), (\text{Odd}, \text{Nat}), (\text{Even}, \text{Even}), (\text{Odd}, \text{Odd})\}$$

  This ordering describes compatibility, indicating that every $\mathbb{N}$ sorted interpretation of the signature is compatible with using an Even or Odd sorted interpretation instead.

- The operations are $\mathbb{O}_{\mathbb{N}} = \{\text{zero}, \text{succ}\}$ for computing zero and the successor of a number.

- The arities are given by $\text{arity}_i^{\mathbb{N}} = \{\text{zero} \mapsto 0, \text{succ} \mapsto 1\}$ for every i, i.e. zero needs no arguments and succ needs one argument, both independent of the signature index.

- The domains are given by

  $$\begin{aligned}
  \text{dom}_0^{\mathbb{N}} &= \{\text{zero} \mapsto (), \text{succ} \mapsto \mathbb{N}\} \\
  \text{dom}_1^{\mathbb{N}} &= \{\text{zero} \mapsto (), \text{succ} \mapsto \text{Even}\} \\
  \text{dom}_2^{\mathbb{N}} &= \{\text{zero} \mapsto (), \text{succ} \mapsto \text{Odd}\}.
  \end{aligned}$$

  This means in $\Sigma_0^{\mathbb{N}}$ operation succ takes any natural number, in $\Sigma_1^{\mathbb{N}}$ operation succ takes even numbers, and in $\Sigma_2^{\mathbb{N}}$ it takes odd numbers.

- The ranges are given by

  $$\begin{aligned}
  \text{range}_0^{\mathbb{N}} &= \{\text{zero} \mapsto \mathbb{N}, \text{succ} \mapsto \mathbb{N}\} \\
  \text{range}_1^{\mathbb{N}} &= \{\text{zero} \mapsto \text{Even}, \text{succ} \mapsto \text{Odd}\} \\
  \text{range}_2^{\mathbb{N}} &= \{\text{zero} \mapsto \text{Even}, \text{succ} \mapsto \text{Even}\}.
  \end{aligned}$$

  So, with the prior definition of domains, operation succ takes natural numbers to natural numbers, even numbers to odd numbers, and odd numbers to odd numbers, while zero produces a natural number in $\Sigma_0^{\mathbb{N}}$ and an even number for the signature indexes 1 and 2. Note that $\Sigma_0^{\mathbb{N}}$ is just the standard single sorted signature for natural numbers [70]. Wittgenstein used an earlier notation for the single sorted signature in the Tractatus, sentence 6.02 [202]. □

Every signature family gives rise to a functor $\mathbb{F}$, which describes the inputs required for interpreting operations.

Definition 34 (Subsorted Signature Family Functor) Given an I-indexed $\leq_{\mathbb{S}}$-subsorted signature family $\Sigma_{i\in I} = (\mathbb{S}, \mathbb{O}, \text{arity}_i, \text{dom}_i, \text{range}_i)$, its functor $\mathbb{F}^{\Sigma}$ is defined by:

$$\begin{aligned}
&\mathbb{F}^{\Sigma} : \mathbf{Set}^{\mathbb{S}} \to \mathbf{Set}^{\mathbb{S}} \\
&\mathbb{F}^{\Sigma}(\mathbb{C}) = \left(\bigoplus_{i\in I} \bigoplus_{o\in \text{ranged}(i,s)} \prod_{n=1}^{\text{arity}_i(o)} \mathbb{C}_{\pi_n(\text{dom}_i(o))} \right)_{s\in\mathbb{S}} \\
&\mathbb{F}^{\Sigma}(f)_s(i, o, (x_1, x_2, \ldots, x_{\text{arity}_i(o)})) = \\
&\quad (i, o, (f_{\pi_1(\text{dom}_i(o))}(x_1), f_{\pi_2(\text{dom}_i(o))}(x_2), \ldots, f_{\pi_{\text{arity}_i(o)}(\text{dom}_i(o))}(x_{\text{arity}_i(o)})))
\end{aligned}$$

where

- $\mathbb{C}$ is a $\mathbb{S}$-indexed family of sets
- $\text{ranged}(i, s) = \{o \in \mathbb{O} \mid \text{range}_i(o) \leq_{\mathbb{S}} s\}$ collects all operations that have ranges compatible with s
- $f : (\mathbb{C}_s \to \mathbb{D}_s)_{s\in\mathbb{S}}$ is a morphism in $\mathbf{Set}^{\mathbb{S}}$

The functorial equations $\mathbb{F}^{\Sigma}(\text{id}) = \text{id}$ and $\mathbb{F}^{\Sigma}(f \circ g) = \mathbb{F}^{\Sigma}(f) \circ \mathbb{F}^{\Sigma}(g)$ follow immediately from the categorical equations of **Set**. □

Example 3 (Functor for the signature family of even and odd natural numbers) Let the $\mathbb{S}_{\mathbb{N}}$-indexed family of sets $\mathbb{C}^{\mathbb{S}_{\mathbb{N}}}$ be defined by:

$$\begin{aligned}
&\mathbb{C}^{\mathbb{S}_{\mathbb{N}}}_{\text{Nat}} = \mathbb{N} \\
&\mathbb{C}^{\mathbb{S}_{\mathbb{N}}}_{\text{Even}} = \{0, 2, 4, 6, \ldots\} \\
&\mathbb{C}^{\mathbb{S}_{\mathbb{N}}}_{\text{Odd}} = \{1, 3, 5, 7, \ldots\}
\end{aligned}$$

For Example 2 and sort Odd, signature $\Sigma^{\mathbb{N}}$ has functor $\mathbb{F}^{\Sigma^{\mathbb{N}}}(\mathbb{C}^{\mathbb{S}_{\mathbb{N}}})_{\text{Odd}}$. It is the set of all triples of indexes, operations and arguments usable to obtain odd numbers. Only operation succ from $\Sigma^{\mathbb{N}}_1$ with an even numbered input can produce odd numbers, and so it follows that

$$\mathbb{F}^{\Sigma^{\mathbb{N}}}(\mathbb{C}^{\mathbb{S}_{\mathbb{N}}})_{\text{Odd}} = \{(1, \text{succ}, x) \mid x \in \Sigma^{\mathbb{N}}_{\text{Even}}\}.$$

What does it mean to interpret, or in other words execute, an operation? Mathematically, F^{Σ}-Algebras answer this question.

Definition 35 ($\mathbb{F}^{\Sigma}$-Algebra) Given an indexed $\leq_{\mathbb{S}}$-subsorted signature family Σ, an $\mathbb{F}^{\Sigma}$-Algebra $\mathscr{A}^{\Sigma} = (\mathbb{C}, h)$ is defined by

- A carrier $\mathbb{C}$, which is a $\mathbb{S}$-indexed family of sets, and
- An action $h : \left(\mathbb{F}^{\Sigma}(\mathbb{C})_s \to \mathbb{C}_s\right)_{s\in\mathbb{S}}$, which is a $\mathbb{S}$-indexed family of functions □

Example 4 (An algebra for the signature family of even and odd natural numbers) Continuing Example 3, $\mathscr{A}^{\Sigma^{\mathbb{N}}}_{\mathbb{C}^{\Sigma^{\mathbb{N}}}} = (\mathbb{C}^{\Sigma^{\mathbb{N}}}, h^{\Sigma^{\mathbb{N}}})$ is an algebra defined by $\mathbb{C}^{\Sigma^{\mathbb{N}}}$ and its action

$$h^{\Sigma^{\mathbb{N}}} : \left(\mathbb{F}^{\Sigma^{\mathbb{N}}}(\mathbb{C}^{\Sigma^{\mathbb{N}}})_s \to \mathbb{C}^{\Sigma^{\mathbb{N}}}_s\right)_{s\in\mathbb{S}}$$
$$h^{\Sigma^{\mathbb{N}}}_s(i, \text{zero}, ()) = 0$$
$$h^{\Sigma^{\mathbb{N}}}_s(i, \text{succ}, x) = x + 1$$

For each index, operation and argument-vector, the action computes the result of interpreting that operation. Signatures are interfaces, which can have different implementations. This is expressed by different algebras for the same signature. An alternative algebra $\mathscr{A}^{\Sigma^{\mathbb{N}}}_{\mathbf{Mod2}} = (\mathbf{Mod2}, g^{\Sigma^{\mathbb{N}}})$ can be defined by the carrier

$$\mathbf{Mod2}_{\text{Nat}} = \{0, 1\} \qquad \mathbf{Mod2}_{\text{Even}} = \{0\} \qquad \mathbf{Mod2}_{\text{Odd}} = \{1\}$$

and action $g^{\Sigma^{\mathbb{N}}} : \left(F^{\Sigma^{\mathbb{N}}}(\mathbf{Mod2})_s \to \mathbf{Mod2}_s\right)_{s\in\mathbb{S}}$

$$g^{\Sigma^{\mathbb{N}}}_s(i, \text{zero}, ()) = 0 \qquad g^{\Sigma^{\mathbb{N}}}_{\text{Nat}}(0, \text{succ}, 0) = 1 \qquad g^{\Sigma^{\mathbb{N}}}_{\text{Odd}}(1, \text{succ}, 0) = 1$$
$$g^{\Sigma^{\mathbb{N}}}_{\text{Nat}}(0, \text{succ}, 1) = 0 \qquad g^{\Sigma^{\mathbb{N}}}_{\text{Even}}(2, \text{succ}, 1) = 0$$

In contrast to $\mathscr{A}^{\Sigma^{\mathbb{N}}}_{\mathbb{C}^{\Sigma^{\mathbb{N}}}}$, algebra $\mathscr{A}^{\Sigma^{\mathbb{N}}}_{\mathbf{Mod2}}$ forgets all information about the constructed numbers, except for their parity. □

The problem of automatic program construction can be reformulated to automatic enumeration of the image of the action of an algebra. Suppose each sort describes a desired program property, and the carrier set $\mathbb{C}_s$ of some $\mathbb{F}^{\Sigma}$-algebra $(\mathbb{C}, h)$ is the set of programs with that property. Now each operation in Σ is necessarily interpreted by h as a program transformation, mapping source programs with properties specified by its domain to target programs with the properties specified by its range. Enumerating the range of h_s for a given sort s then produces all programs with property s that are constructed using the transformations specified in Σ. The notion of being constructed by operations in Σ is made precise using the following definition.

Definition 36 (Algebraically Generated Objects) Let Σ be an indexed $\leq_{\mathbb{S}}$-subsorted signature family, and $\mathscr{A}^{\Sigma} = (\mathbb{C}, h)$ be an $\mathbb{F}^{\Sigma}$-algebra. An object is algebraically generated iff it is contained in any set in the family $\mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}$. For any sort s, $\mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}_s \subseteq \mathbb{C}_s$ is the least set closed under the rule:

If $(i, o, (x_1, x_2, \ldots, x_{\text{arity}_i(o)})) \in \mathbb{F}^{\Sigma}(\mathbb{C})_s$
and for all $i \in \{1, 2, \ldots, \text{arity}_i(o)\} : x_i \in \mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}_{\pi_i(\text{dom}_i(o))}$
then $h_s(i, o, (x_1, x_2, \ldots, x_{\text{arity}_i(o)})) \in \mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}_s$

2.6.2 Undecidability of Sort Emptiness with infinite Index Sets

The sort emptiness problem of algebraically generated sets is the following: Given an index set I, an I-indexed $\leq_{\mathbb{S}}$-subsorted signature family Σ, an $\mathbb{F}^{\Sigma}$-algebra $\mathscr{A}^{\Sigma} = (\mathbb{C}, h)$ and a sort s, is $\mathbf{AlgGen}_s^{\mathscr{A}^{\Sigma}} = \emptyset$? This problem is interesting in two ways: it turns out to be undecidable if I is an arbitrary set and the undecidability proof is inspired by the undecidability proof for Combinatory Logic with Hilbert Schematism presented in [163]. The proof idea is to reduce the halting problem of 2-Counter Automata, which are a well-known Turing complete machine model [140; 77; 117], to the emptiness problem by equating $\mathbf{AlgGen}_s^{\mathscr{A}^{\Sigma}}$ with counter values for which the configuration represented by s halts. It immediately follows, that enumerating algebraically generated sets for program construction implicitly includes a programming language to control the enumeration process – a point which has been made before for Combinatory Logic [163].

Definition 37 (2-Counter Automata)
Following the definition in [163], a 2-Counter Automaton is defined by $(S, s_0, s_f, c_1^0, c_2^0, R)$ where

- S is a finite set of states
- $s_0 \in S$ is the initial state
- $s_f \in S$ is the final state
- $c_1^0, c_2^0 \in \mathbb{N}$ are the initial counter values
- $R \subset \mathscr{R}$ is a finite set of commands for the transition relation drawn from

$$\mathscr{R} \ni t ::= \mathtt{Add}_i(s_1, s_2) \mid \mathtt{Sub}_i(s_1, s_2) \mid \mathtt{Tst}_i(s_1, s_2, s_3)$$

  with $i \in \{1, 2\}$ and $s_1, s_2, s_3 \in S$

Automaton configurations are triples in $C = S \times \mathbb{N} \times \mathbb{N}$ containing a state and two counters. The transition relation $\rhd \subset C \times C$ is the least relation closed under the following rules:

$$\frac{\mathtt{Add}_1(s_1, s_2) \in R}{(s_1, c_1, c_2) \rhd (s_2, c_1 + 1, c_2)} \qquad \frac{\mathtt{Add}_2(s_1, s_2) \in R}{(s_1, c_1, c_2) \rhd (s_2, c_1, c_2 + 1)}$$

$$\frac{\mathtt{Sub}_1(s_1, s_2) \in R}{(s_1, c_1 + 1, c_2) \rhd (s_2, c_1, c_2)} \qquad \frac{\mathtt{Sub}_2(s_1, s_2) \in R}{(s_1, c_1, c_2 + 1) \rhd (s_2, c_1, c_2)}$$

$$\frac{\mathtt{Tst}_1(s_1, s_2, s_3) \in R}{(s_1, 0, c_2) \rhd (s_2, 0, c_2)} \qquad \frac{\mathtt{Tst}_2(s_1, s_2, s_3) \in R}{(s_1, c_1, 0) \rhd (s_2, c_1, 0)}$$

$$\frac{\mathtt{Tst}_1(s_1, s_2, s_3) \in R}{(s_1, c_1 + 1, c_2) \rhd (s_3, c_1 + 1, c_2)} \qquad \frac{\mathtt{Tst}_2(s_1, s_2, s_3) \in R}{(s_1, c_1, c_2 + 1) \rhd (s_3, c_1, c_2 + 1)}$$

The reflexive transitive closure of $\rhd$ is denoted by $\blacktriangleright$ and for $k \in C$, notation $k \blacktriangleright$ is short for there exists $c_1, c_2 \in \mathbb{N}$ s.t. $k \blacktriangleright (s_f, c_1, c_2)$. □

Definition 38 (Signatures for 2-Counter Automata) For the 2-Counter Automaton $A = (S, s_0, s_f, c_1^0, c_2^0, R)$ signature family $\Sigma^{\mathbf{2Aut}}(A) = (\mathbb{S}_A, \mathbb{O}_A, \text{arity}_i^A, \text{dom}_i^A, \text{range}_i^A)_{i \in \mathbb{N}\times\mathbb{N}}$ is defined by

- The sorts $\mathbb{S}_A = C = S \times \mathbb{N} \times \mathbb{N}$ ordered by $\leq_{\mathbb{S}_A} = \{(k,k) \mid k \in C\}$
- The operations $\mathbb{O}_A = \{s_f\} \uplus \{\texttt{Add}_i(s_1,s_2) \mid \texttt{Add}_i(s_1,s_2) \in R\} \uplus$
$\{\texttt{Sub}_i(s_1,s_2) \mid \texttt{Sub}_i(s_1,s_2) \in R\} \uplus$
$\{\texttt{Tst}_n^0(s_1,s_2,s_3) \mid \texttt{Tst}_i(s_1,s_2,s_3) \in R\} \uplus$
$\{\texttt{Tst}_n^+(s_1,s_2,s_3) \mid \texttt{Tst}_i(s_1,s_2,s_3) \in R\}$
- The arities $\text{arity}_i^A(o) = \begin{cases} 0 & \text{for } o = s_f \\ 1 & \text{otherwise} \end{cases}$
- The domains $\text{dom}_{(c_1,c_2)}^A(o) = \begin{cases} () & \text{for } o = s_f \\ (s_2, 0, c_2) & \text{for } o = \texttt{Tst}_1^Z(s_1,s_2,s_3) \\ (s_2, c_1, 0) & \text{for } o = \texttt{Tst}_2^Z(s_1,s_2,s_3) \\ (s_3, c_1+1, c_2) & \text{for } o = \texttt{Tst}_1^+(s_1,s_2,s_3) \\ (s_3, c_1, c_2+1) & \text{for } o = \texttt{Tst}_2^+(s_1,s_2,s_3) \\ (s_2, c_1+1, c_2) & \text{for } o = \texttt{Add}_1(s_1,s_2) \\ (s_2, c_1, c_2+1) & \text{for } o = \texttt{Add}_2(s_1,s_2) \\ (s_2, c_1, c_2) & \text{for } o = \texttt{Sub}_n(s_1,s_2) \end{cases}$
- The ranges $\text{range}_{(c_1,c_2)}^A(o) = \begin{cases} (s_f, c_1, c_2) & \text{for } o = s_f \\ (s_1, 0, c_2) & \text{for } o = \texttt{Tst}_1^Z(s_1,s_2,s_3) \\ (s_1, c_1, 0) & \text{for } o = \texttt{Tst}_2^Z(s_1,s_2,s_3) \\ (s_1, c_1+1, c_2) & \text{for } o = \texttt{Tst}_1^+(s_1,s_2,s_3) \\ (s_1, c_1, c_2+1) & \text{for } o = \texttt{Tst}_2^+(s_1,s_2,s_3) \\ (s_1, c_1, c_2) & \text{for } o = \texttt{Add}_n(s_1,s_2) \\ (s_1, c_1+1, c_2) & \text{for } o = \texttt{Sub}_1(s_1,s_2) \\ (s_1, c_1, c_2+1) & \text{for } o = \texttt{Sub}_2(s_1,s_2) \end{cases}$ □

Similar to the type signatures in [163], the idea in the definition of $\Sigma^{\mathbf{2Aut}}$ is to traverse automaton states backwards from the accepting state s_f to the initial state s_0. Each possible automaton transition is implemented by an operation and there exists an additional initial operation s_f without parameters. The signature index controls counter states. Note, that the sorts $\mathbb{S}_A$ directly represent automaton configurations without any need for additional encodings and that no theory of substitutions into sorts is required. This illustrates expressibility of native implementation-specific concerns without the requirement of mappings into approach specific type-structures necessary in [163]. Now an algebra for machine traces can be constructed.

Lemma 34 (Machine Trace Algebra) *For the 2-Counter Automaton* $A = (S, s_0, s_f, c_1^0, c_2^0, R)$, *signature family* $\Sigma^{\mathbf{2Aut}}(A) = (\mathbb{S}_A, \mathbb{O}_A, \text{arity}_i^A, \text{dom}_i^A, \text{range}_i^A)_{i \in \mathbb{N}\times\mathbb{N}}$, *and* $o \in \mathbb{O}_A$, $i \in \mathbb{N}\times\mathbb{N}$, $s \in \mathbb{S}_A$ *the following statements are true*

1. *If* $o \neq s_f$ *then* $\text{range}_i^A(o) \rhd \text{dom}_i^A(o)$

Define algebra $\mathscr{A}^{\Sigma^{\mathbf{2Aut}}(A)}_{\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}} = (\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}, h^{\Sigma^{\mathbf{2Aut}}(A)})$ *with*

$$\mathbb{C}_s^{\Sigma^{\mathbf{2Aut}}(A)} = \{(c_1, c_2) \in \mathbb{N}\times\mathbb{N} \mid s \blacktriangleright (s_f, c_1, c_2)\}$$

$$h_s^{\Sigma^{\mathbf{2Aut}}(A)}((c_1, c_2), o, x) = \begin{cases} (c_1, c_2) & \textit{for } o = s_f \\ x & \textit{otherwise} \end{cases}$$

then

2. *If exists* $(c_1, c_2) \in \mathbb{C}_{s_0}^{\Sigma^{\mathbf{2Aut}}(A)}$ *then* $(s_0, c_1^0, c_2^0) \blacktriangleright (s_f, c_1, c_2)$

3. *If exists* $(c_1, c_2) \in \mathbb{N}\times\mathbb{N}$ *s.t.* $(s_0, c_1^0, c_2^0) \blacktriangleright (s_f, c_1, c_2)$ *then* $(c_1, c_2) \in \mathbf{AlgGen}_{s_0}^{\mathscr{A}^{\Sigma^{\mathbf{2Aut}}(A)}_{\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}}}$

PROOF Proposition 1 is proven by case analysis on o matching range and domain with the rules of $\rhd$. It is used to ensure that $h^{\Sigma^{\mathbf{2Aut}}(A)}$ is well-defined for $o \neq s_f$. Soundness of the carrier $\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}$ and thereby the algebraically generated set is guaranteed by statement 2, which immediately follows from the definition of $\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}$. Finally, proposition 3 states completeness and follows by induction on the steps in $(s_0, c_1^0, c_2^0) \blacktriangleright (s_f, c_1, c_2)$.

The existence of $(c_1, c_2) \in \mathbb{N}\times\mathbb{N}$ s.t. $(s_0, c_1^0, c_2^0) \blacktriangleright (s_f, c_1, c_2)$ is undecidable and so is the by statements 2 and 3 equivalent question for emptiness of $\mathbf{AlgGen}_{s_0}^{\mathscr{A}^{\Sigma^{\mathbf{2Aut}}(A)}_{\mathbb{C}^{\Sigma^{\mathbf{2Aut}}(A)}}}$. ■

2.6.3 Finite Index Sets and Combinatory Logic

Undecidability for infinite index sets does not render the algebraic approach useless. In fact, whenever the index set is finite, type inhabitation in FCL can be used to enumerate $\mathbf{AlgGen}^{\mathscr{A}^\Sigma}$ of any algebra $\mathscr{A}^\Sigma$ for any signature Σ. The result follows from some abstract algebraic properties, which require further definitions and proofs. The first two definitions are extensions of the standard notions for F-CoAlgebras and algebra morphisms [52].

Definition 39 ($\mathbb{F}^\Sigma$-CoAlgebras) Given an indexed $\leq_{\mathbb{S}}$-subsorted signature family Σ, an $\mathbb{F}^\Sigma$-CoAlgebra $\underline{\mathscr{A}}^\Sigma = (\mathbb{C}, h)$ is defined by

- A carrier $\mathbb{C}$, which is a $\mathbb{S}$-indexed family of sets, and
- A co-action $h : \left(\mathbb{C}_s \to \mathbb{F}^\Sigma(\mathbb{C})_s\right)_{s\in\mathbb{S}}$, which is a $\mathbb{S}$-indexed family of functions □

Definition 40 (Algebra Morphism Families) Given an indexed $\leq_{\mathbb{S}}$-subsorted signature family Σ, and F^{Σ}-Algebras $\mathscr{A}^{\Sigma}_{\mathbb{C}} = (\mathbb{C}, h)$, $\mathscr{A}^{\Sigma}_{\mathbb{D}} = (\mathbb{D}, g)$, family $f : (\mathbb{C}_s \to \mathbb{D}_s)_{s \in I}$ is an algebra morphism family, if it makes the following diagram commute

$$\begin{array}{ccc} \mathbb{F}^{\Sigma}(\mathbb{C})_s & \xrightarrow{\mathbb{F}^{\Sigma}(f)_s} & \mathbb{F}^{\Sigma}(\mathbb{D})_s \\ \downarrow h_s & & \downarrow g_s \\ \mathbb{C}_s & \xrightarrow{f_s} & \mathbb{D}_s \end{array}$$

i.e. for all $s \in \mathbb{S}$: $f_s \circ h_s = g \circ \mathbb{F}^{\Sigma}(f)_s$ □

The next lemma extends the work of Paulson [154] to structures with polymorphic indexes.

Lemma 35 (Well-Founded Fixpoint Families) *Let M be a set and $R \subseteq M \times M$ be a well-founded relation (c.f. Lemma 31). Remember that the accessibility predicate of R is the least predicate closed under rule:*

$$\frac{\textit{for all } y\textit{: if } \mathrm{R}(y,x) \textit{ then } \mathrm{Acc}(y)}{\mathrm{Acc}(x)}$$

For any y s.t. $R(y,x)$, let $\mathrm{Acc}^{-1}_x(a_x, y)$ be the proof tree of $\mathrm{Acc}(y)$ in the premise of any given proof tree a_x of $\mathrm{Acc}(x)$. Observe that the proof tree $\mathrm{Acc}^{-1}_x(a_x, y)$ is always strictly smaller than the proof tree a_x.

Given an index set $\mathbb{S}$, an $\mathbb{S}$-indexed family of sets $\mathbb{C}$, a family of sets P indexed over members of $\mathbb{C}$, and a measuring map $m : \prod_{s \in \mathbb{S}} C_s \to M$, define for all $s_1, s_2 \in \mathbb{S}$ and $x \in \mathbb{C}_{s_1}$:

$$\downarrow (x, s_2) = \{y \in \mathbb{C}_{s_2} \mid R(m(s_2, y), m(s_1, x))\}$$

Then given a morphism $F : \prod_{s_1 \in \mathbb{S}} \prod_{x \in \mathbb{C}_{s_1}} \left(\prod_{s_2 \in \mathbb{S}} \prod_{y \in \downarrow(x,s_2)} P_y\right) \to P_x$ the fixpoint morphism

$$\mathrm{FixAcc}_F : \prod_{s \in \mathbb{S}} \prod_{x \in \mathbb{C}} \mathrm{Acc}(m(s,x)) \to P_x$$
$$\mathrm{FixAcc}_F(s_1, x, a_{m(s_1,x)}) = F(s_1, x, \lambda(s_2, y).\, \mathrm{Fix}_F(s_2, y, \mathrm{Acc}^{-1}_{m(s_1,x)}(a_{m(s_1,x)}, m(s_2, y))))$$

can be constructed because of the prior observation on proof tree sizes.

F is extensional in its third argument if for all $s_1 \in \mathbb{S}$, $x \in \mathbb{C}_{s_1}$, and $g, g' : \prod_{s_2 \in \mathbb{S}} \prod_{y \in \downarrow(x,s_2)} P_y$, s.t. for all $s_2 \in \mathbb{S}$, and $y \in \mathbb{C}_{s_2}$: $g(s_2, y) = g'(s_2, y)$, the statement $F(s_1, x, g) = F(s_2, x, g')$ is true. This always holds in extensional set-theory, but has to be shown on a per-case basis in intensional type theory.

For any F that is extensional in its third argument, $s_1 \in \mathbb{S}$, $x \in \mathbb{C}_{s_1}$, proof trees $a_{m(s_1,x)}, a'_{m(s_1,x)}$ of $\mathrm{Acc}(m(s_1,x))$*, and morphisms $G : \prod_{s\in\mathbb{S}} \prod_{y\in\mathbb{C}_s} P_y$ morphism* FixAcc_F *is:*

1. *Invariant in its proof tree argument:* $\mathrm{FixAcc}_F(s_1, x, a_{m(s_1,x)}) = \mathrm{FixAcc}_F(s_1, x, a'_{m(s_1,x)})$
2. *A unique solution to the fixpoint equation: if for all $s_2 \in \mathbb{S}$, $y \in \mathbb{C}_{s_2}$: $G(s_2, y) = F(s_2, y, G)$ then* $\mathrm{FixAcc}_F(s_1, x, a_{m(s_1,x)}) = G(s_1, x)$

This justifies to write

$$\mathrm{Fix}(F)(s,x) = \mathrm{FixAcc}_F(s, x, a_{m(s,x)}) = F(s, x, \mathrm{Fix}(F))$$

where $a_{m(s,x)}$ is any proof tree of $Acc(m(s,x))$ and always exists because R is well-founded. It also yields the well-founded family induction principle:

If for all s_1, x :
  if for all $s_2, y : R(m(s_2,y), m(s_1,x))$ implies $Q(y)$
  then $Q(x)$
then forall $s, x : Q(x)$

where Q is a predicate on elements of $\mathbb{C}$.

PROOF Statements 1 and 2 are both proven by using the well-founded induction principle (c.f. Lemma 31) on $a_{m(s,x)}$ and then follow by simple unfolding of definitions. In the constructive logic of Coq, the well-founded family induction principle exists because propositions are types and recursion and induction coincide. In classical logic the principle is generated by instantiating Fix with $P = (\{y \in \mathbb{C}_s \mid y = x \text{ and } Q(y)\}_{x\in\mathbb{C}_s})_{s\in\mathbb{S}}$ and $F(s,x,f) = x$. Whenever F has a non-empty domain, its argument f exists. Now for any given s_1, x and s_2, y with $R(m(s_2,y), m(s_1,x))$, f computes an element of P, which implies $Q(y)$. Therefore the induction hypothesis is applicable, and $Q(x)$ is true, which means x is an element of P. Thus F is well-defined and existence of $\mathrm{Fix}(F)(s,x)$ as an element of P proves $Q(x)$. ■

Lemma 36 (Canonical Algebra Morphism) *Given an I-indexed $\leq_{\mathbb{S}}$-subsorted signature family Σ, $\mathbb{F}^\Sigma$-Algebras $\mathscr{A}_1^\Sigma = (\mathbb{C}, h)$, $\mathscr{A}_2^\Sigma = (\mathbb{C}, g)$, $\mathbb{F}^\Sigma$-CoAlgebra $\underline{\mathscr{A}}_1^\Sigma = (\mathbb{C}, h^{-1})$, a set M, well-founded relation $R \subseteq M \times M$, and a measuring map $m : \prod_{s\in\mathbb{S}} C_s \to M$ decreasing on h^{-1} s.t.*

for all $s \in \mathbb{S}, x \in \mathbb{C}_s, (i, o, \mathrm{args}) = h_s^{-1}(x), n \in 1, 2, \ldots, \mathrm{arity}_i(o)$:
  $R(m(\pi_n(\mathrm{dom}_i(o)), \pi_n(\mathrm{args})), m(s,x))$

define the canonical algebra morphism family

$$\mathrm{interpret} : (\mathbb{C}_s \to \mathbb{D}_s)_{s\in\mathbb{S}}$$
$$\mathrm{interpret}_s(x) = \mathrm{Fix}(\lambda(s,x,f).(g_s \circ \mathbb{F}^\Sigma(f)_s \circ h_s^{-1})(x))(s,x)$$

Now, if h and h^{-1} cancel out, i.e. for all $s \in \mathbb{S}$, $x \in \mathbb{C}_s$, and $y \in \mathbb{F}^{\Sigma}(\mathbb{C})_s$: $h_s(h_s^{-1}(x)) = x$ and $h_s^{-1}(h_s(y)) = y$, the following diagram commutes for all sorts $s \in \mathbb{S}$:

$$\begin{tikzcd}
\mathbb{F}^{\Sigma}(\mathbb{C})_s \arrow[r, "\mathbb{F}^{\Sigma}(\text{interpret})_s"] \arrow[d, bend left, "h_s"] & \mathbb{F}^{\Sigma}(\mathbb{D})_s \arrow[d, "g_s"] \\
\mathbb{C}_s \arrow[r, "\text{interpret}_s"] \arrow[u, bend left, "h_s^{-1}"] & \mathbb{D}_s
\end{tikzcd}$$

In detail this means the following statements are true:

1. $\text{interpret}_s = g_s \circ \mathbb{F}^{\Sigma}(\text{interpret})_s \circ h_s^{-1}$
2. interpret *is an algebra morphism family, i.e.* $\text{interpret}_s \circ h_s = g_s \circ \mathbb{F}^{\Sigma}(\text{interpret})_s$

Thereby the canonical algebra morphism family is:

3. *Unique, i.e. if $m : (\mathbb{C}_s \to \mathbb{D}_s)_{s \in \mathbb{S}}$ is an algebra morphism family for h and g then $m = \text{interpret}$*
4. *Sound, i.e. for all $x \in \mathbb{C}_s$:* $\text{interpret}_s(x) \in \mathbf{AlgGen}_s^{\mathscr{A}_2^{\Sigma}}$
5. *Complete, i.e. for all $y \in \mathbf{AlgGen}_s^{\mathscr{A}_2^{\Sigma}}$: exists $x \in \mathbb{C}_s$, s.t.* $\text{interpret}_s(x) = y$

PROOF Statement 1 follows from unfolding the definition of interpret and Lemma 35.1. Statement 2 is proven by applying Statement 3 and using that h and h^{-1} cancel out. Statement 3, is shown by replacing interpret using statement 1. The new goal is proven by the well-founded family induction principle. Using the algebra morphism property of m, and cancelation of h and h^{-1}, it is sufficient to show $\mathbb{F}^{\Sigma}(\text{interpret})_s \circ h_s^{-1}(x) = \mathbb{F}^{\Sigma}(m)_s \circ h_s^{-1}(x)$. This follows from commutativity of the canonical morphism family (statement 1), the induction hypothesis, and the decreasing property of m. In the Coq formalization, statements 1 – 3 are slightly weaker and only claim intensional (the functions cannot be distinguished by their input/output behavior) equality. Practically, this means their may always be faster or slower implementations, but they all produce the same results. In traditional Set theory this distinction cannot be made and is therefore omitted in the statement. Statement 4, is again shown by replacing interpret using statement 1 and application of the well-founded family induction principle. It then follows by another application of statement 1, the induction hypothesis, and the decreasing property of m. The carrier element for statement 5 is constructed by induction on the derivation of the proof that x is algebraically generated. An element of $y \in \mathbb{F}^{\Sigma}(\mathbb{C})_s$ with $x = g_s(\mathbb{F}^{\Sigma}(\text{interpret})_s(y))$ is constructed from the index and operation of the $\mathbb{F}^{\Sigma}(\mathbb{D})_s$ element given in the inductive case. Its arguments are provided by the induction hypothesis. Now by the algebra morphism property shown in statement 2, $\text{interpret}_s(h_s(y)) = g_s(\mathbb{F}^{\Sigma}(\text{interpret})_s(y))$, and so $h_s(y)$ is the carrier element for which existence needed to be proven. ∎

At this point, category theory proficient readers will find Lemma 36 reminiscent of Lambek's Lemma [121] and its applications [84]. However, the construction does the opposite. Lambek's Lemma proves that functors with an initial algebra (there is a unique algebra morphism to any other algebra) are fixpoints [1]. Here, conversely, a (family of) fixpoint(s) is used to prove that a given algebra is initial by providing the (family of) canonical algebra morphism(s). The algebra is chosen to have terms typable by Combinatory Logic as its carrier, and the algebra morphism translates them to terms of any algebraically generated language. Meyer [138] clarifies the notion of models for Combinatory Logic with combinators S and K as their base. Inversely, the next construction finds a base for a given model (signature and algebra).

Lemma 37 (Finite Combinatory Logic (Co)Algebra) *Given a $\leq_{\mathbb{S}}$-subsorted signature family $\Sigma_{i\in I} = (\mathbb{S}, \mathbb{O}, \mathrm{arity}_i, \mathrm{dom}_i, \mathrm{range}_i)$ indexed over a finite set I, define:*

- *The set of applicative terms $\mathbb{A}$ formed over combinator base $\mathbf{B} \ni x ::= o \mid \mathrm{idx}_i$ for all $i \in I$ and $o \in \mathbb{O}$*
- *The set of types intersection types $\mathbb{T}$ formed over constructors $\mathbf{C} \ni c ::= s \mid \mathrm{idx}_i$ for all $i \in I$ and $s \in \mathbb{S}$ preordered by $\leq_{\mathbf{C}} = \leq_{\mathbb{S}} \uplus \{(\mathrm{idx}_i, \mathrm{idx}_i) \mid i \in I\}$*
- *The index context $\Gamma_I : I \to \mathbb{T}$ by $\Gamma_I(i) = \mathrm{idx}_i(\omega)$*
- *The sort embedding function $\mathrm{embed} : \mathbb{S} \to \mathbb{T}$ by $\mathrm{embed}(s) = s(\omega)$*
- *The sort unembedding function* $\mathrm{unembed}(A) = \begin{cases} (1, s) & \textit{for } A = s(\omega) \textit{ and } s \in \mathbb{S} \\ (0, ()) & \textit{otherwise} \end{cases}$
- *Function $\mathrm{typeAtIndex} : \mathbb{O} \times I \to \mathbb{T}$ by*

$$\begin{aligned} &\mathrm{typeAtIndex}(o, i) = \\ &\quad \Gamma_I(i) \to \mathrm{mkArrow}(\mathrm{rev}(\mathrm{map}(\lambda n.\mathrm{embed}(\pi_n(\mathrm{dom}_i(o))), \mathrm{enum}(\{1, 2, \ldots, \mathrm{arity}_i(o)\}))), \\ &\qquad\qquad \mathrm{embed}(\mathrm{range}_i(o))) \end{aligned}$$

- *Function $\Gamma_\Sigma : \mathbb{O} \to \mathbb{T}$ by $\Gamma_\Sigma(o) = \bigcap_{i \in \mathrm{enum}(I)} \mathrm{typeAtIndex}(o, i)$*
- *Context $\Gamma : \mathbf{B} \to \mathbb{T}$ by* $\Gamma(x) = \begin{cases} \Gamma_I(i) & \textit{for } x = \mathrm{idx}_i \\ \Gamma_\Sigma(x) & \textit{otherwise} \end{cases}$
- *The Combinatory Logic carrier $\mathbb{C}_s = \{M \in \mathbb{A} \mid \Gamma \vdash M : \mathrm{embed}(s)\}$*
- *The Combinatory Logic $\mathbb{F}^\Sigma$-Algebra $\mathscr{A}^\Sigma_{\mathrm{FCL}} = (\mathbb{C}_s, h)$ with action*

$$h_s(i, o, \mathrm{args}) = \mathrm{revApply}(@(o, \mathrm{idx}_i), \mathrm{rev}(\mathrm{map}(\lambda n.\pi_n(\mathrm{args}), \mathrm{enum}(\{1, 2, \ldots, \mathrm{arity}_i(o)\}))))$$

- *The Combinatory Logic* $\mathbb{F}^{\Sigma}$*-CoAlgebra* $\underline{\mathscr{A}}^{\Sigma}_{\mathrm{FCL}} = (\mathbb{C}_s, h)$ *with co-action*

$$h_s^{-1}(M) = (i, o, (\mathrm{arg}_1, \mathrm{arg}_2, \ldots, \mathrm{arg}_{\mathrm{arity}_i(o)}))$$
$$\textit{for } \mathrm{unapply}(M) = (o, Ns) \textit{ and } \mathrm{rev}(Ns) = [::\mathrm{idx}_i \,\&[::\mathrm{arg}_1 \,\&[::\mathrm{arg}_2 \,\&\ldots[::\mathrm{arity}_i(o)]]]]$$

Which is possible because the following statements are true for all $i \in I$, $o \in \mathbb{O}$, *and* $s, s_1, s_2 \in \mathbb{S}$, $M \in \mathbb{A}$, *and* $Ns, Ns' \in \mathbb{A}^*$, $M_s \in \mathbb{C}_s$, $t \in \mathbb{F}^{\Sigma}(\mathbb{C})_s$, *and* $A_1, A_2 \in \mathbb{T}$, *and* $c_1, c_2 \in \mathbf{C}$, *and* $\mathrm{srcs}_1, \mathrm{srcs}_2 \in \mathbb{T}^*_{\pi}$*:*

1. $\mathrm{unembed}(\mathrm{embed}(s)) = (1, s)$
2. *If* $s_1 \leq_{\mathbb{S}} s_2$ *then* $\mathrm{embed}(s_1) \leq \mathrm{embed}(s_2)$
3. $\Gamma \vdash h_s(t) : \mathrm{embed}(s)$
4. $\mathrm{unapply}(M_s).1 \neq \mathrm{idx}_i$
5. *If* $\mathrm{mkArrow}(\mathrm{srcs}_2, c_2(A_2)) \leq \mathrm{mkArrow}(\mathrm{srcs}_1, c_1(A_1))$ *then* $c_2(A_2) \leq c_1(A_1)$ *and* $\mathrm{lsize}(\mathrm{srcs}_1) = \mathrm{lsize}(\mathrm{srcs}_2)$ *and* *for all* $(\mathrm{src}_1, \mathrm{src}_2)$ *in* $\mathrm{zip}(\mathrm{srcs}_1, \mathrm{srcs}_2)$*:* $\mathrm{src}_1 \leq \mathrm{src}_2$
6. *If* $\Gamma \vdash M : \mathrm{idx}_i(\omega)$ *then* $M = \mathrm{idx}_i$
7. *If* $\mathrm{rev}((\mathrm{unapply}(M_s)).2) = [::N\&Ns]$ *then there exists* $i \in I$ *s.t.* $N = \mathrm{idx}_i$
8. *If* $\mathrm{unapply}(M_s) = (o, Ns)$ *and* $\mathrm{rev}(Ns) = [::\mathrm{idx}_i \,\&Ns']$ *then* $\mathrm{lsize}(Ns') = \mathrm{arity}_i(o)$
9. *If* $\mathrm{unapply}(M_s) = (o, Ns)$ *and* $\mathrm{rev}(Ns) = [::\mathrm{idx}_i \,\&Ns']$ *then* *for all* $1 \leq n \leq \mathrm{lsize}\, Ns'$*:* $\Gamma \vdash \mathrm{nth}(Ns', n-1) : \pi_n(\mathrm{dom}_i(o))$
10. *If* $\mathrm{unapply}(M_s) = (o, Ns)$ *and* $\mathrm{rev}(Ns) = [::\mathrm{idx}_i \,\&Ns']$ *then* $\mathrm{range}_i(o) \leq_{\mathbb{S}} s$

PROOF Statements 1 and 2 immediately follow from the definitions. Statement 3 guarantees that the algebra action h_s is well-defined and follows from the subtype rule ($\leq$) of FCL Definition 12 and generalized application (Lemma 15.1). The other statements are required to ensure that the co-action h_s^{-1} is well defined. Statement 4 follows from inverse application (Lemma 15.2), minimal type correctness (Lemma 16), subtype machine corretness (Theorem 1), and case-analysis on the subtype machine rules (Definition 5). The proof of statement 5 is by inverse induction on the sources srcs_1 and srcs_2, again followed by case-analysis on the subtype machine rules. Statement 6 is shown by normalized induction (Lemma 10) on the derivation and inverse application, minimal type correctness, primality (c.f. Lemma 8) of types in Γ, and statement 5 are used to discharge the applicative case. An analogous proof is used for statements 7, just with inverse application instead of normalized induction. This is repeated for the other statements 8–10, additionally using statements 4 and 7 in inverse induction on Ns followed by inverse induction on the types obtained by applying the inverse application lemma. ■

It remains to show, that $\mathscr{A}^{\Sigma}_{\mathrm{FCL}}$ and $\underline{\mathscr{A}}^{\Sigma}_{\mathrm{FCL}}$ give rise to a canonical algebra morphism family into any other algebra.

Lemma 38 (Finite Combinatory Logic Algebra Morphism Conditions) *For $\mathscr{A}^{\Sigma}_{\mathrm{FCL}}$ and $\underline{\mathscr{A}}^{\Sigma}_{\mathrm{FCL}}$ from Lemma 37 define the forgetful measure map*

$$\mathrm{measure}_{\mathrm{FCL}} : \prod_{s \in \mathbb{S}} \mathbb{C}_s \to \mathbb{A}$$
$$\mathrm{measure}_{\mathrm{FCL}}(s, M) = M$$

and the least relation $\mathrm{IsChild} \subset \mathbb{A} \times \mathbb{A}$ *closed under rule*

$$\frac{M \textit{ is in } (\mathrm{unapply}(N)).2}{\mathrm{IsChild}(M, N)}$$

Then for all predicates P on applicative terms, $M \in \mathbb{A}$, $s \in \mathbb{S}$, $M_s \in \mathbb{C}_s$, $o \in \mathbb{O}$, $i \in I$, $xs \in \prod_{n=1}^{\mathrm{arity}_i(o)} \pi_n(\mathrm{dom}_i(o))$, and $t \in \mathbb{F}^{\Sigma}(\mathbb{C})_s$ the following statements are true:

1. *If for all $c \in \mathbf{B}$ and Ns in $\mathbb{A}^*$:*

   for all N in Ns : $P(N)$ implies $P(\mathrm{revApply}(c, Ns))$

   then $P(M)$

2. *Relation* IsChild *is well-founded:* WF(IsChild)

3. *If $h_s^{-1}(M_s) = (i, o, xs)$ then*
   $\mathrm{IsChild}(\mathrm{measure}_{FCL}(\pi_n(\mathrm{dom}_i(o)), \pi_n(xs)), \mathrm{measure}_{FCL}(s, M_s))$

4. $h_s^{-1}(h_s(t)) = t$

5. $h_s(h_s^{-1}(M_s)) = M_s$

PROOF Statement 2 is an induction principle on applicative terms, which is proven by rewriting the result $P(M)$ to $P(\mathrm{revApply}(M, [::]))$ using Lemma 14.2 and then generalizing to "if N in $[::]$ implies $P(N)$, then $P(\mathrm{revApply}(M, [::]))$" and further to "if N in Ns implies $P(N)$, then $P(\mathrm{revApply}(M, Ns))$" for arbitrary $Ns \in \mathbb{A}^*$. Induction on M is now enough to show the result, using Lemma 14.1 in the case for applications. Statement 2 easily follows from the induction principle in statement 2 and Lemma 14.3. The other statements follow by unfolding the definition of the (co-)action with an additional inverse induction on Ns computed by $h_s^{-1}(M_s) = (i, o, Ns)$ to prove the last statement. ■

With the measure map into the well-founded IsChild relation and the cancelation properties, everything is set up to show the final theorem about uniqueness, soundness and completeness of the translation from Combinatory Logic into the algebraically generated set of any given algebra for any given signature.

Theorem 7 (Canonical Algebra Morphism for the Combinatory Logic (Co)Algebra) *Given a* $\leq_{\mathbb{S}}$*-subsorted signature family* $\Sigma_{i \in I} = (\mathbb{S}, \mathbb{O}, \text{arity}_i, \text{dom}_i, \text{range}_i)$ *indexed over a finite set* I*, and a target language implementation of* Σ *in form of an algebra* $\mathscr{A}^{\Sigma} = (\mathbb{D}, g)$*, let* $\text{interpret}^{\text{FCL}} : (\mathbb{C} \to \mathbb{D})_{s \in \mathbb{S}}$ *be the canonical algebra morphism family constructed for* $\mathscr{A}^{\Sigma}_{\text{FCL}}$ *and* $\underline{\mathscr{A}}^{\Sigma}_{\text{FCL}}$ *from Lemma 37 and the well-founded relation* IsChild*, and measure map* $\text{measure}_{\text{FCL}}$ *from Lemma 38. Now for all* $s \in \mathbb{S}$*,* $M_s \in \mathbb{C}_s$*,* $T_s \in \mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}_s$*, and* $m : (\mathbb{C} \to \mathbb{D})_{s \in \mathbb{S}}$ *the algebra morphism family* $\text{interpret}^{\text{FCL}}$ *is*

1. *Unique:*
   if m *is an algebra morphism family* $m_s \circ h_s = g_s \circ \mathbb{F}^{\Sigma}(m)_s$*, then* $m = \text{interpret}^{\text{FCL}}$.
2. *Sound:*
   $\text{interpret}^{\text{FCL}}(M_s) \in \mathbf{AlgGen}^{\mathscr{A}^{\Sigma}}_s$.
3. *Complete:*
   There exists $N_s \in \mathbb{C}_s$*, s.t.* $\text{interpret}^{FCL}_s(N_s) = T_s$.

PROOF Immediate consequences of the abstract canonical algebra morphism properties shown in Lemma 36. Uniqueness is again intensional in the Coq formalization. ■

Theorem 7 enables the following workflow for language-agnostic synthesis:

1. Define a code generator for some target language $\mathbb{D}$ by an interface Σ and its implementation $\mathscr{A}^{\Sigma}$.
2. Convert the interface into a repository using the algorithm from Lemma 37.
3. Optionally add another semantic domain of discourse using the methods described in Section 2.4.2.
4. Perform type inhabitation according to Theorem 6 with the desired (embedded) sort (and an optional semantic domain type) as target.
5. Now any term in the language of the resulting Tree Grammar can mapped back using $\text{interpret}^{\text{FCL}}$ from Theorem 7.

Uniqueness, soundness, and completeness from Theorem 7 guarantee that the Tree Grammar computed by inhabit is a finite and exact representation of all solutions to the synthesis problem. Emptiness of Tree Grammars is decidable in linear time [42], which means the above procedure also solves the sort emptiness problem. A fully materialized example, which is adapted from [18], can be found in file `Labyrinth.v` of the accompanying formalization [15]. If desired, language specific proofs can be added to ensure coincidence of $\mathbf{AlgGen}^{\Sigma}$ with some set. This is illustrated for the 2-Counter Automata reachability in Lemma 34. If the semantic domain requires formal reasoning, it can also be created from a translated signature family. Note, that $\mathscr{A}^{\Sigma} = (\mathbb{D}, g)$ is freely chosen. Therefore, the target language $\mathbb{D}$ and the meta-language implementing g are parameters, which makes the procedure truly language-agnostic.

Chapter 3

The (CL)S Framework

The (CL)S Framework implements the theoretical results described in Section 2. At the time of writing the framework has been under development for over 7 years. The earliest prototype has been implemented in Prolog and a subsequent incarnation was ported to F#, mainly for performance and maintainability reasons [56; 55]. This implementation has been continued evaluating multiple ideas including availability as a web-service and modal intersection types [59; 20]. Dudenhefner [61] describes the development of its latest incarnation, cls-fsharp [62], with a focus on performance optimizations, mainly by making smart choices when substituting type variables, implementing and extending the theoretical results presented in [58; 67]. The ideas presented in [20] also lead to another branch of development, cls-scala [111], that is implemented in Scala and described in this section. In contrast to cls-fsharp, which is built for evaluating performance gains from different inhabitation strategies, its focus is on practical concerns of software composition synthesis. These include an embedded domain specific language (EDSL) to develop components, built-in support for automatically translating and executing synthesized applicative terms, features to enable language-independent meta-programming, versioned hosting of synthesized artifacts, seamless integration into the Scala ecosystem via standard build-system dependencies, model driven development capabilities, and most recently [18] the capability to visually inspect the synthesis process in order to debug specifications. The architecture of cls-scala is modular and separated into multiple projects. It is described in Section Section 3.1. All of its components are open source and available online together with evolving ecosystem of related projects [27]. Notably, some of these just make use of the metaprogramming features without using inhabitation, e.g. EpCoGen [97], a framework for generating and studying approaches to solve the expression problem [199]. The extension to Scala with an EDSL for specifying software components (Section 3.2), meta-programming support mechanisms and features for versioned artifact hosting have been developed (Section 3.3) in step with the requirements arising from work on software product lines [96; 21; 60; 95; 164]. The implementation is closely connected to its theory described in Chapter 2 and accompanied by a suite of tests is described in Section 3.4. Finally it should not go unmentioned that the proven Coq formalization [15] of Chapter 2 can be extracted

to executable Haskell or OCaml code. It can thus serve as a verified baseline to investigate future bugs of cls-scala. Additionally, if an increased level of confidence is required for some application, it can be used to formally prove correctness of synthesis results.

The Coq formalization is an independent work of the author of this text. Most code of the cls-scala framework has been created by the author, with some contributions from others. The most notable other direct contributors of source code are George T. Heineman and Anna Vasileva. Many design decisions were also made in team with Boris Düdder, whose prior experience developing earlier versions of the framework was invaluable. All projects are publicly available [27] and their source commit history can be traced author by author.

3.1 Architecture Overview

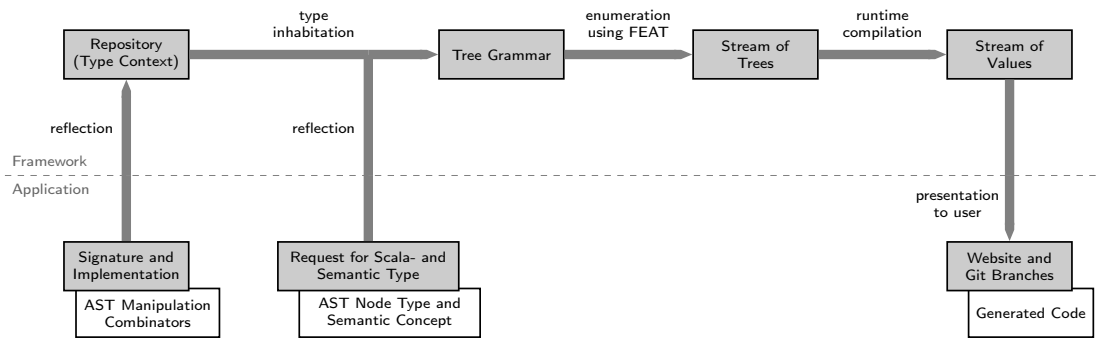

Figure 3.1: Data flow in cls-scala

Figure 3.1 shows the typical data flow when using cls-scala. An application implements components in Scala as combinators that provide or manipulate fragments of abstract syntax trees (ASTs) of a target language. Problem instance specific boxes are drawn in white: combinators can manipulate any Scala objects, and ASTs are just one use-case that is often encountered. The signature of components and their semantic types are collected using reflection and translated into a single type context. A request is made, typically for the type of an AST node and a semantic type. Type inhabitation creates a Tree Grammar describing all applicative solution terms. The Tree Grammar is enumerated as a continuous stream of data using the functional enumeration of algebraic types (FEAT) technique by Duregård et al. [68]. Upon request each inhabitant is translated to Scala using the canonical algebra morphism constructed in Section 2.6.3. The translated term is then runtime-compiled and injected back into the JVM where it is executed. Results are available as a stream. If the results can be serialized, infrastructure is provided to store them in a Git [189] repository and to inspect and control their creation via a website.

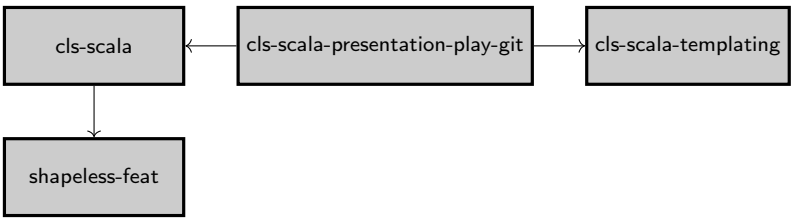

Figure 3.2: Projects in the cls-scala framework and their dependencies

Figure 3.2 shows the main projects of the framework with arrows indicating their interdependencies. Stages above the dotted line of Figure 3.1 are implemented in the cls-scala project. Its implementation of FEAT is separated into a different project. The shapeless-feat project is a port of the original Haskell library [68] with additional integration into the shapeless framework [169], which provides data-type generic programming [128] features beyond the scope

of this text. Separation into a different project enables reuse independent of (CL)S. Similarly, the template-based metaprogramming infrastructure is provided as a separate project. Type inhabitation and its algebraic interpretation works independently of any metaprogramming support, so there is no direct dependency of cls-scala on cls-scala-templating. Vice versa, the templating support is generic enough not to depend on cls-scala. If results are source code artifacts that can be serialized to disk, the cls-scala-presentation-play-git project can host them. It uses the Play framework [33] and its integrated web-server to present users with a website to manage repositories. Git access and hosting is provided using EGit [203].

All projects are built using the sbt build system [93]. Continuous integration provided by Travis [80] automatically builds each project, executes its test-suite and triggers the early release [198] build target. This pushes a snapshot of every successfully tested master branch commit to maven central [11], from where the project can be immediately included as a jar-file dependency into other projects. The sbt build system has cross building support to create, test and publish jar-files for different Scala versions. At the time of writing Scala 2.11 and 2.12 are supported. The coveralls service [137] automatically collects test coverage reports from each build and publishes them with a detailed analysis.

The Software and sources are released under a permissive open source Apache 2 license [4]. This allows free and commercial, open and closed-source projects to include all or parts of the software with limited warranty, liability and possibility of patent-claims against the authors.

3.2 Scala Extensions and their Relation to the Coq-Formalization

The most obvious user-facing extension Scala by cls-scala is an EDSL to specify components used in synthesis. It consists of two parts. The first is a syntax extension to specify (semantic) intersection types in a notation close to the way they are denoted in math. This way type

$$\mathsf{Map}(\mathsf{Person} \cap \mathsf{Owner} \star \mathsf{Pair}(\mathsf{Cat} \star \mathsf{Dog})) \to \mathsf{Person} \cap \mathsf{Owner} \to \mathsf{Either}(\omega \star \mathsf{Pair}(\mathsf{Cat} \star \mathsf{Dog}))$$

can be written

```
'Map('Person :&: 'Owner, 'Pair('Cat <*> 'Dog)) =>:
'Person :&: 'Owner =>:
'Either(Omega, 'Pair('Cat <*> 'Dog))
```

The syntax extensions are activated by importing the implicit conversions defined in package `org.combinators.cls.types.syntax`. Symbol literals in Scala are converted into type constructors. Without arguments, constructor `'C` will be interpreted as type $C(\omega)$, just like in the mathematical notation convention. Constructors with one or more argument will be desugared to the binary product of the arguments. Alternatively, the binary product operator `'Cat <*> 'Dog` can be used directly to specify $\mathsf{Cat} \star \mathsf{Dog}$. Arrows $A \to B$ are represented by `'A =>: 'B` and intersections $A \cap B$ by `'A :&: 'B`. Type constant ω is the constant `Omega` in Scala. All binding rules of Section 2.2 Definition 3 apply. Users not fond of operator overloading (e.g. because of working within a code generator) can alternatively define types manually instantiating case classes `Constructor(String, Type)`, `Product(Type, Type)`, `Arrow(Type, Type)`, and `Intersection(Type, Type)`.

The constructor subtype relation $\leq_{\mathbf{C}}$ is constructed from case class `SubtypeEnvironment(Map[String, Set[String]])`. The map passed as argument stores all smaller constructors for a given constructor name. Instances of `SubtypeEnvironment` compute the transitive reflexive closure of their argument relation. Their contents are imported to add extension methods to trait `Type`, which is implemented by all intersection types. These allow types to be compared according to the BCD subtype $\leq$ relation induced by $\leq_{\mathbf{C}}$. A builder pattern [81] is used to provide some sugar for constructing the $\leq_{\mathbf{C}}$ relation map. Practically, constructor preorder

$$\begin{aligned}\leq_{\mathbf{C}} = \{&(\mathsf{Cat}, \mathsf{Animal}), (\mathsf{Garfield}, \mathsf{Cat}), (\mathsf{Dog}, \mathsf{Animal}), (\mathsf{Odie}, \mathsf{Dog}),\\ &(\mathsf{Animal}, \mathsf{Animal}), (\mathsf{Cat}, \mathsf{Cat}), (\mathsf{Garfield}, \mathsf{Garfield}), (\mathsf{Odie}, \mathsf{Odie}), (\mathsf{Garfield}, \mathsf{Animal}), (\mathsf{Odie}, \mathsf{Animal})\}\end{aligned}$$

is defined by

```
val env =
  SubtypeEnviroment(
    Taxonomy("Animal").addSubtype("Cat").addSubtype("Dog")
```

```
      .merge(Taxonomy("Cat").addSubtype("Garfield"))
      .merge(Taxonomy("Dog").addSubtype("Odie")).underlyingMap)
```

after which variable `results` in

```
import env._
val results =
  ('Pair('Garfield <*> 'Odie).isSubtypeOf('Pair('Cat <*> Animal)),
   'Pair('Garfield <*> 'Odie)
     .isSupertypeOf('Pair('Animal <*> Animal)))
```

contains `(true, false)`. The subtype algorithm is implemented using the machine from Section 2.3 Definition 9.

The Scala implementation allows for finite type schematism. Types are extended with variables and a finite set of well-formed substitutions is declared. When a combinator uses `ownerType` declared in

```
val alpha = Variable("alpha")
val beta = Variable("beta")
val substitutionSpace =
  Kinding(alpha).addOption('Garfield).addOption('Odie)
    .merge(Kinding(beta).addOption('Jon))
val ownerType = 'FeedsWithLasagna(alpha, beta)
```

its use will be expanded to

```
'FeedsWithLasagna('Garfield, 'Jon)
  :&: 'FeedsWithLasagna('Odie, 'Jon)
```

Well-formed substitution spaces can be any finite collection of Scala functions `Variable => Type` and `Kinding` again implements a builder pattern to simplify their construction.

The most basic way to perform type inhabitation is to use an instance of `InhabitationAlgorithm` according to the following type definitions from package `org.combinators.cls.inhabitation`

```
type Repository = Map[String, Type]
type TreeGrammar = Map[Type, Set[(String, Seq[Type])]]
type InhabitationAlgorithm =
  (FiniteSubstitutionSpace, SubtypeEnvironment, Repository) =>
  Seq[Type] => TreeGrammar
```

Inhabitation algorithms take a finite substitution space, a subtype environment, a repository, and a sequence of types to inhabit. They then produce a Tree Grammar representing all requested inhabitants. Substitutions spaces and subtype environments have been described before. Repositories are just maps assigning each combinator a type. In contrast to the Coq formalization, Tree Grammars in the Scala implementation use uncurried representations

(c(M,N) instead of $@(@(c,M),N)$) of applicative terms. This is equivalent except for target type ω, which cannot be represented. Behavior of the Scala implementation is, at the time of writing, undefined for this case, which is rare enough that it only became apparent during the rigorous formalization. The Scala implementation can schedule multiple inhabitation targets in a batch job, resulting in one big Tree Grammar. This is justified by Section 2.5 Lemma 29.5, and the various weakening properties in Lemma 33. It can lead to performance improvements, if the recursive target types in batched inhabitation requests overlap, allowing the algorithm to reuse parts of the previously constructed Tree Grammar. There are two implementations of type `InhabitationAlgorithm`, both contained in package `org.combinators.cls.inhabitation`. The first one by class `BoundedCombinatoryLogic` performs all substitutions and then dispatches to the second one by class `FiniteCombinatoryLogic`, which performs inhabitation for a context without type variables. Implementations of splitTy (c.f. Section 2.5 Definition 26) and the cover machine (c.f. Section 2.5 Definition 21) match the Coq formalization up to minor language differences. The function for inhabitation_step, additionally to using uncurried applicative terms, accumulates covers (c.f. Section 2.5 Definition 29) for multiple combinators in parallel by turning the foldl into a map-reduce [122] operation. The resulting performance benefit has been discussed at length by Düdder [55] and Scala has parallel collections in its standard library [161], which allow the algorithm to be kept almost unchanged. The implementation as a state machine operating with single invocations of inhabitation_step makes it possible to debug the inhabitation process for a given repository in order to find out, which targets are dropped for being uninhabited. See [18] for a detailed description of a debugger implemented on top of cls-scala.

The extensions described up to this point are similar to the capabilities of cls-fsharp and earlier incarnations of the (CL)S framework. The cls-scala implementation is made unique by the second extension, which builds upon the algebraic theory of Section 2.6. It allows to specify Section 2.4.2 Example 1 as shown in Listing 3.1.

```
class MotivationRepository {
  val performer: Variable = Variable("performer")
  val defaultPerformerSpace =
    Kinding(performer).addOption('Springsteen)

  @combinator object motivationSong {
    def apply(): LilyPond[Music] =
      LilyPond.loadMusic("noretreat.ly")
    val semanticType: Type = 'Springsteen
  }
  @combinator object addLyrics {
    def apply(music: LilyPond[Music],
              lyrics: String): LilyPond[Song] =
      music.setLyrics(lyrics)
    val semanticType: Type = performer =>: performer =>: performer
  }
  class LyricsCombinator(lyrics: String, performerName: String) {
```

```
18      def apply(): String = lyrics
19      val semanticType: Type = Constructor(performerName)
20    }
21
22    def reflectedRepository:
23        ReflectedRepository[MotivationRepository] = {
24      val crawler = CrawlerDemo
25      val (performerSpace, lyricsCombinators) =
26        crawler
27        .findLyrics("surrender")
28        .foldLeft((defaultPerformerSpace,
29                   Seq.empty[LyricsCombinator])) {
30          case ((s, cs), (lyrics, performerName)) =>
31            (s.addOption(Constructor(performerName)),
32             cs :+ new LyricsCombinator(lyrics, performerName))
33        }
34      lyricsCombinators.foldLeft(
35        ReflectedRepository[MotivationRepository](
36          substitutionSpace = performerSpace,
37          inst = this,
38          classLoader = this.getClass.getClassLoader)) {
39          case (repo, combinator) => repo.addCombinator(combinator)
40        }
41    }
42
43    def results: InhabitationResult[LilyPond[Song]] =
44      reflectedRepository.inhabit[LilyPond[Song]]('Springsteen)
45  }
```

Listing 3.1: Definition of Section 2.4.2 Example 1 in cls-scala

Each synthesis component is declared in a combinator object. Combinator objects have a single apply-method and an optional field `semanticType`. They can either be static objects belonging to a class and carrying the annotation `@combinator` (e.g. `motivationSong` in line 6) or classes (e.g. `LyricsCombinator` in line 17). Methods in combinators may contain arbitrary code. This is especially useful when, as shown here, program sources of some target programming language are manipulated. Method `reflectedRepository` uses reflection to construct a context for inhabitation. The results of a web-crawler searching for lyrics are converted into new combinators and substitution space options in lines 25–33. Newly added substitution space options allow the semantic type of combinator `addLyrics` to grow with each new performer (c.f. line 15). The inhabitation context is created in lines 34–40. First, a `ReflectedRepository` for the current instance of class `MotivationRepository` is constructed with the extended substitution space. The Java Virtual Machine class loader is used to obtain reflection information about static combinators belonging to the instance reflected upon. An algebraic signature gets extracted from this information according to the signatures of all apply methods in `@combinator` annotated inner objects. Scala types become sorts of the

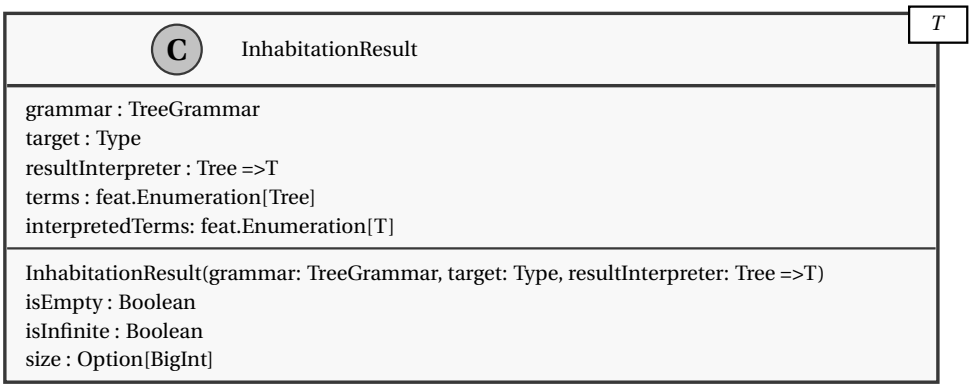

Figure 3.3: Class to encapsulate inhabitation results

signature, which are ordered by their language native subtype relation. Additionally, the semantic type information is included (Section 2.4.2 explains how and why native and semantic types may be freely mixed). If required, semantic types can also be ordered by a subtype environment passed to the `ReflectedRepository` constructor. Webcrawler results cannot be known statically at compile time. Their combinators are added dynamically to the repository at run time (line 39). Signature operations for static combinators are just their object names, while dynamic combinators are suffixed with a unique identifier, which allows to have multiple instances of the same class inserted as multiple different combinators. Standard Scala objects and classes are used for component specifications. Therefore, combinator implementations can be shared using inheritance and other object-oriented features, and the learning curve for users not familiar with algebra or Combinatory Logic is reduced. Line 44 shows the request for all inhabitants that have native type `LilyPond[Song]` and the semantic type Springsteen. Results are encapsulated in class `InhabitationResult` shown in Figure 3.3. It includes the Tree Grammar, the requested target type, and in the constructor `ReflectedRepository` also passes the canonical algebra morphism for Combinatory Logic (Section 2.6 Theorem 7), which translates applicative Terms of type `Tree` to the native type `T`. The Scala algebra for this morphism is implemented by creating calls to the combinator objects collected by `ReflectedRepository`. Code for these calls is compiled, injected back into the class loader and then executed. Field `InhabitationResult.terms` stores a lazily computed enumeration of all terms that are words of the Tree Grammar for the target non-terminal (c.f. Section 2.5.2 Definition 28). Terms in this enumeration are ordered ascending by their number of terminal symbols. The `InhabitationResult.interpretedTerms` field stores the result of lazily mapping the translation to Scala over `InhabitationResult.interpretedTerms`. Additionally, the class offers methods to check if the Tree Grammar is empty, infinite, or if it is finite to compute the number of results. See [42] for the algorithms used in these computations.

3.3 Metaprogramming Support Mechanisms

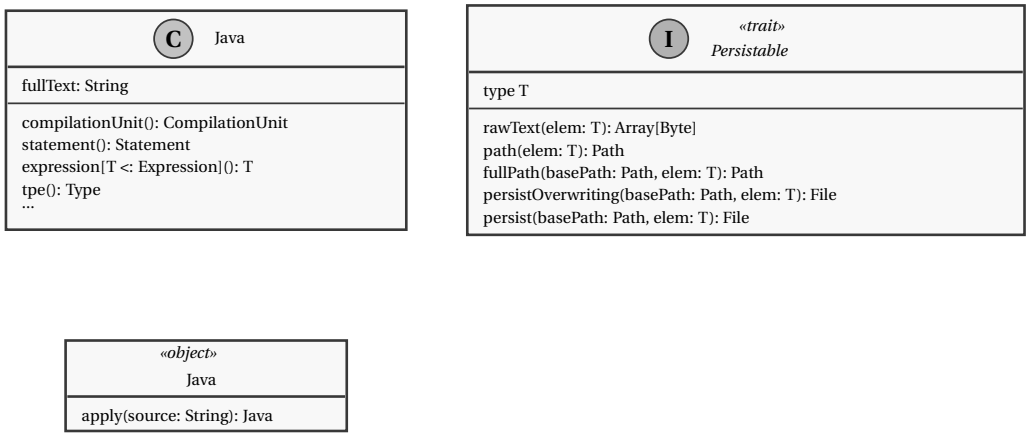

Figure 3.4: Main components of project templating for target language Java

The templating project adds lightweight metaprogramming support. Figure 3.4 shows the main components for this (exemplified for the target language Java). Class Java is a wrapper around a string to provide extension methods to invoke a parser. They are implemented using the JavaParser project [195], which offers parsing, pretty printing and AST manipulation support for Java. The most useful syntactic sorts in the AST are accessible via these methods. Together with the apply method defined in its companion object and the automatic string interpolation support of Scala, this allows to write

```
val exp = Java("42").expression()
val stmt = Java(s"System.out.println($exp);").statement()
```

which will cause variable `stmt` to reference an AST-representative of the Java statement `"System.out.println(42);"`. For the syntactic sort `CompilationUnit`, the project offers an instance of the trait `Persistable`, with `type T = CompilationUnit`. This instance helps to persist the java file represented by the `CompilationUnit` on disk. The trait offers method `rawText` to convert an element of its type `T` to a serialized byte array, method `fullPath` to determine the path of an element relative to some location, and methods to persist an element, optionally overwriting previously existing files. For Java `CompilationUnit` elements the default instance can automatically determine paths, because Java source file names and their contents are linked. For other languages, such as Python, where the source file name is not clear from the contents, type `T` has to be instantiated as a pair which includes enough information to find the file-system destination of elements. Currently, Java and Python are supported languages, but users can add their own implementations on demand in their projects. The aforementioned EpCoGen [97] has instances for Haskell, C++, and even an old Java dialect called Generic Java. The language specific extension method provider classes do not have to contain all AST elements. In fact, they can just wrap target language strings. It is up to the user to decide on the trade-off between usage comfort and implementation effort for parsers.

For the versioned hosting of artifacts, it is enough to setup a standard Play application [33] and to configure a Play webserver route to a class inheriting from `org.combinators.cls.git.InhabitationController` and trait `org.combinators.cls.git.RoutingEntries`, which are both part of cls-scala-presentation-play-git. The class has to override three fields. Listing Listing 3.2 shows the implementation for class `MotivationRepository` from Listing 3.1

```
class MotivationController @Inject()(webJars: WebJarsUtil,
  lifeCycle: ApplicationLifecycle)
    extends InhabitationController(webJars, lifeCycle)
    with RoutingEntries {
  lazy val repository = new MotivationRepository
  lazy val Gamma = repository.reflectedRepository
  override lazy val combinatorComponents:
    Map[String, CombinatorInfo] = Gamma.combinatorComponents
  override lazy val results: Results =
    EmptyResults().add(repository.results)
  override lazy val controllerAddress: String = "motivation"
}
```

Listing 3.2: Definition of a controller to host generated MotivationRepository (Listing 3.1) solutions

After the built-in Play webserver has been started, results are accessible via the address specified in field `controllerAddress` relative to the route configured in Play. For a default configuration and class `MotivationController`, the address is http://localhost:9000/motivation/. The website shows an overview of the types in the reflected repository `Gamma`. Types for the overview are provided via field `combinatorComponents`. Results are added to the website using field `results`. Its type is another class implementing a builder pattern, which can collect any `InhabitationResult[T]` type for which an implicit instance of `Persistable` for type `T` is available in the declaration context. Multiple calls to the `add` method compute the Cartesian product of results to combine all possibilities. Additionally, external artifacts which have not been synthesized can be added. This is used in EpCoGen. The full code for the example, including a `Persistable` instance for class `LilyPond`, is available online [16]. When running, Play detects any changes to the source code, and subsequently recompile and restart the application. This speeds up the development process, because refreshing the browser is enough to obtain updates on the result website. For every computed code artifact, a link to a Git branch with instructions on how to obtain it is available with the results.

3.4 Software Tests

The cls-scala, shapeless-feat, and templating projects come with tests that achieve above 90% code coverage. The cls-scala-presentation-play-git project has above 65% test coverage. A continuous integration pipeline triggers builds and tests on every commit to the project code-base. Commits that result in decreasing test coverage are by default rejected. This enforces increasing test coverage over time. Tests are implemented using a mixture of unit, integration, and regression testing embedded into the ScalaTest framework [197]. Some tests in shapeless-feat use ScalaCheck [144] for randomized property based testing, which pushes its test coverage above 96%. The comparatively low coverage in cls-scala-presentation-play-git is due to the side-effects associated with running a webserver and git to host solutions. In contrast to the other projects this makes cls-scala-presentation-play-git stateful, which is inherently harder to test than the otherwise purely functional implementations. The practice to turn all reported bugs into integration tests has proven to be very useful, both for debugging and future developments, which sometimes repeated old mistakes. Future implementation plans include further convergence of the Scala implementation with its Coq counterpart. At some point this may lead to a situation, where the lemmas in this text can be used for property based testing. All tests and coverage reports are publicly available with the source code [27].

The extracted Coq code for `Labyrinth.v` has been tested manually and found to produce the desired results.

Ongoing independent third-party use of cls-scala has been useful for finding bugs in the past. Usage scenarios, which have been implemented are discussed in the next section.

Chapter 4

Applications and Impact

The work presented in this text has been used by other researchers, who were able to develop multiple successful applications. Some of these will be discussed in this chapter. Applications are grouped into three categories with one section per category. These are: research which has been conducted in tight collaboration with the author, work done in student projects, some of which have been advised by the author, and research that has been conducted independently from the author. Each of the categories adds confidence to the validity and usefulness of the overall approach.

4.1 Work done in collaboration with the Author

An ongoing research effort with Boris Düdder, George Heineman, and Jakob Rehof investigates applications of (CL)S to software product lines [95; 21; 96; 26]. A large scale application of this work to a software product line of solitaire games is available online [98]. At the time of writing, the product line has 17 families of solitaire games, where each family has up to 9 sub-variations. The whole product line is generated from just under 13500 lines of code including test cases. The individual games are implemented in Java and some have an additionally synthesized Python implementation. In comparison, the older F# version of (CL)S required approximately 4500 lines of code to synthesize 4 variations that were only available as Java implementations. The results show two things about cls-scala. On the one hand, the framework is scalable to larger code-bases than most related synthesis approaches (e.g. the largest example in [160] is a single red-black-tree balancing function). This effect is mainly due to the larger building blocks, where each combinator can generate arbitrary amounts of code instead of just a single component of a functional expression. On the other hand, cls-scala has improved over previous implementations by allowing a massive scale-down in the amount of specification required to generate code. This has two reasons. The first reason is the possibility to share code by implementing combinators in regular Scala classes with inheritance and programmatic

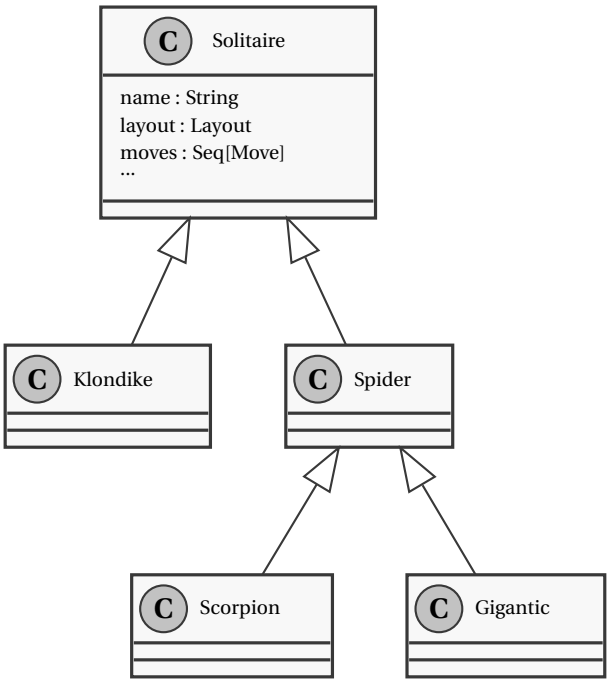

Figure 4.1: Simplified Solitaire Domain Model

access to them, instead of just relying on static text file fragments written in a domain specific Lambda Calculus like language. The second reason is reuse and structuring from domain modeling. An abstract domain model M is a set of classes to abstractly describe the application domain without containing information about how it is implemented. Figure 4.1 shows a highly simplified version of the solitaire domain. Every instance of the base-class Solitaire has a name, a layout to specify how cards are placed, and a sequence of possible moves. Klondike has no sub variations, while the family of Spider solitaire games has the subvariations Scorpion and Gigantic. For simplicity, subvariations are modeled by inheritance, but more flexible models using delegation are also possible. When presented with an instance of the Solitaire class, the implementation will create a repository with combinators specifically selected and instantiated for that instance. This shrinks the search space, which now does not include combinators for arbitrary other variations. It also motivates designing combinators that are parameterized over instance properties, so they can work for arbitrary instead of specific instances. Families can share model components by instantiating them as soon as they become available in the model class-hierarchy (e.g. Spider solitaire games might all fix a certain card layout). Algebraically, if M is a set specifying all allowed domain models, the implementation provides a family of algebras $(\mathscr{A}^{\Sigma(m)})_{m \in M}$, where each model instance $m \in M$ gives rise to a signature $\Sigma(m)$ and its implementation in form of an algebra $\mathscr{A}^{\Sigma(m)}$. Exactly the same happened for the machine trace algebra in Lemma 34, where the model are 2-Counter automaton specifications. In earlier implementations of (CL)S, it was impossible to determine why a specific combinator was included in the repository. This question can now be answered by finding the model component for which the combinator was instantiated.

Another line of work done is developed in collaboration with Anna Vasileva, and Moritz Roidl. It aims to investigate how to help experts in non-synthesis related domains with using and understanding cls-scala. A prototype implementation of the labyrinth example from [18] is provided together with an MQTT [107] based interface to externally control inhabitation. This interface is accessed by the Unity game engine [192], which then controls a laser based visualization installation to optically illustrate solutions. The code and a demonstration video are available online [29]. The use-case illustrates that first steps have been taken to make the framework usable, even by non experts in computer science.

4.2 Work done by Students

Various applications of cls-scala have been successfully explored by students in bachelor's and master's thesis projects. In [173] Docker container [54] configurations are synthesized based on a state machine model. Configuration data collected from a user interface is used by the inhabitation algorithm to generate transition inputs for a state machine, which then generates code in form of docker configurations and shell scripts. These are used to setup a distributed system with replicated databases and variable topologies for redundancy. Results have been further refined in [204], where additional constraint-solver scripts are synthesized for computing job-to-node-assignments within a cluster. In [155] the idea of directly integrating computations into the search process of synthesis, which is illustrated by the 2-Counter machine trace Algebra in Lemma 34, is implemented for various examples in cls-scala. These include deterministic and non-deterministic finite automata and the word problem Tree Grammars. Multiple better implementation strategies for the robot control programs generated in [23] are realized with cls-scala in [170]. One of them controls robot behaviors with arrows, which most recently has spawned some research interest [76]. In particular [170] also illustrates how abstract types in Scala can facilitate reusing the same signature structure for multiple independent implementation strategies. A tool for the design of flexible electronic business processes is developed on top of cls-scala in [48]. While traditional business process modeling needs pre-designed branches for every possible eventuality, the idea in [48] is to synthesize a custom-tailored process per use-case. This process is smaller and thereby easier to understand and maintain. Block diagrams for numeric processing are synthesized in [183]. Language-agnostic synthesis is not restricted to computer executed code, which is illustrated in [187], where variations of net-plan structures are generated for factory production steps carried out by humans. The metaprogramming support of (CL)S is soon to be enhanced by two successful projects which implement parsers, pretty printers, and AST manipulation tools for Coq [205] and Python [142].

At the time of writing, a project group of 13 master students has started to evaluate possibilities for synthesizing cyber physical systems. Their goal is to create different configurations of intelligent plant management systems and their project is limited to one year. One preliminary success is tool integration of ASTs for 3D-printable OpenSCAD [114] models.

Beyond applicability of cls-scala to specific problems, two observations can be made: Most of the students involved in the projects mentioned above did not have prior formal training in type theory. This illustrates, that the framework is usable without expert knowledge. The second observation concerns the scope of applicability. It includes abstract machine models, configuration files and shell scripts, 3D printer models, and human executed factory production plans. This is strong evidence, that the main goal of this work, language-agnostic synthesis, has been achieved.

4.3 Work done by other Researchers

There exists externally peer-reviewed evidence for successful applications of the work provided in this text. These applications have been implemented, investigated and reviewed independently of any involvement of the author.

An earlier version of the Coq formalization for the BCD subtyping algorithm is used by Honsell et al. [103] to investigate different variations of intersection type systems in the Δ-Framework. The authors were able to add union types to the subtype relation and still obtain a working verified decision procedure.

In [172] business processes to model clinical path-ways are synthesized using cls-scala. The authors manage to develop a structured methodology for the flexible generation of patient individualized healthcare workflows. Their results are evaluated using a case-study, which comes to a positive conclusion on applicability of the framework. Domain experts on clinical business processes and logistics have been involved in the investigation.

Another independent investigation with involvement of domain experts shows, that the cls-scala framework can be used to synthesize plans for building factories [201]. A case-study based on a real-world scenario is used to show how generated plans can help to identify bottle-necks, which would cause the build process to stall. The authors conclude that manual planning labor can be automated, which leads to a better informed planning process.

The above findings indicate theoretical usefulness of the formalized proofs, and practical usefulness of the implementation. Again, language-agnosticism is crucial to allow for the vastly different domains of application.

Chapter 5

Conclusions and Future Work

The journey took a long formal proof, but in the end this text has shown that synthesis can be understood in a general language-agnostic way by going back to the first principles of Combinatory Logic and abstract algebra. The 100 year old statement 6.002 of Wittgenstein's Tractatus [202] proved to be adequate for software synthesis: all programming language statements can be constructed by applying (algebraic) operations to other statements. The author hopes that this insight, together with the perspective it can give on the discussion of related work in Section 1.1, helps to bring some clarity into the sometimes confusing state of software synthesis.

Along the way to a fully formalized mechanically checked proof, many results about intersection types, Combinatory Logic and abstract algebra have been formalized. The outline of novelties is given in Section 1.2 and maybe some of these results can be used beyond this text. At least the analysis in Chapter 4 seems to indicate that there is some hope, that the theoretical results of Chapter 2 and the practical results of Chapter 3 have brought some benefit other the author's pleasure to develop them.

All that remains to discuss at this point are some of the boundaries of knowledge, which are salient points of future work.

It is natural to ask, if there is a lower bound on the complexity of the sort emptiness problem for signature families. The answer to this question has not been mechanically formalized yet. It is therefore not included in the main results of this text. The following proof sketch should be seen as a conjecture to be elaborated in future work:

Given a number n and n Tree Grammars $G_n = (S_n, N_n, \mathscr{F}, R_n)$, the problem of deciding $\bigcap_{1 \leq i \leq n} L(G_i) \neq \emptyset$ is EXPTIME-complete [42; 196] and sort-emptiness can be reduced to it. Let $f^k \in \mathscr{F}$ be a non-terminal consistently used with k (including k = 0) arguments, then a signature family can be specified with $\Sigma_{i \in I} = (\mathbb{S}, \mathbb{O}, \text{arity}_i, \text{dom}_i, \text{range}_i)$, where

- I is a set of binary numbers s.t.
  $i \in I$ iff $\text{decode}(i) \in (N_1 \times N_2 \times \cdots \times N_n)^{1+\max\{k \mid f^k \in \mathscr{F}\}}$
- $\mathbb{S} = I \uplus \{\bullet, \circ\}$ ordered by $\leq_{\mathbb{S}} = \{(s,s) \mid s \in \mathbb{S}\}$
- $\mathbb{O} = \mathscr{F}$
- $\text{arity}_i(f^k) = k$
- $\text{ok}(\text{decode}(i), f^k)$ iff for all $1 \leq m \leq n$ there exists

$$\begin{aligned}&\pi_m(\pi_{k+1}(\text{decode}(i))) \mapsto \\ &\quad f^k(\pi_m(\pi_1(\text{decode}(i))), \pi_m(\pi_2(\text{decode}(i))), \ldots, \pi_m(\pi_k(\text{decode}(i)))) \in R_m\end{aligned}$$

- $\text{dom}_i(f^k) = \begin{cases} (\pi_1(\text{decode}(i)), \pi_2(\text{decode}(i)), \ldots, \pi_k(\text{decode}(i))) & \text{for ok}(\text{decode}(i), f^k) \\ \underbrace{(\bullet, \bullet, \ldots, \bullet)}_{k-\text{times}} & \text{otherwise} \end{cases}$
- $\text{range}_i(f^k) = \begin{cases} \pi_{k+1}(\text{decode}(i)) & \text{for ok}(\text{decode}(i), f^k) \\ \circ & \text{otherwise} \end{cases}$

The idea is to have operations for each terminal f^k matching the product of the grammar rules for that operation. Non-determinism of rules is encoded into the choice of $i \in I$. Invalid constellations are filtered by setting inputs to $\bullet$, which cannot be produced by any operation. Family Σ is polynomial in the input size, because the cardinality of the set $(N_1 \times N_2 \times \cdots \times N_n)^{1+max\{k \mid f^k \in \mathscr{F}\}}$ is less or equal to $n^{\max\{|N_j| \mid 1 \leq m \leq n\} \cdot (1+\max\{k \mid f^k \in \mathscr{F}\})}$ and I therefore can be specified by stating the amount of bits required for its elements, which is a number less than $\dfrac{\log_2(\max\{|N_j| \mid 1 \leq m \leq n\} \cdot (1 + \max\{k \mid f^k \in \mathscr{F}\}))}{\log_2(n)}$. Computing decode and ok is also possible in polynomial time because both are simple lookup functions. Now the algebra $\mathscr{A}^{\Sigma}_{\cap}$

- with carrier

$$\mathbb{C}_s = \begin{cases} \{t \mid t \in L(G_1, A_1) \cap L(G_2, A_2) \cap \cdots \cap L(G_n, A_3)\} \\ \qquad \text{for } s \in I \text{ and } \pi_1(\text{decode}(i)) = (A_1, A_2, \ldots, A_n) \\ \mathbb{1} \qquad \text{for } s = \circ \\ \emptyset \qquad \text{for } s = \bullet \end{cases}$$

- and action

$$h_s(i, f^k, ts) = \begin{cases} f^k(ts) & \text{for ok}(\text{decode}(i), f^k) \\ () & \text{otherwise} \end{cases}$$

generates $\mathbf{AlgGen}_s^{\mathscr{A}_\cap^\Sigma} = \bigcap_{1\leq i\leq n} L(G_i)$ for sort s with $\mathrm{decode}(s) = (S_1, S_2, \ldots, S_n)$. Deciding emptiness of $\mathbf{AlgGen}_s^{\mathscr{A}_\cap^\Sigma}$ therefore solves the intersection problem of Tree Grammars and sort-emptiness is EXPTIME-hard.

From a practical standpoint, the conjecture above raises the question of when scalability limits of the approach will be reached. When they are reached at some point, a possible idea is to try to overcome the compositionaly limitations of machine learning discussed in Section 1.1. Insights from the CoqHammer project [44] show, that a major problem for constructing proofs (or programs) out of large knowledge bases is preimse selection. In Combinatory Logic, this could be addressed by giving more structure to the context Γ. It would be interesting to investigate adding a rule such as

$$\frac{\mathrm{CanReach}(\Gamma_1, \Gamma_2) \qquad \mathrm{PlausibleFor}(\Gamma_2, A) \qquad \Gamma_2 \vdash M : A}{\Gamma_1 \vdash M : A}$$

where predicate $\mathrm{CanReach}(\Gamma_1, \Gamma_2)$ is a user-defined or machine-learned relation on contexts, and $\mathrm{PlausibleFor}(\Gamma_2, A)$ is a machine-learned predicate on contexts and types, that limits the search space.

A current drawback of the completely language-agnostic approach is that sometimes irrelevant terms can be generated. An example is $\mathrm{id}(\ldots\mathrm{id}(\mathrm{id}(x))\ldots)$ where x is the only interesting inhabitant. Some of the equational methods used in SyGus might be applicable for filtering these irrelevant results. Most authors in literature equip algebraic signatures with equations and so it seems obvious that the work presented here should be extended in this direction.

Finally many more practical applications of language-agnostic synthesis remain to be found. The promising successes regarding synthesis problems in software product lines and the initial steps toward more usability, even by non-experts, should be a start, rather than an end of a journey.

Appendix A

Appendix

Section/Statement		File	Name in Coq
1	2.3/Lemma 1	Types.v	`Semantics_functional`
2	2.3/Lemma 2	Types.v	`subtype_total`
3	2.3/Lemma 3.1	Types.v	`bcd_cat_bigcap_f`
4	2.3/Lemma 3.2	Types.v	`bigcap_omega`
5	2.3/Lemma 3.3	Types.v	`bigcap_cast__ctor`
6	2.3/Lemma 3.4	Types.v	`bigcap_cast__Arr`
7	2.3/Lemma 3.5	Types.v	`bigcap_cast__Prod`
8	2.3/Lemma 3.6	Types.v	`bcd__Arr`
9	2.3/Lemma 3.7	Types.v	`bcd__ProdDist`
10	2.3/Lemma 3.8	Cover.v	`bcd_subset_f`
11	2.3/Lemma 4	Types.v	`subty__sound`
12	2.3/Lemma 5.1	Types.v	`subty__omega`
13	2.3/Lemma 5.2	Types.v	`check__tgt__subseq`
14	2.3/Lemma 5.3	Types.v	`Omega__tgts`
15	2.3/Lemma 5.4	Types.v	`Omega__subty`
16	2.3/Lemma 5.5	Types.v	`weaken_check_tgt`
17	2.3/Lemma 5.6	Types.v	`subty__weaken`
18	2.3/Lemma 5.7	Types.v	`subty__cat`
19	2.3/Lemma 5.8	Types.v	`subty__CtorTrans`
20	2.3/Lemma 5.9	Types.v	`omegaDoneTgt`
21	2.3/Lemma 5.10	Types.v	`subty__Refl`
22	2.3/Lemma 5.11	Types.v	`split_tgts_for_srcs_gte`
23	2.3/Lemma 5.12	FCL.v	`tgt_for_srcs_gte_cat`
24	2.3/Lemma 5.13	Types.v	`subty__left`, `subty__right`
25	2.3/Lemma 5.14	Types.v	`can_cast_trans__Ctor`, `can_cast_trans__Prod`

Section/Statement		File	Name in Coq
26	2.3/Lemma 5.15	Types.v	subty__Trans
27	2.3/Lemma 5.16	Types.v	subty__CtorDist
28	2.3/Lemma 5.17	Types.v	subty__Idem
29	2.3/Lemma 6	Types.v	subty_complete
30	2.3/Theorem 1	Types.v	subtype_machine_correct, subtypeMachineP
31	2.3/Lemma 7	Types.v	Domain_size
32	2.3/Lemma 8	Types.v	primeComponentPrime
33	2.3/Lemma 9.1	Types.v	addAndFilterLeq__A
34	2.3/Lemma 9.2	Types.v	addAndFilterLeq__DeltaA
35	2.3/Lemma 9.3	Types.v	addAndFilter_has_le_weaken
36	2.3/Lemma 9.4	Types.v	addAndFilter_in
37	2.3/Lemma 9.5	Types.v	bigcap_has_le
38	2.3/Lemma 9.6	Types.v	addAndFilter_monotonic
39	2.3/Lemma 9.7	FCL.v	all_addAndFilter
40	2.3/Lemma 9.8	Types.v	primeFactors_rec_prime
41	2.3/Lemma 9.9	Types.v	primeFactors_monotonic
42	2.3/Lemma 9.10	Types.v	primeFactors_rec_leq
43	2.3/Lemma 9.11	Types.v	addAndFilterGeq__Delta
44	2.3/Lemma 9.12	Types.v	bcd_all_ge
45	2.3/Lemma 9.13	Types.v	addAndFilterGeq
46	2.3/Lemma 9.14	Types.v	primeFactors_rec_geq
47	2.3/Lemma 9.15	Types.v	addAndFilter_nosubdup
48	2.3/Lemma 9.16	Types.v	primeFactors_nosubdup
49	2.3/Lemma 9.17	Types.v	primeFactors__notOmega
50	2.3/Lemma 9.18	Types.v	desubdup_nosubdupb
51	2.3/Lemma 9.19	Types.v	desubdup_all
52	2.3/Lemma 9.20	Types.v	desubdup_leq
53	2.3/Lemma 9.21	Types.v	desubdup_geq
54	2.3/Lemma 9.22	Types.v	addAndFilter_size
55	2.3/Lemma 9.23	Types.v	desubdup_size
56	2.3/Lemma 9.24	Types.v	desubdup_notOmega
57	2.3/Lemma 9.25	Types.v	bcd_prime_ge_all
58	2.3/Lemma 9.26	Types.v	prime_filter_le
59	2.3/Lemma 9.27	Types.v	nosubdup_unique
60	2.3/Lemma 9.28	Types.v	nosubdup_prime_injective
61	2.3/Lemma 9.29	Types.v	nosubdup_prime_bijective
62	2.3/Lemma 9.30	Types.v	nosubdup_weaken
63	2.3/Lemma 9.31	Types.v	bcd_prime_strengthen
64	2.3/Lemma 9.32	Types.v	nosubdup_everywhere
65	2.3/Lemma 9.33	Types.v	nosubdup_prime_perm

Section/Statement		File	Name in Coq
66	2.3/Lemma 9.34	Types.v	PermUpToSubtyping_size
67	2.3/Theorem 2.1	Types.v	primeFactors_leq
68	2.3/Theorem 2.2	Types.v	primeFactors_geq
69	2.3/Theorem 2.3	Types.v	primeFactors_prime
70	2.3/Theorem 2.4	Types.v	primeFactors_minimal
71	2.4/Lemma 10.1	FCL.v	FCL_ind
			(automatically generated by Coq)
72	2.4/Lemma 10.2	FCL.v	FCL_Var_le
73	2.4/Lemma 10.3	FCL.v	FCL_MP_inv
74	2.4/Lemma 10.4	FCL.v	FCL_normalized_ind
75	2.4/Lemma 11	FCL.v	FCL_II, FCL_Omega
76	2.4/Lemma 12	FCL.v	FCL_weaken
77	2.4/Lemma 13.1	Cover.v	mkArrow_arrow
78	2.4/Lemma 13.2	Cover.v	mkArrow_rcons
79	2.4/Lemma 13.3	Cover.v	mkArrow_arity
80	2.4/Lemma 13.4	Cover.v	mkArrow_tgt_le
81	2.4/Lemma 13.5	Cover.v	omega_mkArrow_tgt
82	2.4/Lemma 13.6	Cover.v	mkArrow_prime
83	2.4/Lemma 13.7	Cover.v	mkArrow_dist
84	2.4/Lemma 13.8	Cover.v	mkArrow_srcs_ge
85	2.4/Lemma 14.1	Algebra.v	revApply_rcons
86	2.4/Lemma 14.2	Algebra.v	revApply_nil
87	2.4/Lemma 14.3	FCL.v	revApply_unapply
88	2.4/Lemma 14.4	FCL.v	unapply_revApply
89	2.4/Lemma 15.1	FCL.v	FCL__App
90	2.4/Lemma 15.2	FCL.v	FCL__invApp
91	2.4/Lemma 16.1	FCL.v	minimalArrowType_le
92	2.4/Lemma 16.2	FCL.v	minimalType_sound
93	2.4/Lemma 16.3	FCL.v	minimalArrowType_minimal
94	2.4/Lemma 16.4	FCL.v	minimalType_minimal
95	2.4/Theorem 3	FCL.v	fclP
96	2.4.2/Lemma 17.1	FCL.v	inPartition1_bigcap
97	2.4.2/Lemma 17.2	FCL.v	inPartition2_bigcap
98	2.4.2/Lemma 17.3	FCL.v	inPartition1_minimalType
99	2.4.2/Lemma 17.4	FCL.v	inPartition2_minimalType
100	2.4.2/Lemma 17.5	FCL.v	minimalType_partitioned
101	2.4.2/Theorem 4	FCL.v	FCL__split
102	2.4.2/Lemma 18.1	FCL.v	bigcap_hom
103	2.4.2/Lemma 18.2	FCL.v	hom_arrow_cast
104	2.4.2/Lemma 18.3	FCL.v	minimalType_hom
105	2.4.2/Lemma 18.4	FCL.v	FCL__hom

	Section/Statement	File	Name in Coq
106	2.4.2/Lemma 19.1	FCL.v	lift_arrow_hom
107	2.4.2/Lemma 19.2	FCL.v	lift_inter_hom
108	2.4.2/Lemma 19.3	FCL.v	lift_arrow_preimage
109	2.4.2/Lemma 19.4	FCL.v	lift_omega_hom
110	2.4.2/Lemma 19.5	FCL.v	unlift_lift
111	2.4.2/Lemma 19.6	FCL.v	lift_map_bigcap
112	2.4.2/Lemma 19.7	FCL.v	lift_cast_ctor
113	2.4.2/Lemma 19.8	FCL.v	isOmega_lift
114	2.4.2/Lemma 19.9	FCL.v	lift_cast_arr
115	2.4.2/Lemma 19.10	FCL.v	lift_cast_prod
116	2.4.2/Lemma 19.11	FCL.v	lift_cast_inter_prod
117	2.4.2/Lemma 19.12	FCL.v	lift_subtype_hom
118	2.4.2/Lemma 19.13	FCL.v	inPartition_lift1, inPartition_lift2
119	2.4.2/Lemma 19.14	FCL.v	dist_arr_inPartition
120	2.4.2/Lemma 19.15	FCL.v	dist_inter_inPartition
121	2.4.2/Lemma 19.16	FCL.v	omega_inPartition
122	2.4.2/Lemma 19.17	FCL.v	inPartition_cast_ctor
123	2.4.2/Lemma 19.18	FCL.v	st_omega_inPartition
124	2.4.2/Lemma 19.19	FCL.v	primeComponents_inPartition
125	2.4.2/Lemma 19.20	FCL.v	st_irrel_check_inPartition
126	2.4.2/Theorem 5	FCL.v	canonicalCoproductLifted
127	2.5.1/Lemma 20.1	Cover.v	coverMachineFunctional_step
128	2.5.1/Lemma 20.2	Cover.v	coverMachineFunctional
129	2.5.1/Lemma 21.1	Cover.v	stepSize
130	2.5.1/Lemma 21.2	Cover.v	maxSteps
131	2.5.1/Lemma 21.3	Cover.v	Domain_total
132	2.5.1/Lemma 22.1	Cover.v	step_programStack
133	2.5.1/Lemma 22.2	Cover.v	step_stateMonotonic
134	2.5.1/Lemma 22.3	Cover.v	steps_stateMonotonic
135	2.5.1/Lemma 23.1	Cover.v	filterMergeMultiArrows_cat
136	2.5.1/Lemma 23.2	Cover.v	filterMergeMultiArrows_subseq
137	2.5.1/Lemma 23.3	Cover.v	filterMergeMultiArrows_map_cons
138	2.5.1/Lemma 23.4	Cover.v	step_mergeComponents
139	2.5.1/Lemma 23.5	Cover.v	step_sound
140	2.5.1/Lemma 23.6	Cover.v	splitsTail
141	2.5.1/Lemma 23.7	Cover.v	step_mergeComponents_in
142	2.5.1/Lemma 23.8	Cover.v	sound_reverse
143	2.5.1/Lemma 23.9	Cover.v	semantics_mergeComponents
144	2.5.1/Lemma 24.1	Cover.v	partitionCover_subset
145	2.5.1/Lemma 24.2	Cover.v	partitionCover_notSubset

Section/Statement		File	Name in Coq
146	2.5.1/Lemma 24.3	Cover.v	partitionCover_subseq1
147	2.5.1/Lemma 24.4	Cover.v	partitionCover_subseq2
148	2.5.1/Lemma 24.5	Cover.v	instructions_covered_step
149	2.5.1/Lemma 24.6	Cover.v	not_omega_instruction_step
150	2.5.1/Lemma 24.7	Cover.v	arity_equal_step
151	2.5.1/Lemma 24.8	Cover.v	mergeMultiArrows_arity
152	2.5.1/Lemma 24.9	Cover.v	mergeMultiArrow_tgt_le
153	2.5.1/Lemma 24.10	Cover.v	mergeMultiArrow_tgt_ge
154	2.5.1/Lemma 24.11	Cover.v	mergeMultiArrow_srcs_le
155	2.5.1/Lemma 24.12	Cover.v	mergeMultiArrow_srcs_ge
156	2.5.1/Lemma 24.13	Cover.v	mergeMultiArrows_tgt_le
157	2.5.1/Lemma 24.14	Cover.v	mergeMultiArrows_tgt_ge
158	2.5.1/Lemma 24.15	Cover.v	mergeMultiArrows_srcs_le
159	2.5.1/Lemma 24.16	Cover.v	mergeMultiArrows_srcs_ge
160	2.5.1/Lemma 24.17	Cover.v	toCover_prime_step
161	2.5.1/Lemma 24.18	Cover.v	partitionCover_prime
162	2.5.1/Lemma 24.19	Cover.v	filterMergedArrows_in_cons
163	2.5.1/Lemma 24.20	Cover.v	covered_head_tgt
164	2.5.1/Lemma 24.21	Cover.v	partitionCover_drop1
165	2.5.1/Lemma 24.22	Cover.v	partitionCover_drop2
166	2.5.1/Lemma 24.23	Cover.v	mergeMultiArrows_cons_arity
167	2.5.1/Lemma 24.24	Cover.v	partitionCover_complete
168	2.5.1/Lemma 24.25	Cover.v	partition_cover_both
169	2.5.1/Lemma 24.26	Cover.v	complete_partitionCover
170	2.5.1/Lemma 24.27	Cover.v	currentResultNotDone_step
171	2.5.1/Lemma 24.28	Cover.v	notDone_incomplete
172	2.5.1/Lemma 24.29	Cover.v	filterMergeMultiArrows_map_cons2
173	2.5.1/Lemma 24.30	Cover.v	mergeMultiArrow_srcs_monotonic
174	2.5.1/Lemma 24.31	Cover.v	cap_dcap
175	2.5.1/Lemma 24.32	Cover.v	dcap_cap
176	2.5.1/Lemma 24.33	Cover.v	mergeMultiArrow_srcs_map_zip
177	2.5.1/Lemma 24.34	Cover.v	impossible_notSubtype
178	2.5.1/Lemma 24.35	Cover.v	complete_reverse
179	2.5.1/Lemma 24.36	Cover.v	steps_complete
180	2.5.1/Lemma 25.1	Cover.v	step_tgt_sound
181	2.5.1/Lemma 25.2	Cover.v	step_tgt_sound_reverse
182	2.5.1/Lemma 25.3	Cover.v	steps_tgt_sound
183	2.5.1/Lemma 26.1	Cover.v	arity_increasing_cat
184	2.5.1/Lemma 26.2	Cover.v	splitRec_arity
185	2.5.1/Lemma 26.3	Cover.v	splitTy_arity
186	2.5.1/Lemma 26.4	Cover.v	arity_increasing_arity_equal

Section/Statement		File	Name in Coq
187	2.5.1/Lemma 26.5	Cover.v	splitRec_monotonic
188	2.5.1/Lemma 26.6	Cover.v	splitRec_context_size_eq
189	2.5.1/Lemma 26.7	Cover.v	splitRec_context_monotonic
190	2.5.1/Lemma 26.8	Cover.v	splitRec_split_context
191	2.5.1/Lemma 26.9	Cover.v	splitRec_sound
192	2.5.1/Lemma 26.10	Cover.v	splitTy_sound
193	2.5.1/Lemma 26.11	Cover.v	merge_assoc
194	2.5.1/Lemma 26.12	Cover.v	merges0
195	2.5.1/Lemma 26.13	Cover.v	merge0s
196	2.5.1/Lemma 26.14	Cover.v	splitRec_merge
197	2.5.1/Lemma 26.15	Cover.v	map_merge
198	2.5.1/Lemma 26.16	Cover.v	splitRec_rcat
199	2.5.1/Lemma 26.17	Cover.v	splitTy_slow_splitTy
200	2.5.1/Lemma 26.18	Cover.v	splitTy_slow_omega
201	2.5.1/Lemma 26.19	Cover.v	nth_merge
202	2.5.1/Lemma 26.20	Cover.v	splitTy_slow_inter_subseq2
203	2.5.1/Lemma 26.21	Cover.v	splitTy_complete_ctor
204	2.5.1/Lemma 26.22	Cover.v	splitTy_complete_prod
205	2.5.1/Lemma 26.23	Cover.v	splitTy_complete_omega
206	2.5.1/Lemma 26.24	Cover.v	splitTy_complete_alternative
207	2.5.1/Lemma 26.25	Cover.v	splitTy_complete
208	2.5.1/Lemma 27.1	Cover.v	splitTy_instructionsCovered
209	2.5.1/Lemma 27.2	Cover.v	coverMachine_splitTy_complete
210	2.5.1/Lemma 27.3	Cover.v	bcd_multiArrow_Dist
211	2.5.1/Lemma 27.4	Cover.v	coverMachine_splitTy_sound
212	2.5.1/Lemma 27.5	Cover.v	coverMachine_splitTy_tgt_sound
213	2.5.1/Lemma 28.1	Cover.v	reduction_subseq
214	2.5.1/Lemma 28.2	Cover.v	soundnessPreserving
215	2.5.1/Lemma 28.3	Cover.v	tgt_soundnessPreserving
216	2.5.1/Lemma 28.4	Cover.v	completenessPreserving
217	2.5.2/Lemma 29.1	FCL.v	FCL_sound_sound
218	2.5.2/Lemma 29.2	FCL.v	dropTargets_suffix
219	2.5.2/Lemma 29.3	FCL.v	suffix_word
220	2.5.2/Lemma 29.4	FCL.v	suffix_sound
221	2.5.2/Lemma 29.5	FCL.v	cat_sound
222	2.5.2/Lemma 29.6	FCL.v	commitMultiArrow_sound
223	2.5.2/Lemma 29.7	FCL.v	commitUpdates_sound
224	2.5.2/Lemma 29.8	FCL.v	accumulateCovers_sound
225	2.5.2/Lemma 29.9	FCL.v	foldl_accumulateCovers_sound
226	2.5.2/Lemma 29.10	FCL.v	inhabit_cover_sound
227	2.5.2/Lemma 29.11	FCL.v	OmegaRules_sound

Section/Statement		File	Name in Coq
228	2.5.2/Lemma 29.12	FCL.v	inhabitation_step_sound
229	2.5.2/Lemma 29.13	FCL.v	inhabit_multistep_sound
230	2.5.2/Lemma 30	FCL.v	InhabitationSemantics_functional_step
231	2.5.2/Lemma 31.1	FCL.v	grammarTypes_src_mem
232	2.5.2/Lemma 31.2	FCL.v	grammarTypes_tgt_mem
233	2.5.2/Lemma 31.3	FCL.v	commitMultiArrow_parameterTypes_subset
234	2.5.2/Lemma 31.4	FCL.v	commitUpdates_parameterTypes_subset
235	2.5.2/Lemma 31.5	FCL.v	accumulateCovers_parameterTypes_subset
236	2.5.2/Lemma 31.6	FCL.v	maxParameterTypes_initialTarget
237	2.5.2/Lemma 31.7	FCL.v	foldl_accumulateCovers_ parameterTypes_subset
238	2.5.2/Lemma 31.8	FCL.v	inhabit_cover_parameterTypes_subset
239	2.5.2/Lemma 31.9	FCL.v	OmegaRules_params
240	2.5.2/Lemma 31.10	FCL.v	OmegaRules_subset
241	2.5.2/Lemma 31.11	FCL.v	inhabitation_step_subset
242	2.5.2/Lemma 31.12	FCL.v	inhabit_step_rel_wf
243	2.5.2/Lemma 31.13	FCL.v	inhabit_step_rel_cons
244	2.5.2/Lemma 31.14	FCL.v	dropTargets_size
245	2.5.2/Lemma 31.15	FCL.v	commitUpdates_nil_eq
246	2.5.2/Lemma 31.16	FCL.v	reduceMultiArrows_nil
247	2.5.2/Lemma 31.17	FCL.v	accumulateCovers_failed_targets_eq
248	2.5.2/Lemma 31.18	FCL.v	accumulateCovers_failed_rev
249	2.5.2/Lemma 31.19	FCL.v	foldl_accumulateCovers_failed_rev
250	2.5.2/Lemma 31.20	FCL.v	foldl_accumulateCovers_ failed_targets_eq
251	2.5.2/Lemma 31.21	FCL.v	inhabit_cover_failed_targets_eq
252	2.5.2/Lemma 31.22	FCL.v	computeFailExisting_notFound
253	2.5.2/Lemma 31.23	FCL.v	inhabitation_step_sizes
254	2.5.2/Lemma 31.24	FCL.v	domain_start
255	2.5.2/Lemma 32.1	FCL.v	FailSound_cat
256	2.5.2/Lemma 32.2	FCL.v	cat_FailSound
257	2.5.2/Lemma 32.3	FCL.v	computeFailExisting_FailSound
258	2.5.2/Lemma 32.4	FCL.v	accumulateCovers_FailSound
259	2.5.2/Lemma 32.5	FCL.v	inhabit_cover_FailSound
260	2.5.2/Lemma 32.6	FCL.v	OmegaRules_FailSound
261	2.5.2/Lemma 32.7	FCL.v	inhabit_step_FailSound
262	2.5.2/Lemma 32.8	FCL.v	noTargetFailures_suffix
263	2.5.2/Lemma 32.9	FCL.v	inhabit_cover_noTargetFailures
264	2.5.2/Lemma 32.10	FCL.v	inhabitation_step_noTargetFailures
265	2.5.2/Lemma 33.1	FCL.v	cancel_group_flatten

Section/Statement		File	Name in Coq
266	2.5.2/Lemma 33.2	FCL.v	updateGroups0
267	2.5.2/Lemma 33.3	FCL.v	group_notComb
268	2.5.2/Lemma 33.4	FCL.v	group_comb
269	2.5.2/Lemma 33.5	FCL.v	dropTargets_notCombinator
270	2.5.2/Lemma 33.6	FCL.v	dropTargets_combinatorOrEmpty
271	2.5.2/Lemma 33.7	FCL.v	group_split
272	2.5.2/Lemma 33.8	FCL.v	FCL_complete_emptyTargets
273	2.5.2/Lemma 33.9	FCL.v	future_word_word
274	2.5.2/Lemma 33.10	FCL.v	future_word_weaken
275	2.5.2/Lemma 33.11	FCL.v	rule_absorbl
276	2.5.2/Lemma 33.12	FCL.v	future_word_dropFailed
277	2.5.2/Lemma 33.13	FCL.v	computeFailExisting_failed_complete
278	2.5.2/Lemma 33.14	FCL.v	rule_absorbl_apply
279	2.5.2/Lemma 33.15	FCL.v	computeFailExisting_existing
280	2.5.2/Lemma 33.16	FCL.v	OmegaRules_future_word
281	2.5.2/Lemma 33.17	FCL.v	rule_MP
282	2.5.2/Lemma 33.18	FCL.v	FCL_Omega_complete
283	2.5.2/Lemma 33.19	FCL.v	accumulateCovers_cat
284	2.5.2/Lemma 33.20	FCL.v	inhabit_cover_flatten
285	2.5.2/Lemma 33.21	FCL.v	inhabit_cover_empty
286	2.5.2/Lemma 33.22	FCL.v	commitMultiArrow_cons
287	2.5.2/Lemma 33.23	FCL.v	commitUpdates_flatten
288	2.5.2/Lemma 33.24	FCL.v	commitMultiArrow_size
289	2.5.2/Lemma 33.25	FCL.v	commitMultiArrow_nth
290	2.5.2/Lemma 33.26	FCL.v	commitMultiArrows_combinatorOrEmpty
291	2.5.2/Lemma 33.27	FCL.v	nextTargets_combinatorOrEmpty
292	2.5.2/Lemma 33.28	FCL.v	group_commitMultiArrow
293	2.5.2/Lemma 33.29	FCL.v	future_word_weaken_inhabapply
294	2.5.2/Lemma 33.30	FCL.v	commitMultiArrow_parameters
295	2.5.2/Lemma 33.31	FCL.v	future_word_weaken_targets1
296	2.5.2/Lemma 33.32	FCL.v	future_word_weaken_targets2
297	2.5.2/Lemma 33.33	FCL.v	future_word_covers
298	2.5.2/Lemma 33.34	FCL.v	rule_absorbl_apply_covers
299	2.5.2/Lemma 33.35	FCL.v	prefix_target_groups
300	2.5.2/Lemma 33.36	FCL.v	nil_reduceMultiArrows
301	2.5.2/Lemma 33.37	FCL.v	empty_inhabit_cover
302	2.5.2/Lemma 33.38	FCL.v	cover_targets
303	2.5.2/Lemma 33.39	FCL.v	inhabit_cover_complete
304	2.5.2/Lemma 33.40	FCL.v	inhabit_step_complete
305	2.5.2/Lemma 33.41	FCL.v	inhabit_multistep_complete
306	2.5.2/Theorem 6.1	FCL.v	inhabit_sound

Section/Statement		File	Name in Coq
307	2.5.2/Theorem 6.2	FCL.v	`inhabit_complete`
308	2.6.2/Lemma 34.1	TwoCounter.v	`sound_Sigma__Aut`
309	2.6.2/Lemma 34.2	TwoCounter.v	`sound`
310	2.6.2/Lemma 34.3	TwoCounter.v	`complete`
311	2.6.3/Lemma 35.1	DependentFixpoint.v	`Fix_F_inv`
312	2.6.3/Lemma 35.2	DependentFixpoint.v	`isFix_unique`
313	2.6/Lemma 36.1	Algebra.v	`canonical_morphism_commutes`
314	2.6/Lemma 36.2	Algebra.v	`canonical_morphism_alg_morphism`
315	2.6/Lemma 36.3	Algebra.v	`canonical_morphism_unique`
316	2.6/Lemma 36.4	Algebra.v	`canonical_morphism_sound`
317	2.6/Lemma 36.5	Algebra.v	`canonical_morphism_complete`
318	2.6/Lemma 37.1	Algebra.v	`embed_unembed`
319	2.6/Lemma 37.2	Algebra.v	`embed_le`
320	2.6/Lemma 37.3	Algebra.v	`proofAction_FCL`
321	2.6/Lemma 37.4	Algebra.v	`unapplyNotIndex`
322	2.6/Lemma 37.5	Algebra.v	`arrow_le`
323	2.6/Lemma 37.6	Algebra.v	`indexType_sound`
324	2.6/Lemma 37.7	Algebra.v	`unapplyIsIndex`
325	2.6/Lemma 37.8	Algebra.v	`termCoAction_size`
326	2.6/Lemma 37.9	Algebra.v	`proofCoAction__FCL`
327	2.6/Lemma 37.10	Algebra.v	`range_coAction`
328	2.6/Lemma 38.1	Algebra.v	`Term_unapply_ind`
329	2.6/Lemma 38.2	Algebra.v	`IsChild_wf`
330	2.6/Lemma 38.3	Algebra.v	`dec_coAction__FCL`
331	2.6/Lemma 38.4	Algebra.v	`cancel_action_coAction__FCL`
332	2.6/Lemma 38.5	Algebra.v	`cancel_coAction__FCL_action`
333	2.6/Theorem 7.1	Algebra.v	`unique`
334	2.6/Theorem 7.2	Algebra.v	`sound`
335	2.6/Theorem 7.3	Algebra.v	`complete`

Table A.1: Statements and where to find their Coq formalization in [15]

Bibliography

[1] J. Adámek and S. Milius. Introduction to Category Theory, Algebras and Coalgebra, 2010. URL https://www.tu-braunschweig.de/Medien-DB/iti/survey_full.pdf. Lecture Notes of the European Summer School on Logic, Language and Information (ESSLLI 2010).

[2] R. Alur, R. Bodík, 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 *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013. URL http://ieeexplore.ieee.org/document/6679385/.

[3] R. Alur, D. Fisman, R. Singh, A. Solar-Lezama, et al. Syntax Guided Synthesis (SyGus) Competitions, 2014–2019. URL http://sygus.seas.upenn.edu/.

[4] Apache Foundation. Apache License Version 2.0, 2004. URL http://www.apache.org/licenses/LICENSE-2.0.

[5] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.*, 39(1):63–79, 2013. doi: 10.1109/TSE.2011.120. URL https://doi.org/10.1109/TSE.2011.120.

[6] Association for Computing Machinery (ACM) Special Interest Group on programming languages (SIGPLAN). Synthesis Track at Principles of Programming Languages (POPL), 2013–2019. URL http://www.sigplan.org/Conferences/POPL/.

[7] L. Augustsson. Djinn, a theorem prover in Haskell, for Haskell, 2005. URL http://hackage.haskell.org/package/djinn.

[8] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.*, 48(4):931–940, 1983. doi: 10.2307/2273659. URL https://doi.org/10.2307/2273659.

[9] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. ISBN 978-0-521-76614-2. URL http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.

[10] B. Barras. Sets in Coq, Coq in Sets. *J. Formalized Reasoning*, 3(1):29–48, 2010. doi: 10.6092/issn.1972-5787/1695. URL https://doi.org/10.6092/issn.1972-5787/1695.

[11] M. Ben-Soula, J. Hesse, et al. Maven Central - Repository for jar-file based dependencies, 2019. URL https://search.maven.org/.

[12] C.-B. Ben-Yelles. *Type assignment in the lambda-calculus: Syntax and semantics.* PhD thesis, University College of Swansea, 1979.

[13] M. Benke, A. Schubert, and D. Walukiewicz-Chrzaszcz. Synthesis of Functional Programs with Help of First-Order Intuitionistic Logic. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, pages 12:1–12:16, 2016. doi: 10.4230/LIPIcs.FSCD.2016.12. URL https://doi.org/10.4230/LIPIcs.FSCD.2016.12.

[14] J. Bessai. Synthesizing Dependency Injection Configurations for the Spring Framework, 2014. Master's Thesis, Technical University of Dortmund.

[15] J. Bessai. cls-coq: The Coq Formalization of the (CL)S Framework, 2019. URL https://doi.org/10.5281/zenodo.3243398.

[16] J. Bessai. Usage example of cls-scala, 2019. URL https://github.com/combinators/nosurrender.

[17] J. Bessai. cls-coq: Old version of the Coq Formalization of the (CL)S Framework, 2019. URL https://doi.org/10.5281/zenodo.3243395.

[18] J. Bessai and A. Vasileva. User Support for the Combinator Logic Synthesizer Framework. In *Proceedings 4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC 2018, Oxford, England, 14 July 2018.*, pages 16–25, 2018. doi: 10.4204/EPTCS.284.2. URL https://doi.org/10.4204/EPTCS.284.2.

[19] J. Bessai, B. Düdder, A. Dudenhefer, and M. Martens. Delegation-based Mixin Composition Synthesis. *Proceedings of Intersection Types and Related Systems (ITRS'14), Lecture Notes in Computer Science. Springer, 2014a.(Cited on pages 14 and 115)*, 2014.

[20] 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. doi: 10.1007/978-3-662-45234-9_3. URL https://doi.org/10.1007/978-3-662-45234-9_3.

[21] J. Bessai, B. Düdder, G. T. Heineman, and J. Rehof. Combinatory Synthesis of Classes Using Feature Grammars. In *Formal Aspects of Component Software - 12th International Conference, FACS 2015, Niterói, Brazil, October 14-16, 2015, Revised Selected Papers*, pages 123–140, 2015. doi: 10.1007/978-3-319-28934-2_7. URL https://doi.org/10.1007/978-3-319-28934-2_7.

[22] J. Bessai, A. Dudenhefner, B. Düdder, T. Chen, U. de'Liguoro, and J. Rehof. Mixin Composition Synthesis Based on Intersection Types. In *13th International Conference*

on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland, pages 76–91, 2015. doi: 10.4230/LIPIcs.TLCA.2015.76. URL https://doi.org/10.4230/LIPIcs.TLCA.2015.76.

[23] 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. URL https://doi.org/10.1007/978-3-319-47166-2_19.

[24] J. Bessai, A. Dudenhefner, B. Düdder, and J. Rehof. Extracting a formally verified Subtyping Algorithm for Intersection Types from Ideals and Filters. *Types*, 2016.

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

[26] J. Bessai, B. Düdder, G. T. Heineman, and J. Rehof. Towards Language-independent Code Synthesis, 2018. URL https://popl18.sigplan.org/details/PEPM-2018/12/Towards-Language-independent-Code-Synthesis-Poster-Demo-Talk-. Poster and Talk.

[27] J. Bessai, B. Düdder, G. T. Heineman, et al. (CL)S Ecosystem, 2019. URL https://www.combinators.org.

[28] J. Bessai, J. Rehof, and B. Düdder. Fast Verified BCD Subtyping. In T. Margaria, S. Graf, and K. G. Larsen, editors, *Models, Mindsets, Meta: The What, the How, and the Why Not?*, volume 11200 of *LNCS*. Springer, 2019. To appear.

[29] J. Bessai, A. Vasileva, and M. Roidl. Laser Visualization User Support for a cls-scala Use-Case, 2019. URL https://github.com/combinators/labyrinth.

[30] X. Bi, B. C. d. S. Oliveira, and T. Schrijvers. The Essence of Nested Composition. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, pages 22:1–22:33, 2018. URL https://doi.org/10.4230/LIPIcs.ECOOP.2018.22.

[31] R. Bodík, S. Gulwani, and E. Yahav. Software Synthesis (Dagstuhl Seminar 12152). *Dagstuhl Reports*, 2(4):21–38, 2012. doi: 10.4230/DagRep.2.4.21. URL https://doi.org/10.4230/DagRep.2.4.21.

[32] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, pages 107–121, 2010. doi: 10.1007/978-3-642-14203-1_9. URL https://doi.org/10.1007/978-3-642-14203-1_9.

[33] G. Bort et al. Play Web Framework, 2019. URL https://www.playframework.com/.

[34] B. Brayton et al. International Workshop on Logic and Synthesis, 1987–2019. URL http://www.iwls.org.

[35] S. Broda and L. Damas. Counting a Type's (Principal) Inhabitants. *Fundam. Inform.*, 45 (1-2):33–51, 2001. URL http://content.iospress.com/articles/fundamenta-informaticae/fi45-1-2-03.

[36] J. R. Büchi and L. H. Landweber. Solving Sequential Conditions by Finite-State Strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[37] W. E. Byrd and D. P. Friedman. From Variadic Functions to Variadic Relations – A miniKaren Perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06*, pages 105–117, 2006.

[38] W. E. Byrd, N. Amin, et al. Workshop on miniKaren, 2019. URL https://icfp19.sigplan.org/home/minikanren-2019.

[39] W. E. Byrd et al. Homepage of the miniKaren Language, 2019. URL http://minikanren.org/.

[40] F. Cardone and J. R. Hindley. Lambda-Calculus and Combinators in the 20th Century. In *Logic from Russell to Church*, pages 723–817. Elsevier, 2009. doi: 10.1016/S1874-5857(09)70018-4. URL https://doi.org/10.1016/S1874-5857(09)70018-4.

[41] J. Cockx, D. Devriese, and F. Piessens. Eliminating dependent pattern matching without K. *J. Funct. Program.*, 26:e16, 2016. doi: 10.1017/S0956796816000174. URL https://doi.org/10.1017/S0956796816000174.

[42] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. Online publication, October 2007. URL http://www.grappa.univ-lille3.fr/tata.

[43] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Math. Log. Q.*, 27(2-6):45–58, 1981. doi: 10.1002/malq.19810270205. URL https://doi.org/10.1002/malq.19810270205.

[44] L. Czajka, B. Ekici, and C. Kaliszyk. Concrete Semantics with Coq and CoqHammer. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pages 53–59, 2018. doi: 10.1007/978-3-319-96812-4_5. URL https://doi.org/10.1007/978-3-319-96812-4_5.

[45] F. M. Damm. Subtyping with Union Types, Intersection Types and Recursive Types. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 687–706, 1994. URL https://doi.org/10.1007/3-540-57887-0_121.

[46] F. M. Damm. *Subtyping with Union Types, Intersection Types and Recursive Types II*. PhD thesis, INRIA, 1994.

[47] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001. doi: 10.1145/382780.382785. URL https://doi.org/10.1145/382780.382785.

[48] J. Denis. Entwurf und Implementierung eines synthesebasierten Ansatzes für die flexible Ausführung von Geschäftsprozessen, 2018. Bachelor's Thesis, Technical University of Dortmund.

[49] M. Dezani-Ciancaglini and J. R. Hindley. Intersection Types for Combinatory Logic. *Theor. Comput. Sci.*, 100(2):303–324, 1992. doi: 10.1016/0304-3975(92)90306-Z. URL https://doi.org/10.1016/0304-3975(92)90306-Z.

[50] M. Dezani-Ciancaglini and J. R. Hindley. Intersection Types for Combinatory Logic. *Theor. Comput. Sci.*, 100(2):303–324, 1992. doi: 10.1016/0304-3975(92)90306-Z. URL https://doi.org/10.1016/0304-3975(92)90306-Z.

[51] C. Dion. I surrender. In *A New Day Has Come*. Columbia Records, 2002. Written by Louis Biancaniello and Sam Watters.

[52] E. Doberkat. *Special Topics in Mathematics for Computer Scientists - Sets, Categories, Topologies and Measures*. Springer, 2015. ISBN 978-3-319-22749-8. doi: 10.1007/978-3-319-22750-4. URL https://doi.org/10.1007/978-3-319-22750-4.

[53] E.-E. Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Walter de Gruyter, 2012. ISBN 978-3486714173.

[54] Docker, Inc. The Docker Container Platform, 2019. URL www.docker.com.

[55] B. Düdder. *Automatic Synthesis of Component & Connector Software Architectures with Bounded Combinatory Logic*. PhD thesis, Technical University Dortmund, Germany, 2014.

[56] B. Düdder, O. Garbe, M. Martens, J. Rehof, and P. Urzyczyn. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Composition Synthesis. In *Proceedings Sixth Workshop on Intersection Types and Related Systems, ITRS 2012, Dubrovnik, Croatia, 29th June 2012.*, pages 18–34, 2012. doi: 10.4204/EPTCS.121.2. URL https://doi.org/10.4204/EPTCS.121.2.

[57] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Bounded Combinatory Logic. In *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL, CSL 2012, September 3-6, 2012, Fontainebleau, France*, pages 243–258, 2012. doi: 10.4230/LIPIcs.CSL.2012.243. URL https://doi.org/10.4230/LIPIcs.CSL.2012.243.

[58] B. Düdder, M. Martens, and J. Rehof. Intersection Type Matching with Subtyping. In *Typed Lambda Calculi and Applications, 11th International Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, pages 125–139, 2013. doi: 10.1007/978-3-642-38946-7_11. URL https://doi.org/10.1007/978-3-642-38946-7_11.

[59] B. Düdder, M. Martens, and J. Rehof. Staged Composition Synthesis. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 67–86, 2014. doi: 10.1007/978-3-642-54833-8_5. URL https://doi.org/10.1007/978-3-642-54833-8_5.

[60] 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 https://doi.org/10.1145/2791060.2793677.

[61] A. Dudenhefner. *Algorithmic Aspects of Type-Based Program Synthesis*. PhD thesis, Technical University Dortmund, Germany, 2019.

[62] A. Dudenhefner. cls-fsharp, 2019. URL https://github.com/mrhaandi/cls-fsharp.

[63] A. Dudenhefner and J. Rehof. The Complexity of Principal Inhabitation. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, pages 15:1–15:14, 2017. doi: 10.4230/LIPIcs.FSCD.2017.15. URL https://doi.org/10.4230/LIPIcs.FSCD.2017.15.

[64] A. Dudenhefner and J. Rehof. Typability in bounded dimension. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, 2017. doi: 10.1109/LICS.2017.8005127. URL https://doi.org/10.1109/LICS.2017.8005127.

[65] A. Dudenhefner and J. Rehof. Intersection type calculi of bounded dimension. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 653–665, 2017. URL http://dl.acm.org/citation.cfm?id=3009862.

[66] A. Dudenhefner and J. Rehof. Principality and approximation under dimensional bound. *PACMPL*, 3(POPL):8:1–8:29, 2019. doi: 10.1145/3290321. URL https://doi.org/10.1145/3290321.

[67] A. Dudenhefner, M. Martens, and J. Rehof. The Algebraic Intersection Type Unification Problem. *Logical Methods in Computer Science*, 13(3), 2017. URL https://doi.org/10.23638/LMCS-13(3:9)2017.

[68] J. Duregård, P. Jansson, and M. Wang. Feat: functional enumeration of algebraic types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, pages 61–72, 2012. doi: 10.1145/2364506.2364515. URL https://doi.org/10.1145/2364506.2364515.

[69] R. Dyckhoff. Contraction-Free Sequent Calculi for intuitionistic Logic: a correction. *J. Symb. Log.*, 83(4):1680–1682, 2018. doi: 10.1017/jsl.2018.38. URL https://doi.org/10.1017/jsl.2018.38.

[70] H. Ehrich. On the Theory of Specification, Implementation, and Parametrization of Abstract Data Types. *Journal of the ACM*, 29(1):206–227, 1982. URL https://doi.org/10.1145/322290.322303.

[71] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations und Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985. ISBN 3-540-13718-1. URL https://doi.org/10.1007/978-3-642-69962-7.

[72] K. Ellis, E. Dechter, and J. B. Tenenbaum. Dimensionality Reduction via Program Induction. In *2015 AAAI Spring Symposia, Stanford University, Palo Alto, California, USA, March 22-25, 2015*, 2015. URL http://www.aaai.org/ocs/index.php/SSS/SSS15/paper/view/10284.

[73] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 422–436, 2017. doi: 10.1145/3062341.3062351. URL https://doi.org/10.1145/3062341.3062351.

[74] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017. URL http://dl.acm.org/citation.cfm?id=3009851.

[75] J. Filliâtre and S. Conchon. Type-safe modular hash-consing. In *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*, pages 12–19, 2006. doi: 10.1145/1159876.1159880. URL https://doi.org/10.1145/1159876.1159880.

[76] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Synthesizing Functional Reactive Programs. *CoRR*, abs/1905.09825, 2019. URL http://arxiv.org/abs/1905.09825.

[77] P. C. Fischer, A. R. Meyer, and A. L. Rosenberg. Counter Machines and Counter Languages. *Mathematical Systems Theory*, 2(3):265–283, 1968. doi: 10.1007/BF01694011. URL https://doi.org/10.1007/BF01694011.

[78] J. Frankle, P. 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 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815, 2016. doi: 10.1145/2837614.2837629. URL https://doi.org/10.1145/2837614.2837629.

[79] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. MIT Press, 2005. ISBN 978-0-262-56214-0.

[80] S. Fuchs et al. Travis CI - Continuous Integration Build Service, 2019. URL https://travis-ci.org/.

[81] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, pages 406–431, 1993. doi: 10.1007/3-540-47910-4_21. URL https://doi.org/10.1007/3-540-47910-4_21.

[82] S. Ghilezan. Strong Normalization and Typability with Intersection Types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996. URL https://doi.org/10.1305/ndjfl/1040067315.

[83] M. Gogolla. Partially Ordered Sorts in Algebraic Specifications. In *Proc. Of the Conference on Ninth Colloquium on Trees in Algebra and Programming*, pages 139–153, New York, NY, USA, 1984. Cambridge University Press. ISBN 0-521-26750-1. URL http://dl.acm.org/citation.cfm?id=2868.2878.

[84] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial Algebra Semantics and Continuous Algebras. *J. ACM*, 24(1):68–95, 1977. doi: 10.1145/321992.321997. URL https://doi.org/10.1145/321992.321997.

[85] G. Gonthier, A. Mahboubi, and E. Tassi. A small scale reflection extension for the Coq system. Technical report, Inria Saclay Ile de France, 2013. URL https://hal.inria.fr/inria-00258384v17.

[86] G. Grätzer. *Lattice theory: First concepts and distributive lattices*. Dover Books on Mathematics. Dover Publications, 2009.

[87] C. Green et al. Kestrel Institute, 1981-2019. URL https://www.kestrel.edu/.

[88] C. Green et al. International Conference on Automated Software Engineering (ASE), 1987–2019. URL InternationalConferenceonAutomatedSoftwareEngineering.

[89] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 631–642, 2016. doi: 10.1145/2950290.2950334. URL https://doi.org/10.1145/2950290.2950334.

[90] S. Gulwani, O. Polozov, and R. Singh. Program Synthesis. *Foundations and Trends in Programming Languages*, 4(1-2):1–119, 2017. doi: 10.1561/2500000010. URL https://doi.org/10.1561/2500000010.

[91] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 27–38, 2013. doi: 10.1145/2491956.2462192. URL https://doi.org/10.1145/2491956.2462192.

[92] C. Haack, B. Howard, A. Stoughton, and J. B. Wells. Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, pages 83–98, 2002. doi: 10.1007/3-540-45719-4_7. URL https://doi.org/10.1007/3-540-45719-4_7.

[93] M. Harrah, E. Yokota, et al. sbt - The Scala Build Tool, 2019. URL https://www.scala-sbt.org.

[94] M. Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4): 413–436, 1998. URL http://journals.cambridge.org/action/displayAbstract?aid=44199.

[95] G. T. Heineman, A. Hoxha, B. Düdder, and J. Rehof. Towards migrating object-oriented frameworks to enable synthesis of product line members. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 56–60, 2015. doi: 10.1145/2791060.2791076. URL https://doi.org/10.1145/2791060.2791076.

[96] 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. URL https://doi.org/10.1007/978-3-319-47166-2_21.

[97] G. T. Heineman, J. Bessai, , and B. Düdder. EpCoGen - Expression Problem Code Generator, 2019. URL https://github.com/combinators/expression-problem.

[98] G. T. Heineman et al. Nextgen Solitaire - A Software Product Line of Solitaire Games, 2019. URL https://github.com/combinators/nextgen-solitaire.

[99] J. R. Hindley. *Basic Simple Type Theory*, volume 42. Cambridge University Press, 1997.

[100] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

[101] R. Hinze and J. Jeuring. Weaving a web. *J. Funct. Program.*, 11(6):681–689, 2001. doi: 10.1017/S0956796801004129. URL https://doi.org/10.1017/S0956796801004129.

[102] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12 (10):576–580, 1969. doi: 10.1145/363235.363259. URL https://doi.org/10.1145/363235.363259.

[103] F. Honsell, L. Liquori, C. Stolze, and I. Scagnetto. The Delta-Framework. In *38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, pages 37:1–37:21, 2018. doi: 10.4230/LIPIcs.FSTTCS.2018.37. URL https://doi.org/10.4230/LIPIcs.FSTTCS.2018.37.

[104] G. P. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, 1997. URL http://journals.cambridge.org/action/displayAbstract?aid=44121.

[105] J. P. Inala, N. Polikarpova, X. Qiu, B. S. Lerner, and A. Solar-Lezama. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 247–263, 2017. doi: 10.1007/978-3-662-54577-5_14. URL https://doi.org/10.1007/978-3-662-54577-5_14.

[106] International Organization for Standardization (ISO). ISO/IEC 13211-1:1995: Information Technology—Programming Languages—Prolog—Part 1: General Core. *ISO: Geneva, Switzerland*, pages 1–199, 1995.

[107] 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.

[108] D. Jackson. *Software Abstractions - Logic, Language, and Analysis.* MIT Press, 2006. ISBN 978-0-262-10114-1. URL http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928.

[109] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.1418.

[110] S. Jacobs, R. Bloem, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, M. Luttenberger, P. J. Meyer, T. Michaud, M. Sakr, S. Sickert, L. Tentrup, and A. Walker. The 5th Reactive Synthesis Competition (SYNTCOMP 2018): Benchmarks, Participants & Results. *CoRR*, abs/1904.07736, 2019. URL http://arxiv.org/abs/1904.07736.

[111] G. T. H. Jan Bessai, Boris Düdder. cls-scala, 2019. URL https://github.com/combinators/cls-scala.

[112] C. Kaliszyk and J. Urban. Learning-Assisted Automated Reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014. doi: 10.1007/s10817-014-9303-3. URL https://doi.org/10.1007/s10817-014-9303-3.

[113] C. Kaliszyk, J. C. Blanchette, et al. HaTT: Workshop on Hammers for Type Theories and related tools, 2016. URL https://hatt2016.inria.fr.

[114] M. Kintel et al. OpenSCAD - The Programmers Solid 3D CAD Modeller, 2019. URL https://www.openscad.org/about.html.

[115] O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn,*

Estonia, September 26-28, 2005, pages 192–203, 2005. doi: 10.1145/1086365.1086390. URL https://doi.org/10.1145/1086365.1086390.

[116] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 379–394, 2019. doi: 10.1145/3314221.3314602. URL https://doi.org/10.1145/3314221.3314602.

[117] D. Kozen. *Automata and Computability*. Undergraduate texts in computer science. Springer, 1997. ISBN 978-0-387-94907-9.

[118] T. Kurata and M. Takahashi. Decidable Properties of Intersection Type Systems. In *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, pages 297–311, 1995. URL https://doi.org/10.1007/BFb0014060.

[119] Kurt-Gödel-Gesellschaft. Vienna Summer of Logic, 2014. URL http://vsl2014.at.

[120] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266):1332–1338, 2015.

[121] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103 (2):151–161, 1968. doi: 10.1007/BF01110627. URL https://doi.org/10.1007/BF01110627.

[122] R. Lämmel. Google's MapReduce programming model - Revisited. *Science of Computer Programming*, 70(1):1–30, 2008. URL https://doi.org/10.1016/j.scico.2007.07.001.

[123] D. Larchey-Wendling and J.-F. Monin. Simulating Induction-Recursion for Partial Algorithms. In *24th International Conference on Types for Proofs and Programs, TYPES 2018, June 18-21, 2018, Braga, Portugal*, 2018.

[124] K.-K. Lau et al. LOPSTR: Logic-Based Program Synthesis and Transformation, 1991–2019. URL http://lopstr.webs.upv.es/.

[125] O. Laurent. Intersection subtyping with constructors. In *Proceedings of the 9th Workshop on Intersection Types and Related Systems*, 2018.

[126] P. Liang, M. I. Jordan, and D. Klein. Learning Programs: A Hierarchical Bayesian Approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 639–646, 2010. URL https://icml.cc/Conferences/2010/papers/568.pdf.

[127] P. L. M. Clavel, S. Eker and J. Meseguer. Principles of Maude. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[128] J. P. Magalhães and A. Löh. A Formal Comparison of Approaches to Datatype-Generic Programming. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2012, Tallinn, Estonia, 25 March 2012.*, pages 50–67, 2012. doi: 10.4204/EPTCS.76.6. URL https://doi.org/10.4204/EPTCS.76.6.

[129] A. Mahboubi, E. Tassi, Y. Bertot, and G. Gonthier. *Mathematical Components.* Online publication, 2018. URL https://math-comp.github.io/mcb/.

[130] Z. Manna and R. J. Waldinger. Synthesis: Dreams - Programs. *IEEE Trans. Software Eng.*, 5(4):294–328, 1979. doi: 10.1109/TSE.1979.234198. URL https://doi.org/10.1109/TSE.1979.234198.

[131] Z. Manna and R. J. Waldinger. A Deductive Approach to Program Synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980. doi: 10.1145/357084.357090. URL https://doi.org/10.1145/357084.357090.

[132] Z. Manna and R. J. Waldinger. Deductive Synthesis of the Unification Algorithm. *Sci. Comput. Program.*, 1(1-2):5–48, 1981. doi: 10.1016/0167-6423(81)90004-6. URL https://doi.org/10.1016/0167-6423(81)90004-6.

[133] P. Mayer and A. Bauer. An empirical analysis of the utilization of multiple programming languages in open source projects. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, EASE 2015, Nanjing, China, April 27-29, 2015*, pages 4:1–4:10, 2015. doi: 10.1145/2745802.2745805. URL https://doi.org/10.1145/2745802.2745805.

[134] P. Mayer, M. Kirsch, and M. A. Le. On multi-language software development, cross-language links and accompanying tools: a survey of professional software developers. *J. Software Eng. R&D*, 5:1, 2017. doi: 10.1186/s40411-017-0035-z. URL https://doi.org/10.1186/s40411-017-0035-z.

[135] C. McBride. Djinn, Monotonic. In *PAR@ ITP*, pages 14–17, 2010.

[136] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, pages 124–144, 1991. doi: 10.1007/3540543961_7. URL https://doi.org/10.1007/3540543961_7.

[137] N. Merwin, L. Donahoe, et al. Coveralls.io - Test coverage analysis service, 2019. URL https://coveralls.io.

[138] A. R. Meyer. What is a Model of the Lambda Calculus? *Information and Control*, 52(1):87–122, 1982. doi: 10.1016/S0019-9958(82)80087-9. URL https://doi.org/10.1016/S0019-9958(82)80087-9.

[139] R. Milner. A Theory of Type Polymorphism in Programming. *J. Comput. Syst. Sci.*, 17 (3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL https://doi.org/10.1016/0022-0000(78)90014-4.

[140] M. Minsky. Recursive Unsolvability of Post's Problem of "tag": And Other Topics in Theory of Truing Machines. *Annals of Math.*, 74:437–455, 1961.

[141] J. Monin and X. Shi. Handcrafted Inversions Made Operational on Operational Semantics. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 338–353, 2013. doi: 10.1007/978-3-642-39634-2_25. URL https://doi.org/10.1007/978-3-642-39634-2_25.

[142] D. Naczinski. Implementierung eines Java-basierten Python3-Parsers und Syntaxbaum-manipulationswerkzeugs, 2019. Bachelor's Thesis, Technical University of Dortmund.

[143] H. Nienhuys, J. Nieuwenhuizen, et al. The LilyPond music engraving program, 2019. URL http://lilypond.org/.

[144] R. Nilsson et al. ScalaCheck: Property-based testing for Scala, 2019. URL https://www.scalacheck.org/.

[145] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2.

[146] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015. doi: 10.1145/2737924.2738007. URL https://doi.org/10.1145/2737924.2738007.

[147] P.-M. S. Osera. *Program synthesis with types*. PhD thesis, University of Pennsylvania, 2015.

[148] P. Padawitz. The Equational Theory of Parameterized Specifications. *Inf. Comput.*, 76 (2/3):121–137, 1988. doi: 10.1016/0890-5401(88)90006-5. URL https://doi.org/10.1016/0890-5401(88)90006-5.

[149] P. Padawitz. Expander2: Program Verification Between Interaction and Automation. *Electr. Notes Theor. Comput. Sci.*, 177:35–57, 2007. doi: 10.1016/j.entcs.2007.01.003. URL https://doi.org/10.1016/j.entcs.2007.01.003.

[150] P. Padawitz. From Grammars and Automata to Algebras and Coalgebras. In *Algebraic Informatics - 4th International Conference, CAI 2011, Linz, Austria, June 21-24, 2011. Proceedings*, pages 21–43, 2011. URL https://doi.org/10.1007/978-3-642-21493-6_2.

[151] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. D. Millstein. FlashProfile: a framework for synthesizing data profiles. *PACMPL*, 2(OOPSLA):150:1–150:28, 2018. doi: 10.1145/3276520. URL https://doi.org/10.1145/3276520.

[152] C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN 978-0-201-53082-7.

[153] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-Symbolic Program Synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL https://openreview.net/forum?id=rJ0JwFcex.

[154] L. C. Paulson. Constructing Recursion Operators in Intuitionistic Type Theory. *J. Symb. Comput.*, 2(4):325–355, 1986. doi: 10.1016/S0747-7171(86)80002-5. URL https://doi.org/10.1016/S0747-7171(86)80002-5.

[155] M. Pennekamp. Simulation von Berechnungsmodellen innerhalb des Softwaresynthese-frameworks CLS, 2017. Bachelor's Thesis, Technical University of Dortmund.

[156] B. C. Pierce. *A decision procedure for the subtype relation on intersection types with bounded variables*. PhD thesis, Carnegie-Mellon University. Department of Computer Science, 1989. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.8003&rep=rep1&type=pdf.

[157] R. Piskac. *Decision Procedures for Program Synthesis and Verification*. PhD thesis, EPFL, 2011. URL https://infoscience.epfl.ch/record/168994.

[158] A. Pnueli and R. Rosner. On the Synthesis of a Reactive Module. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 179–190, 1989. doi: 10.1145/75277.75293. URL https://doi.org/10.1145/75277.75293.

[159] N. Polikarpova and I. Sergey. Structuring the synthesis of heap-manipulating programs. *PACMPL*, 3(POPL):72:1–72:30, 2019. doi: 10.1145/3290385. URL https://doi.org/10.1145/3290385.

[160] 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 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538, 2016. doi: 10.1145/2908080.2908093. URL https://doi.org/10.1145/2908080.2908093.

[161] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky. A Generic Parallel Collection Framework. In *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part II*, pages 136–147, 2011. doi: 10.1007/978-3-642-23397-5_14. URL https://doi.org/10.1007/978-3-642-23397-5_14.

[162] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999. ISBN 0-201-59604-0.

[163] J. Rehof. Towards Combinatory Logic Synthesis. In *1st International Workshop on Behavioural Types, BEAT*, 2013. URL https://pdfs.semanticscholar.org/8754/d21b46bf9b1a66b22936d1d71c377d3c1fdb.pdf.

[164] J. Rehof and G. T. Heineman. Modular synthesis of product lines (ModSyn-PL). In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, page 393, 2015. doi: 10.1145/2791060.2791061. URL https://doi.org/10.1145/2791060.2791061.

[165] 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.

[166] J. Rehof and M. Y. Vardi. Design and Synthesis from Components (Dagstuhl Seminar 14232). *Dagstuhl Reports*, 4(6):29–47, 2014. ISSN 2192-5283. doi: 10.4230/DagRep.4.6.29. URL http://drops.dagstuhl.de/opus/volltexte/2014/4683.

[167] A. Reynolds and C. Tinelli. SyGuS Techniques in the Core of an SMT Solver. In *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017.*, pages 81–96, 2017. doi: 10.4204/EPTCS.260.8. URL https://doi.org/10.4204/EPTCS.260.8.

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

[169] M. Sabin et al. shapeless: Generic Programming for Scala, 2019. URL https://github.com/milessabin/shapeless.

[170] T. Schäfer. Process Synthesis based on the Higher-Order Arrow Calculus, 2018. Master's Thesis, Technical University of Dortmund.

[171] 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 *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pages 472–486, 2018. doi: 10.1007/978-3-030-03427-6_35. URL https://doi.org/10.1007/978-3-030-03427-6_35.

[172] 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 *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pages 472–486, 2018. doi: 10.1007/978-3-030-03427-6_35. URL https://doi.org/10.1007/978-3-030-03427-6_35.

[173] D. Scholtyssek. Synthese von Docker-Konfigurationen unter Zuhilfenahme eines Inhabitationsalgorithmus, 2017. Bachelor's Thesis, Technical University of Dortmund.

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

[175] R. Shin, I. Polosukhin, and D. Song. Improving Neural Program Synthesis with Inferred Execution Traces. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 8931–8940, 2018. URL http://papers.nips.cc/paper/8107-improving-neural-program-synthesis-with-inferred-execution-traces.

[176] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[177] B. Springsteen. No Surrender. In *Born in the U.S.A.* Columbia Records, 1984.

[178] Y. V. Srinivas and R. Jüllig. Specware: Formal Support for Composing Software. In *Mathematics of Program Construction, MPC'95, Kloster Irsee, Germany, July 17-21, 1995, Proceedings*, pages 399–422, 1995. doi: 10.1007/3-540-60117-1_22. URL https://doi.org/10.1007/3-540-60117-1_22.

[179] R. Statman. Intuitionistic Propositional Logic is Polynomial-Space Complete. *Theor. Comput. Sci.*, 9:67–72, 1979. doi: 10.1016/0304-3975(79)90006-9. URL https://doi.org/10.1016/0304-3975(79)90006-9.

[180] R. Statman. A Finite Model Property for Intersection Types. In *Proceedings Seventh Workshop on Intersection Types and Related Systems, ITRS 2014, Vienna, Austria, 18 July 2014.*, pages 1–9, 2014. URL https://doi.org/10.4204/EPTCS.177.1.

[181] B. Steffen, T. Margaria, and M. von der Beeck. Automatic Synthesis of Linear Process Models from Temporal Constraints: An Incremental Approach. In *ACM/SIGPLAN Int. Workshop on Automated Analysis of Software (AAS'97)*, 1997.

[182] P. Tabuada et al. SYNT 2018: Seventh Workshop on Synthesis, 2012–2019. URL http://synt2018.seas.ucla.edu/index.html.

[183] M. Tesche. Generierung von Blockdiagrammen im Linked Process Blocks Editor unter Verwendung des Combinatory Logic Synthesizers, 2018. Bachelor's Thesis, Technical University of Dortmund.

[184] The Coq Development Team. The Coq Proof Assistant, version 8.8.0, apr 2018. URL https://doi.org/10.5281/zenodo.1219885.

[185] W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software,*

ETAPS 2009, York, UK, March 22-29, 2009. Proceedings, pages 1–14, 2009. doi: 10.1007/978-3-642-00596-1_1. URL https://doi.org/10.1007/978-3-642-00596-1_1.

[186] A. Timany and M. Sozeau. Cumulative Inductive Types In Coq. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-077-4. doi: 10.4230/LIPIcs.FSCD.2018.29. URL http://drops.dagstuhl.de/opus/volltexte/2018/9199.

[187] A. Tinis. Generierung von Prozessvarianten für Microsoft Project Netzpläne mittels CLS, 2019. Bachelor's Thesis, Technical University of Dortmund.

[188] F. Tomassetti and M. Torchiano. An empirical assessment of polyglot-ism in GitHub. In *18th International Conference on Evaluation and Assessment in Software Engineering, EASE '14, London, England, United Kingdom, May 13-14, 2014*, pages 17:1–17:4, 2014. doi: 10.1145/2601248.2601269. URL https://doi.org/10.1145/2601248.2601269.

[189] L. Torvalds et al. Git Version Control System, 2005. URL https://git-scm.com/.

[190] J. Traugott. Deductive Synthesis of Sorting Programs. *J. Symb. Comput.*, 7(6): 533–572, 1989. doi: 10.1016/S0747-7171(89)80040-9. URL https://doi.org/10.1016/S0747-7171(89)80040-9.

[191] A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi: 10.1112/plms/s2-42.1.230. URL https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/plms/s2-42.1.230.

[192] Unity Labs. Unity Game Engine, 2019. URL www.unity.com.

[193] P. Urzyczyn. Inhabitation of Low-Rank Intersection Types. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 356–370, 2009. doi: 10.1007/978-3-642-02273-9_26. URL https://doi.org/10.1007/978-3-642-02273-9_26.

[194] S. van Bakel. Complete Restrictions of the Intersection Type Discipline. *Theor. Comput. Sci.*, 102(1):135–163, 1992. doi: 10.1016/0304-3975(92)90297-S. URL https://doi.org/10.1016/0304-3975(92)90297-S.

[195] D. van Bruggen et al. JavaParser Project, 2019. URL https://javaparser.org/.

[196] M. Veanes. On computational complexity of basic decision problems of finite tree automata. Technical report, UPMAIL Technical Report 133, Uppsala University, Computing Science Department, 1997. URL https://it.uu.se/research/csd/reports/0133.pdf.

[197] B. Venners, G. Berger, C. C. Seng, et al. Scala Test - A testing tool for Scala, 2019. URL http://www.scalatest.org/.

[198] J. Vican et al. sbt-release-early - sbt plugin for early publishing of build artifacts, 2019. URL https://github.com/scalacenter/sbt-release-early.

[199] P. Wadler. The Expression Problem, 1998. URL http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Email to the Java Genericity Mailing List.

[200] X. Wang, I. Dillig, and R. Singh. Synthesis of data completion scripts using finite tree automata. *PACMPL*, 1(OOPSLA):62:1–62:26, 2017. doi: 10.1145/3133886. URL https://doi.org/10.1145/3133886.

[201] 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 *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part IV*, pages 487–503, 2018. doi: 10.1007/978-3-030-03427-6_36. URL https://doi.org/10.1007/978-3-030-03427-6_36.

[202] L. Wittgenstein. *Tractatus logico-philosophicus*. Suhrkamp, Verlag Frankfurt, 2003. ISBN 978-3-518-10012-7. Written 1918 in Vienna.

[203] T. Wolf, M. Keppler, et al. EGit: Eclipse Team provider for the Git version control system, 2019. URL https://www.eclipse.org/egit/.

[204] J. Zappe. Automatische Erstellung eines verteilten Systems unter optimaler Ausnutzung der vorhandenen Ressourcen, 2018. Master's Thesis, Technical University of Dortmund.

[205] T. Zuckmantel. Concept and Implementation of a Java-based Parser and Pretty-Printer for the Coq Proof Assistant, 2018. Bachelor's Thesis, Technical University of Dortmund.