ALGORITHMIC ASPECTS OF TYPE-BASED PROGRAM 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

ANDREJ DUDENHEFNER

Dortmund

2019

Tag der mündlichen Prüfung: 24. Juni 2019
Dekan: Prof. Dr.-Ing. Gernot A. Fink
Gutachter:
Prof. Dr. Jakob Rehof (TU Dortmund University, Germany),
Prof. Dr. Hab. Paweł Urzyczyn (University of Warsaw, Poland)

Abstract

In the area of type-based program synthesis, inhabitant search in typed calculi can be utilized to enumerate programs that satisfy a specification given as a type. In particular, the decision problem of inhabitation (given a type environment Γ and a type τ, is there a term M such that M can be assigned the type τ in Γ?) corresponds to existence of a program (term M) that satisfies the given specification (type τ) under additional assumptions (type environment Γ).

Since the untyped λ-calculus is a Turing complete functional programming language, inhabitation in typed λ-calculi can be seen as program synthesis from scratch. Complementarily, inhabitation in combinatory logic can be seen as domain-specific program synthesis. Specifically, we may choose a computationally weak basis (set of combinators) containing a selection of domain-specific components from which the synthesized program is composed.

Varying the type language allows us to express different properties of synthesized results. While simple types can express higher-order functional dependencies, intersection types can express higher-order tabular dependencies.

Further restrictions on inhabitant search, such as principality and relevance restrictions, yield inhabitants that are more closely tied to given specifications. Alternatively, dimension, rank, order, and arity restrictions provide means to control the complexity of inhabitant search.

This work provides an overview over selected results in type-based program synthesis varying the term language (λ-terms and combinatory terms) as well as the type language (simple types and intersection types). Additionally further type-theoretic restrictions (principality, dimension, rank, order, and arity) are considered.

For λ-calculus we depict PSpace-completeness of principal inhabitation in the simply typed λ-calculus, undecidability of inhabitation in λ-calculus with intersection types, and undecidability of inhabitation in λ-calculus with intersection types in bounded dimension.

For combinatory logic we depict undecidability of inhabitation in subintuitionistic combinatory logic, $(o+2)$-ExpTime-completeness of inhabitation in combinatory logic with intersection types with instantiation of bounded order o, and ExpTime-hardness of intersection type unification.

Additionally, we provide an overview over the implementation of the inhabitant search algorithm (CL)S-F# for combinatory logic with intersection types. Finally, we evaluate (CL)S-F# in functional program synthesis, object-oriented program synthesis, and process synthesis scenarios.

Preface

This work provides an overview over a selection of the author's contributions in type-based program synthesis during his time between January 2014 and April 2019 as a PhD student under the supervision of Prof. Dr. Jakob Rehof at TU Dortmund University, Germany. Most of the presented results are published and are referenced accordingly. Since a mere collection of already published results would waste the reader's time, this work tries to paint a coherent picture focusing on motivation, interrelations, and positioning of the individual results in type-based program synthesis. Additional examples (which often are not part of the corresponding publications due to space limits) are provided to convey a better intuitive understanding of the technical results. Whenever a detailed proof of a contribution is already published (or formalized in a proof assistant), it is not repeated in this work in detail but instead explained at a higher level.

Acknowledgments

First and foremost, I am grateful for the helpful guidance and supervision of Jakob Rehof during my enlightening journey through type theory. During my time as a graduate student I did not have any strong feelings regarding type theory up until I visited Jakob's lecture on the Curry-Howard isomorphism. It sparked my interest not only in typed calculi, but also in the foundations of logic itself. Jakob's ability to ask intriguing research question and immediately point to relevant existing work has proven both a source of inspiration and a valuable resource. Overall, I am glad to have had Jakob as my adviser.

Many thanks go to Paweł Urzyczyn, whose ingenious contributions in typed λ-calculus and typed combinatory logic paved my journey through type theory. During the time as a PhD student I have spent countless hours trying to develop a deeper understanding of his sophisticated automata-based proofs. After all, "its all in the proof".

I want to extend my thanks to Ugo De'Liguoro and Simona Ronchi Della Rocca who in personal discussions provided deep insights into the intersection type discipline, that has been at the core of my research.

Finally, I want to thank my parents as well as my colleagues Boris Düdder and Jan Bessai for their support during my time in Dortmund.

Contents

Chapter 1

Introduction

Combinatory logic, pioneered by Moses Schönfinkel in 1920s, and λ-calculus, developed by Alonzo Church in 1930s, both predate Turing machines as Turing complete computation formalisms. From a modern perspective, both can be considered minimalistic functional programming languages (of course, the development has been vice versa). Between 1930s and 1940s, typed variants of both combinatory logic and λ-calculus were developed by Haskell Curry and Alonzo Church based on the notion of type theories used by Bertrand Russel in 1900s. Later, Haskell Curry and William Alvin Howard discovered that the simply typed λ-calculus corresponds to the implicational fragment of intuitionistic propositional logic. This correspondence intimately ties together logic and computation, extending far beyond intuitionistic propositional logic.

In this work we view combinatory logic and λ-calculus as clean room environments to study type-based functional program synthesis, inspecting properties of different type systems. A *type system* consists of a *term language* (here either combinatory terms or λ-terms), a type language (here either simple types or intersection types), and *type rules* to assign types to terms in *type environments* (here sets of pairs of term variable and type). Assignment of a type τ to a term M in the type environment Γ in a type system ($\vdash$) is denoted by the *judgement* $\Gamma \vdash M : \tau$. Key to type-based program synthesis is the decision problem of *inhabitation* (given a type environment Γ and a type τ, is there a term M such that $\Gamma \vdash M : \tau$ is derivable?). In particular, τ corresponds to a desired program specification and Γ contains additional domain-specific assumptions while M corresponds to the synthesized program satisfying the desired specification. As we will see throughout this work, intersection types provide a concise specification language for higher-order tabular dependencies, and are of interest in realistic synthesis scenarios.

There is a wide spectrum of expressiveness of type-based synthesis for λ-calculus (Table 1.1) and combinatory logic (Table 1.2) depending on the chosen type system (and possibly a restriction). This is reflected in varying complexity of the underlying decision problem of inhabitation.

The following Table 1.1 provides an overview over complexity of inhabitation in various typed λ-calculi.

Table 1.1: Complexity of Inhabitation in Typed λ-Calculi

Type Language	**Problem**	**Complexity**
Simple Types	Inhabitation	PSPACE-complete [62, 65]
Simple Types	Principal Inhabitation	PSPACE-complete [37] (Section 2.1)
Simple Types	Relevant Inhabitation	2-EXPTIME-complete [57]
Simple Types	Principal, Relevant Inhabitation	unknown (Section 2.1)
Intersection Types	Inhabitation in rank ≥ 3	undecidable [66] (Section 2.2)
Intersection Types	Inhabitation in rank 2	EXPSPACE-complete [66]
Intersection Types	Principal Inhabitation	in PTIME [39]
Explicit Intersection Types	Inhabitation	EXPSPACE-complete [56]
Non-idempotent Intersection Types	Inhabitation	NP-complete [16, 34]
Intersection Types	Inhabitation in bounded set-dimension	undecidable [34] (Section 2.3.3)
Intersection Types	One-dimensional Inhabitation	conjectured NP-complete (Section 2.3.1)
Intersection Types	Inhabitation in fixed set-dimension	conjectured undecidable (Section 2.3.3)
Intersection Types	Inhabitation in bounded multiset-dimension	EXPSPACE-complete [34]

Inhabitation in typed λ-calculi can be used for program synthesis *from scratch*. Most notably, the restriction to *principal* inhabitants (the given type is, in a sense, most general for the constructed inhabitant) is of interest to strengthen the correspondence between the specification and the synthesized program. For simple types principality does not influence the complexity of inhabitation. Surprisingly, for intersection types the complexity of inhabitation changes from undecidable to PTIME, if principality is required. Unfortunately, this jump in complexity implies a disproportionate increase in specification size.

Alternatively, the rank (functional nesting of intersection) restriction can be considered. However, the jump from rank 2 (EXPSPACE-complete) to rank 3 (undecidable) makes rank not well-suited as a practical control parameter.

Recently, *dimensional* restrictions of intersection typed λ-calculi have been considered. The dimension can be understood as the number of distinct "features" necessary to describe each part of a program, and is therefore a very practical measure. While also exposing decidable flavors, inhabitation in bounded dimension appears promising for type-based synthesis from scratch.

Complementarily, the following Table 1.2 provides an overview over complexity of inhabitation in various typed typed combinatory logics.

Table 1.2: Complexity of Inhabitation in Typed Combinatory Logics

Type Language	**Problem**	**Complexity**
Simple Types	SK-Inhabitation	PSPACE-complete [62]
Simple Types	BCIW-Inhabitation	2-EXPTIME-complete [57]
Simple Types	Relativized Inhabitation	undecidable [50]
Simple Types	Relativized, Principal Inhabitation	undecidable [35] (Section 3.1)
Simple Types	Relativized Inhabitation below $\alpha \to \beta \to \alpha$	undecidable [12] (Section 3.1)
Simple Types	Relativized Inhabitation below $\alpha \to \beta \to \beta$	in PTIME [35] (Section 3.1)
Intersection Types	SK-Inhabitation	undecidable [66, 27]
Explicit Intersection Types	SK-Inhabitation	EXPSPACE-complete [56, 27]
Monomorphic Intersection Types	Relativized Inhabitation	EXPTIME-complete [55]
Intersection Types with Constants	Relativized Inhabitation in level k	$(k+2)$-EXPTIME-complete [55]
Intersection Types with Constants	Relativized Inhabitation in order o and arity a	$(o+2)$-EXPTIME-complete (Section 3.2.1)
Intersection Types with Constants	Relativized Inhabitation in order o and arity a w/o Intersection Introduction	conjectured $(o+1)$-EXPTIME-complete (Section 3.2.2)

Inhabitation in combinatory logic with fixed bases is comparable to inhabitation in λ-calculi due to corresponding interpretation theorems. For type-based synthesis that include domain-specific knowledge *relativized* inhabitation in combinatory logic (the basis is part of the input and represents domain-specific components) is particularly useful. Differently from λ-calculus, relativized inhabitation is undecidable in combinatory logic even for simple types (also under principal and subintuitionistic restrictions). However, in practice (see Chapter 4 for an evaluation) bases are tailored to specific domains of interest and exhibit tractable inhabitant search.

Thesis Outline

This work as a whole is divided into three parts.

The first part (Chapter 2) focuses on typed λ-calculi. First, Section 2.1 outlines the contribution showing that principal inhabitation in the simply typed λ-calculus is PSPACE-complete [37]. The upper bound (Section 2.1.2) is shown algorithmically and the lower bound (Section 2.1.3) is shown by adapting Urzyczyn's construction [65] to encompass principality. Second, Section 2.2 describes the proof and formalization in the Coq proof assistant of undecidability of intersection type inhabitation based on [40]. Specifically, an undecidable word problem in simple semi-Thue systems is reduced to intersection type inhabitation (Section 2.2.2) referring to formalization of soundness and completeness [31]. Third, Section 2.3 gives an overview over the line of work covering key decision problems (such as inhabitation, typability, type checking) in dimensionally bounded λ-calculi with intersection types [34, 38, 39].

The second part (Chapter 3) focuses on typed combinatory logic. First, Section 3.1 outlines the contribution showing that inhabitation in the simply typed combinatory logic (even under several restrictions for considered type environments) is undecidable [35]. For this purpose, it is shown that recognizing principal axiomatizations of $\alpha \to \beta \to \alpha$ is undecidable (Section 3.1.2). Complementarily, it is shown that recognizing principal axiomatizations of $\alpha \to \beta \to \beta$ is decidable in linear time (Section 3.1.4). Second, Section 3.2 inspects combinatory logic with intersection types where instantiation is bounded by functional order $o \geq 1$ and functional arity a. Specifically, inhabitation in this type system is shown to be $(o+2)$-EXPTIME-complete (Section 3.2.1). Third, Section 3.3 inspects complexity of three problems associated with intersection type subtyping. In particular, an algorithm to decide intersection type subtyping in quadratic time is given in Section 3.3.1, fixed parameter intractability of intersection type matching wrt. the number of type variables [33] is outlines in Section 3.3.2, and an EXPTIME lower bound for intersection type unification [33] is described in Section 3.3.3.

The third part (Chapter 4) gives an overview over the implementation of an inhabitant search algorithm (CL)S-F# [32] for combinatory logic with intersection types with constructors. First, Section 4.1 outlines the theoretical foundation of (CL)S-F#. Second, Section 4.2 provides an overview over the inhabitant search interface (Section 4.2.5) and implementation (Section 4.2.6) of (CL)S-F#. Third, Section 4.3 contains an evaluation of (CL)S-F# in context of functional program synthesis (Section 4.3.1), object-oriented program synthesis (Section 4.3.2), and process synthesis (Section 4.3.3). Additionally, scalability of (CL)S-F# in deterministic (Section 4.3.4) and non-deterministic (Section 4.3.5) scenarios is inspected.

Chapter 2

Lambda-Calculus

The untyped λ-calculus can be seen as a Turing complete (by the Church-Turing Thesis) functional programming language. In fact, invented by Alonzo Church (Turing's doctoral advisor) in the 1930s, it predates Turing machines as a model of computation.

Since λ-terms (Definition 1) provide an elegant way to present functional programs, λ-calculus is well suited to study functional program synthesis.

Definition 1 (λ-Terms)**.**

$$M, N ::= x \mid (\lambda x.M) \mid (M\,N) \text{ where } x, y, z \text{ range over term variables}$$

Adopting the notation of [61], we omit superfluous parentheses, treat application as left-associative, and group consecutive abstractions, i.e. $\lambda xyz.x\,y\,z = (\lambda x.(\lambda y.(\lambda z.((x\,y)\,z))))$. We denote by $\mathrm{FV}(M)$ the set of all free variables in M and by $M[x := N]$ the substitution of the free variable x in M by N. We assume the standard Barendregt hygiene condition that all bound variables have fresh names and free variables are not captured by substitution.

We denote by $\to_\beta$ (β-*reduction*) the contextual closure of the relation $((\lambda x.M)\,N) \to_\beta M[x := N]$, and by $\twoheadrightarrow_\beta$ the reflexive transitive closure of $\to_\beta$. We call a λ-term of the shape $((\lambda x.M)\,N)$ a *redex* and say a λ-term N is in β-*normal form*, if N does not contain a redex.

All type systems considered in this chapter will have the λ-terms as their term language. We will use *type judgements* of the shape $\Gamma \vdash M : \tau$ consisting of the *type environment* Γ, the *subject* M, and the *assigned type* $\tau \in \mathbb{T}$ in the type system $\vdash$ for some type language $\mathbb{T}$. Type environments Γ are finite sets $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ of *type assumptions* $x : \sigma$ where x is a term variable and $\sigma \in \mathbb{T}$ is the corresponding assigned type. Additionally, we demand that x_i for $i = 1 \ldots n$ are distinct, thus allowing us to treat Γ as a function $\Gamma(x_i) = \sigma_i$ for $i = 1 \ldots n$ with domain $\mathrm{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and codomain $\mathrm{ran}(\Gamma) = \{\sigma_1, \ldots, \sigma_n\}$. We write $\mathcal{D} \triangleright \Gamma \vdash M : \tau$, if the type judgement $\Gamma \vdash M : \tau$ can be derived in the type system $\vdash$ by the *derivation* $\mathcal{D}$. In particular, $\mathcal{D}$ is a finite tree having $\Gamma \vdash M : \tau$ as its root and rules of $\vdash$ as edges. Whenever the specific derivation is immaterial, we write just $\Gamma \vdash M : \tau$ stating that there exists a corresponding derivation.

If $\Gamma \vdash M : \tau$ is derivable, we call M an *inhabitant* of τ in Γ and say τ is *inhabited* in Γ. In this work, the most important decision problem for a type system $\vdash$ is the *inhabitation* problem $\Gamma \vdash ? : \tau$ (Problem 1).

Problem 1 (Inhabitation, $\Gamma \vdash ? : \tau$)**.** *Given a type environment* Γ *and a type* τ*, is there a* λ*-term* M *such that* $\Gamma \vdash M : \tau$ *is derivable?*

In the Context of Synthesis

The inhabitation problem $\Gamma \vdash ? : \tau$ in typed λ-calculi can be considered as program synthesis from scratch. Types assume the role of specifications, where Γ represents a collection of existing components specified by their assigned types, and τ represents the desired features of the synthesized result. Therefore, algorithms that decide $\Gamma \vdash ? : \tau$ by explicitly constructing inhabitants are at the core of type-based synthesis.

There are several key properties for a type system $\vdash$ based on λ-calculus that will be of significance throughout this work: *subject reduction* (Definition 2), and *normalization* (Definition 3). Subject reduction describes type stability under computation. Normalization describes termination of typable terms.

Definition 2 (Subject Reduction)**.** *We say a type system* $\vdash$ *has the* subject reduction *property, if* $\Gamma \vdash M : \tau$ *and* $M \to_\beta N$ *imply* $\Gamma \vdash N : \tau$*.*

Definition 3 (Normalization)**.** *We say a type system* $\vdash$ *has the* normalization *property, if* $\Gamma \vdash M : \tau$ *implies* $\Gamma \vdash N : \tau$ *for some* β*-normal form* N *such that* $M \twoheadrightarrow_\beta N$*.*

If $\vdash$ has the subject reduction and normalization properties, then $\Gamma \vdash M : \tau$ implies $\Gamma \vdash N : \tau$ for some β-normal form N. This means that in order to decide $\Gamma \vdash ? : \tau$ it suffices to consider only β-normal forms N as potential subjects.

Chapter Outline In this chapter we inspect properties of three distinct type systems having λ-terms as their term language.

First, in Section 2.1 we consider the simply typed λ-calculus [45] where types coincide with propositional implicational formulae. We show that principal inhabitation (given a simple type, is there a λ-term in β-normal form having the given type as its principal type?) in this type system is PSPACE-complete [37].

Second, in Section 2.2 we consider the Coppo-Dezani-Venneri intersection type assignment system [24] (also known as the strict intersection type system [68]) where a λ-term can be assigned an intersection of two types, if it can be assigned both types individually. We outline a formalized (in the Coq proof assistant) proof that inhabitation in this type system is undecidable [40].

Third, in Section 2.3 we inspect bounded-dimensional fragments of the strict intersection type system [34, 38]. We show that even in bounded dimension inhabitation is undecidable [34]. Additionally, we outline the notion of bases that is related to the notion of principality in bounded dimension [39]. Since for any given λ-term the corresponding basis is unique, finite, and computable, it can be used for problems related to type inference.

2.1 Simply Typed Lambda-Calculus

The type language of *simple types* (Definition 4) coincides with propositional implicational formulae (the type constructor $\to$ corresponds to logical implication), and is the type language of the simply typed λ-calculus [45] (Definition 5).

Definition 4 (Simple Types, $\mathbb{T}^{\to}$)**.**

$$\mathbb{T}^{\to} \ni \sigma, \tau ::= \alpha \mid \sigma \to \tau \text{ where } \alpha, \beta, \gamma \text{ range over type variables } \mathbb{V}$$

Definition 5 (Simply Typed λ-calculus)**.**

$$\frac{}{\Gamma, x : \sigma \vdash_{\lambda(\to)} x : \sigma}\ (\text{Ax}) \qquad \frac{\Gamma, x : \sigma \vdash_{\lambda(\to)} M : \tau}{\Gamma \vdash_{\lambda(\to)} \lambda x.M : \sigma \to \tau}\ (\to\text{I})$$

$$\frac{\Gamma \vdash_{\lambda(\to)} M : \sigma \to \tau \qquad \Gamma \vdash_{\lambda(\to)} N : \sigma}{\Gamma \vdash_{\lambda(\to)} M\ N : \tau}\ (\to\text{E})$$

Under the Curry-Howard-Isomorphism [61], if $\Gamma \vdash_{\lambda(\to)} M : \tau$ is derivable, then the λ-term M corresponds to the proof of τ in the implicational fragment of intuitionistic propositional logic under assumptions $\text{ran}(\Gamma)$. Conversely, if τ can be proven under assumptions $\{\sigma_1, \ldots, \sigma_n\}$, then there exists a type environment Γ and a λ-term M such that $\text{ran}(\Gamma) = \{\sigma_1, \ldots, \sigma_n\}$ and $\Gamma \vdash_{\lambda(\to)} M : \tau$ is derivable. Therefore, inhabitation in the simply typed λ-calculus corresponds to provability in intuitionistic propositional implicational logic and is PSpace-complete [62]. Additionally, the simply typed λ-calculus is a pivotal element in the framework of Barendregt's λ-cube [2] constituting a common core of richer type systems.

In the Context of Synthesis

The simply typed λ-calculus can be considered a minimalistic monomorphic functional programming language. For example, it can be used to encode natural numbers by Church-numerals to cover the class of the so-called extended polynomials [59].

In the simply typed λ-calculus each typable λ-term M has a unique (up to type variable renaming) *principal type* (Definition 6) that under type substitutions captures any other type assignable to M [45, Theorem 3A6].

Definition 6 (Principal Type)**.** *We say that τ is a* principal type *of M, if $\emptyset \vdash_{\lambda(\to)} M : \tau$ and for all types σ such that $\emptyset \vdash_{\lambda(\to)} M : \sigma$ there exists a substitution S such that $S(\tau) = \sigma$.*

Let us extend the notion of inhabitants using the notion of principality in the following Definition 7.

Definition 7 (Normal Principal Inhabitant, [45, Definition 8A11])**.** *We say that a λ-term M in β-normal form is a* normal principal inhabitant *of τ, if τ is the principal type of M.*

In the Context of Synthesis

Principal inhabitants can be thought of as satisfying the given specification (their type) in the most accurate way (Example 1). Any non-principal inhabitant of a given type can be assigned a strictly more general type, namely its principal type.
Normal principal inhabitants are not guaranteed to exists (Example 2), even if the given type is inhabited. In this case, the given specification should be reassessed.

Example 1. *Both* $M_1 = \lambda x.x$ *and* $M_2 = \lambda xy.x\,y$ *are inhabitants of* $\tau = (\alpha \to \beta) \to \alpha \to \beta$*. However, only* M_2 *is the normal principal inhabitant of* τ*. The principal type of* M_1 *is* $\alpha \to \alpha$*.*

Example 2. *Although type* $\tau = \alpha \to \alpha \to \alpha$ *is inhabited by* $\lambda xy.x$*,* τ *has no corresponding normal principal inhabitant.*

Hindley posed the question whether normal principal inhabitants can be counted algorithmically [45, Problem 8D10 (i)], which was answered positively by Broda and Damas [14]. However, the question of complexity to decide *principal inhabitation* (Problem 2), i.e. whether the set of normal principal inhabitants is empty, remained open.

Problem 2 (Principal Inhabitation)**.** *Given a simple type* τ*, does there exist a* λ*-term* M *in* β*-normal form such that* τ *is the principal type of* M*?*

In the Context of Synthesis

Using simple types, we can specify some meaningful properties of corresponding normal principal inhabitants, e.g. that a λ-term is a signum function for Church-numerals (Example 3).

Example 3. *Let*

$$\begin{aligned}\tau_0 &= \alpha_0 \to \beta_0 \to \beta_0\\ \tau_1 &= (\alpha_1 \to \beta_1) \to (\alpha_1 \to \beta_1)\\ \sigma &= \big(\gamma \to (\alpha_2 \to \beta_2) \to (\alpha_2 \to \beta_2)\big) \to (\alpha_3 \to \beta_3 \to \beta_3) \to \delta\end{aligned}$$

and observe that

- τ_0 *has exactly the Church-numeral zero, i.e.* $\lambda f x.x$*, as its normal principal inhabitant*
- τ_1 *has exactly the Church-numeral one, i.e.* $\lambda f x.f\,x$*, as its normal principal inhabitant*
- *a unifier of* σ *and* τ_0 *maps* δ *to instances of* τ_0*, and a unifier of* σ *and* τ_1 *maps* δ *to instances of* τ_1
- *the normal principal inhabitant of* $\sigma \to \delta$ *is* $\lambda n.n\,(\lambda f x.f\,x)\,(\lambda f x.x)$*, i.e. the* signum *function for Church-numerals that maps zero to zero, and any non-zero number to one*

There are several aspects of principal inhabitation that sharply distinguish it from inhabitation.

First, if $\Gamma \vdash_{\lambda(\to)} M : \tau$ is derivable, then for some N there is a derivation of $\Gamma \vdash_{\lambda(\to)} N : \tau$ that does not contain judgements $\Gamma_1 \vdash_{\lambda(\to)} N_1 : \tau_1$ and $\Gamma_2 \vdash_{\lambda(\to)} N_2 : \tau_2$ such that $\Gamma_1 = \Gamma_2$, $\tau_1 = \tau_2$, and $N_1 \neq N_2$. For principal inhabitation this does not hold (Example 4).

Example 4. *Let $\tau = (\alpha \to \alpha) \to \alpha \to \alpha$. The normal principal inhabitants of τ are exactly the Church numerals greater or equal to two, i.e. $\lambda f.\lambda x.f\ (f\ x)$, $\lambda f.\lambda x.f\ (f\ (f\ x))$, ... The corresponding type derivations necessarily assign the type α to the terms $x, f\ x$ and $f\ (f\ x)$ in identical type environments.*

Second, term variables with identical types are interchangeable in the simple type system. However, this may violate principality (Example 5).

Example 5. *Let $\tau = (\alpha \to \alpha \to \alpha) \to \alpha \to \alpha \to \alpha$, $M = \lambda f x y.f\ (f\, x\, y)\, (f\, y\, x)$, $M_x = \lambda f x y.f\ (f\ x\ x)\ (f\ x\ x)$, and $M_y = \lambda f x y.f\ (f\ y\ y)\ (f\ y\ y)$. Each M, M_x and M_y is an inhabitant of τ. However, only M of the three is a normal principal inhabitant of τ.*

Section Outline The remainder of this section outlines the contribution showing that principal inhabitation is PSPACE-complete [37]. First, a calculus capturing necessary identifications in a type derivation is presented in Section 2.1.1 from which a characterization of principality is derived (Theorem 1). The upper bound (Section 2.1.2) is shown algorithmically and the lower bound (Section 2.1.3) is shown by adapting Urzyczyn's construction [65] to encompass principality.

Authorship Statement Since contributions presented in this section are part of joint work [37], this mandatory paragraph lists the following contributions attributed to the author.

- the subformula calculus (Definition 8)
- the characterization of principality (Theorem 1)
- principal inhabitation PSPACE-completeness (Algorithm INH, Lemma 7, and Lemma 8)

2.1.1 Subformula Calculus

To distinguish distinct subformula occurrences in a given type τ, we use *paths* π in the syntax tree of τ, which are defined as follows

$$\pi \in \{1, 2\}^*$$

We denote the empty sequence by ε. Since paths are character sequences, we use abbreviations such as $\pi 2^n$ for the path π followed by n twos. We access a subformula at path π in a given type τ by $\tau(\pi)$, defined as

$$\tau(\varepsilon) = \tau \qquad (\sigma \to \tau)(1\pi) = \sigma(\pi) \qquad (\sigma \to \tau)(2\pi) = \tau(\pi)$$

The above definition implies that we use types as functions from the set of their paths to their subformulae. In particular, $\mathrm{dom}(\tau)$ is the set of paths in τ and $\mathrm{ran}(\tau)$ is the set of subformulae in τ. Similarly to the simply typed λ-calculus, we define *path environments* $\Delta = \{x_1 : \pi_1, \ldots, x_n : \pi_n\}$, where $\mathrm{dom}(\Delta) = \{x_1, \ldots, x_n\}$. For a relation R on paths, the calculus $(\vdash_R)$ is given by rules $(\to_R \mathrm{I})$ and $(\to_R \mathrm{E})$ in the following Definition 8.

Definition 8 (Calculus $\vdash_R$)**.**

$$\frac{\Delta, x : \pi 1 \vdash_R M : \pi 2}{\Delta \vdash_R \lambda x.M : \pi} \,(\to_R \mathrm{I})$$

$$\frac{\pi 2^n \; R \; \pi' \qquad \Delta, x : \pi \vdash_R M_i : \pi 2^{i-1} 1 \text{ for } i = 1 \ldots n}{\Delta, x : \pi \vdash_R x \, M_1 \ldots M_n : \pi'} \,(\to_R \mathrm{E})$$

We call conditions of the form $\pi \; R \; \pi'$ *side conditions*. The above calculus $(\vdash_R)$, similarly to the calculus $\mathrm{TA}_{\mathrm{pln}}$ in [13], captures as side conditions identities imposed by the typed term. In contrast to $\mathrm{TA}_{\mathrm{pln}}$ it does not contain or require actual type information. Additionally, for any closed λ-term M in β-normal form there exists a relation R such that $\emptyset \vdash_R M : \varepsilon$ is derivable. In particular, this is true for terms that are not typable in the simply typed λ-calculus (Example 6).

Example 6. *Let $M = \lambda x.x\,x$ and $R = \{(12, 2), (1, 11)\}$. The term M does not contain free variables and is in β-normal form. We have*

$$\dfrac{\dfrac{12 \; R \; 2 \qquad \dfrac{1 \; R \; 11}{\{x : 1\} \vdash_R x : 11}\,(\to_R \mathrm{E})}{\{x : 1\} \vdash_R x\,x : 2}\,(\to_R \mathrm{E})}{\emptyset \vdash_R M : \varepsilon}\,(\to_R \mathrm{I})$$

However, M is not typable in the simply typed λ-calculus.

Intuitively, a $(\vdash_R)$-derivation contains as side conditions necessary equality constraints on subformulae that are required to type a given term M. We are interested in the least relation R such that $\emptyset \vdash_R M : \varepsilon$.

Definition 9 (R_M)**.** *Given a closed λ-term M in β-normal form, let R_M be a minimal (wrt. inclusion) equivalence relation such that $\emptyset \vdash_{R_M} M : \varepsilon$.*

Most importantly, R_M exists and is unique (Lemma 1). Given a relation R let us denote the *symmetric, transitive closure* of R by $R^{\leftrightarrow}$.

Lemma 1. *Given a closed λ-term M in β-normal form there exists exactly one minimal (wrt. inclusion) equivalence relation R_M such that $\emptyset \vdash_{R_M} M : \varepsilon$.*

Proof. Side conditions in $(\vdash_R)$ are uniquely defined by the concluding judgement. Therefore, let R' be the unique set of side conditions to derive $\emptyset \vdash_R M : \varepsilon$ for some R. Take $R_M = (R')^{\leftrightarrow}$. Since $(\vdash_R)$ is monotonous in R, we have $\emptyset \vdash_{R_M} M : \varepsilon$. Since for any R a derivation of $\emptyset \vdash_R M : \varepsilon$ would require $R' \subseteq R$, we have that R_M is minimal and unique. □

Note that in [37] we also take the reflexive closure. However, this is not necessary and potentially confusing later on. By a variance (even/odd number of ones) argument, we have $\pi R \pi'$ only if $\pi \neq \pi'$. Taking the symmetric transitive closure we obtain $\pi R^{\leftrightarrow} \pi$ and $\pi' R^{\leftrightarrow} \pi'$.

Example 7. *We have $R_{\lambda x.\lambda y.x} = \{(1, 22)\}^{\leftrightarrow} = \{(1, 22), (22, 1), (1, 1), (22, 22)\}$ and $R_{\lambda x.\lambda y.y} = \{(21, 22)\}^{\leftrightarrow} = \{(21, 22), (22, 21), (21, 21), (22, 22)\}$. The domain of $R_{\lambda x.\lambda y.x}$ (resp. $R_{\lambda x.\lambda y.y}$) does not contain the path 21 (resp. 1) which would correspond to the type of y (resp. x).*

Although in general we cannot identify term variables having the same types without changing side conditions, we may identify term variables in the path environment that are bound to same paths (Example 8).

Example 8. *Consider $M = \lambda f.f\ (\lambda x.f\ (\lambda y.y))$ and $M' = \lambda f.f\ (\lambda x.f\ (\lambda y.x))$. Both M and M' are normal principal inhabitants of $((\alpha \to \alpha) \to \alpha) \to \alpha$. Let $\Delta = \{f : 1, x : 111, y : 111\}$. The only difference between the derivation of $\emptyset \vdash_{R_M} M : \varepsilon$ and a derivation of $\emptyset \vdash_{R_{M'}} M' : \varepsilon$ is the leaf judgement. For the former it is $\Delta \vdash_{R_M} y : 112$ and for the latter $\Delta \vdash_{R_{M'}} x : 112$. Since the rest of the derivations is identical, x and y are interchangeable and we have $R_M = R_{M'}$.*

The equivalence relation R_M intuitively captures equality constraints on subformulae imposed by a given term M. Complementarily, given a type τ, we are interested in equality constraints on subformulae satisfied by τ. To capture such constraints we define the relation R_τ (Definition 10).

Definition 10 (R_τ)**.** *Given a type τ we define the relation R_τ on paths in* $\mathrm{dom}(\tau)$ *as $R_\tau = \{(\pi, \pi') \mid \pi \neq \pi' \wedge \tau(\pi) = \tau(\pi') \in \mathbb{V}\}^{\leftrightarrow}$.*

The condition $\pi \neq \pi'$ in the definition of R_τ excludes singular occurrences of type variables in τ from the domain of R_τ while the subsequent closure ensures reflexivity. This is illustrated in the following Example 9.

Example 9. *We have $R_{a\to b\to a} = \{(1, 22)\}^{\leftrightarrow} = \{(1, 22), (22, 1), (1, 1), (22, 22)\}$ and $R_{a\to b\to b} = \{(21, 22)\}^{\leftrightarrow} = \{(21, 22), (22, 21), (21, 21), (22, 22)\}$. Similarly to Example 7 the domain of $R_{a\to b\to a}$ (resp. $R_{a\to b\to b}$) does not contain the path 21 (resp. 1).*

Let us denote by $\mathrm{Long}(\tau)$ the set of so-called η-long β-normal inhabitants of τ [45, Definition 8A7] (cf. [37, Definition 6]) that capture maximally η-expanded inhabitants.

The relationship between principality and equality of R_M and R_τ (Example 7 and Example 9) is systematic. In particular, we have the following necessary condition (Lemma 2) for principal inhabitation.

Lemma 2 ([37, Lemma 27])**.** *Given a type τ let $M \in \mathrm{Long}(\tau)$. If τ is the principal type of M, then $R_\tau = R_M$.*

Unfortunately, the converse of the above Lemma 2 is not true as illustrated in the following Example 10.

Example 10. *Consider $M = \lambda x.\lambda y.x$ and $\tau = \alpha \to (\beta \to \gamma) \to \alpha$. We have $R_M = \{(1, 22)\}^{\leftrightarrow} = R_\tau$. However, τ has no normal principal inhabitant.*

One could follow the approach of [13] of marking necessary arrows in derivations (requiring further interplay between terms, derivations, and types) to close the gap exposed in the above Example 10. At first sight, taking arrow subformulae in derivations into account appears inevitable. Surprisingly, this is not the case. Certain types such as $\alpha \to (\beta \to \gamma) \to \alpha$ that have no normal principal inhabitants have a simple syntactic characterization. Strikingly, formulated as a necessary (and easy to verify) condition (Definition 11) we are able to close the mentioned gap without additional constraints on terms or derivations.

Definition 11 (Agreeable, [37, ($\star$)])**.** *We say τ is* agreeable*, if*

$$\forall \pi \in \mathrm{dom}(\tau).(\tau(\pi 2) \in \mathbb{V} \Rightarrow (\pi 2, \pi 2) \in R_\tau)$$

A type τ is agreeable, if τ has no subformula $\sigma \to \alpha$, where α occurs exactly once as a subformula of τ, regardless of whether the occurrence is positive or negative. This coincides with the first property in [15, Proposition 4.3] and is a necessary condition for principal inhabitation (Lemma 3).

Lemma 3. *If τ is not agreeable, then τ has no normal principal inhabitant.*

Proof. (Sketch) If τ is not agreeable, then there exists a path $\pi \in \mathrm{dom}(\tau)$ such that $\tau(\pi 2) \in \mathbb{V}$ and $(\pi 2, \pi 2) \notin R_\tau$. Assume τ has a normal principal inhabitant $M \in \mathrm{Long}(\tau)$ (cf. [45, Lemma 8A11.2]). By Lemma [37, Lemma 25] there exists a derivation $\mathcal{D} \rhd \emptyset \vdash_{R_\tau} M : \varepsilon$. Since $(\pi 2, \pi 2) \notin R_\tau$ the derivation $\mathcal{D}$ contains no judgement of the shape $\Delta \vdash_{R_\tau} N : \pi 2$ for some path environment Δ and term N. Therefore, replacing paths by corresponding subformulae in τ, there exists a derivation $\mathcal{D}' \rhd \emptyset \vdash_{\lambda(\to)} M : \tau$ such that α does not occur as an assigned type in $\mathcal{D}'$, where $\tau(\pi) = \sigma \to \alpha$ for some type σ. We can use the technique of subformula filtration [37, Definition 11] to replace the type $\sigma \to \alpha$ by a single type variable [37, Lemma 12]. As a result the type τ is not the principal type of M [37, Lemma 14], which is a contradiction. □

In the Context of Synthesis

If τ is not agreeable, then it specifies a functional property that cannot be satisfied by any λ-term in β-normal form.

Considering only agreeable types, we can formulate a sufficient condition (Lemma 4) for principal inhabitation.

Lemma 4. *Given an agreeable type τ let $M \in \mathrm{Long}(\tau)$. If $R_\tau = R_M$, then τ is the principal type of M.*

Proof. Assume M has a strictly more general principal type τ'. Fix the substitution S such that $S(\tau') = \tau$. Since generalization does not affect η-longness we have $M \in \mathrm{Long}(\tau')$. Therefore, by Lemma 2 we have $R_M = R_{\tau'}$. We show that $R_\tau \neq R_{\tau'}$, therefore $R_\tau \neq R_M$.

Case $S : \mathbb{V} \to \mathbb{V}$: There exist π, π' such that $\tau(\pi) = \tau(\pi') \in \mathbb{V}$ and $\tau'(\pi) \neq \tau'(\pi')$. Therefore, $(\pi, \pi') \in R_\tau$ but $(\pi, \pi') \notin R_{\tau'}$.

Case $S(\alpha) = \sigma_1 \to \ldots \to \sigma_n \to \beta$ for some $n > 0$ and $\alpha \in \mathrm{ran}(\tau') \cap \mathbb{V}$: Fix any path $\pi \in \mathrm{dom}(\tau')$ such that $\tau'(\pi) = \alpha$. Since $\tau(\pi 2^n) = \beta$, $n > 0$, and τ is agreeable, we have $(\pi 2^n, \pi 2^n) \in R_\tau$. However, $\tau'(\pi 2^n)$ is undefined, therefore $(\pi 2^n, \pi 2^n) \notin R_{\tau'}$. □

In sum, the equality $R_M = R_\tau$ characterizes principality in the sense of the following Theorem 1.

Theorem 1 ([37, Theorem 32])**.** *Given an agreeable type τ and a λ-term M such that $M \in \mathrm{Long}(\tau)$ we have that τ is the principal type of M iff $R_M = R_\tau$.*

Proof. **"$\Longrightarrow$"** by Lemma 2. **"$\Longleftarrow$"** by Lemma 4. □

Bearing resemblance to the characterization in [13, Proposition 17], the above characterization in Theorem 1 has two benefits. First, it does not require marking of arrows in derivations. Second, it is factored into R_M (uniquely defined by M) and R_τ (uniquely defined by τ). Since the size of R_τ is polynomial in the size of τ, we only require polynomial space for principal inhabitation in the following Section 2.1.2.

In the Context of Synthesis

The above Theorem 1 shows that a long normal principal inhabitant M of τ satisfies functional dependencies collected in R_M which are exactly those collected in the corresponding specification R_τ.

2.1.2 Principal Inhabitation Upper Bound

In this section we present a polynomial space algorithm to decide principal inhabitation [37, Section 5]. Given a type τ, the idea behind the following Algorithm INH to decide principal inhabitation (Problem 2) is as follows. Start by verifying that τ is agreeable. Continue with the auxiliary Algorithm AUX to construct a relation R corresponding to R_M for some long normal inhabitant M (which is not constructed explicitly). Last, verify that $R_M = R_\tau$.

Algorithm 1 Algorithm INH deciding existence of normal principal inhabitants

1: *Input:* simple type τ
2: *Output:* **accept** iff there exists a normal principal inhabitant of τ
3: **if** $\neg\Big(\forall \pi \in \mathrm{dom}(\tau).(\tau(\pi 2) \in \mathbb{V} \Rightarrow (\pi 2, \pi 2) \in R_\tau)\Big)$ **then**
4: **fail**
5: **end if**
6: $R := \mathsf{AUX}(\tau, \emptyset, \varepsilon, \emptyset)$
7: **if** $R = R_\tau$ **then**
8: **accept**
9: **else**
10: **fail**
11: **end if**

Algorithm 2 Non-deterministic Algorithm AUX

1: *Input:* simple type τ, set of paths P, path π, relation on paths R
2: *Output:* updated relation on paths R
3: **if** $\tau(\pi) = \sigma \to \tau$ **then**
4: **return** $\mathsf{AUX}(\tau, P \cup \{\pi 1\}, \pi 2, R)$
5: **else if** $\tau(\pi) = \alpha$ for some $\alpha \in \mathbb{V}$ **then**
6: **choose** $\pi' \in P$ such that $\tau(\pi' 2^n) = \alpha$ for some $n \geq 0$
7: $R := (R \cup \{(\pi' 2^n, \pi)\})^{\Leftrightarrow}$
8: **for** $i = 1$ **to** n **do**
9: $R := \mathsf{AUX}(\tau, P, \pi' 2^{i-1} 1, R)$
10: **end for**
11: **end if**
12: **return** R

Let us illustrate a run of the Algorithm INH (including recursive calls to Algorithm AUX) in the following Example 11.

Example 11. *Let $\tau = ((\alpha \to \alpha) \to \alpha) \to \alpha$ and consider $\mathsf{INH}(\tau)$. Since τ is agreeable, the condition in line 3 does not trigger a failure.*

- *Proceed with $\mathsf{AUX}(\tau, \emptyset, \varepsilon, \emptyset)$, which corresponds to inhabitant search of $\tau(\varepsilon) = \tau$.*
- *Since $\tau(\varepsilon)$ is an arrow type, take the first branch (line 4). This induces a potential inhabitant of the shape $\lambda f.N$ for a fresh f and some λ-term N. Proceed with $\mathsf{AUX}(\tau, \{1\}, 2, \emptyset)$, which corresponds to the search for N of type $\tau(2) = a$ in the type environment $\{f : \tau(1) = (\alpha \to \alpha) \to \alpha\}$.*

- *Since $\tau(2) = \alpha = \tau(12)$, take the second branch (lines 6–10) choosing the path $1 \in P$. This induces $N = f\ L$ for some λ-term L.*
  Proceed with $\mathsf{AUX}(\tau, \{1\}, 11, \{(12,2)\}^{\Leftrightarrow})$, searching for L of type $\tau(11) = a \to a$ in the type environment $\{f : \tau(1) = (\alpha \to \alpha) \to \alpha\}$.

- *Since $\tau(11) = \alpha \to \alpha$ is an arrow type, take the first branch, i.e. $L = \lambda x.L'$ for a fresh x and some λ-term L'.*
  Proceed with $\mathsf{AUX}(\tau, \{1, 111\}, 112, \{(12,2)\}^{\Leftrightarrow})$, searching for L' of type $\tau(112) = \alpha$ in the type environment
  $\{f : \tau(1) = (\alpha \to \alpha) \to \alpha, x : \tau(111) = \alpha\}$.

- *Since $\tau(112) = \alpha$, take the second branch. There are two options. The first option is to choose the path 111, since $\tau(112) = \tau(111)$. In this case, AUX would return control to INH with the result $R = \{(12,2),(111,112)\}^{\Leftrightarrow}$ and INH would fail. The corresponding run of INH would induce the inhabitant $\lambda f.f\ (\lambda x.x)$, which is not a normal principal inhabitant of τ. The second option is to choose the path 1 since $\tau(112) = \tau(12)$ and proceed with $\mathsf{AUX}(\tau, \{1, 111\}, 11, \{(12,2),(12,112)\}^{\Leftrightarrow})$. Choose the second option.*

- *Again, $\tau(11) = \alpha \to \alpha$ is an arrow type, take the first branch and proceed with $\mathsf{AUX}(\tau, \{1, 111\}, 112, \{(12,2),(12,112)\}^{\Leftrightarrow})$.*

- *Again, $\tau(112) = \alpha$, take the second branch, choosing the path 111. After AUX returns $R = \{(12,2),(12,112),(111,112)\}^{\Leftrightarrow}$ to INH, INH accepts. The corresponding run of INH induces the normal principal inhabitant $\lambda f.f\ (\lambda x.f\ (\lambda y.x))$ (Example 8) of τ.*

The normal principal inhabitant of τ is not constructed explicitly by Algorithm INH due to polynomial space restriction. However, Algorithm AUX can be easily modified to compute the inhabitant corresponding to the constructed relation R.

In the Context of Synthesis

Algorithm INH verifies that the given specification τ is agreeable and uses Algorithm AUX to satisfy the required specification given by τ (specifically, R_τ). The recursively constructed relation R collects all functional dependencies satisfied by the term structure of the potential inhabitant.

Lemma 5 (Soundness of INH)**.** *Given a type τ, if Algorithm 1 accepts, then there exists a normal principal inhabitant of τ.*

Proof. A successful run of Algorithm 1 induces a derivation of $\emptyset \vdash_{R_M} M : \varepsilon$ for some λ-term M. In particular, line 4 in Algorithm AUX induces a λ-abstraction and lines 6–10 in Algorithm AUX induce an application with head variable of type $\tau(\pi')$ and n arguments. It suffices to take the variable that is bound to π' and in M is abstracted outermost. Line 3 in in Algorithm INH ensures that τ is agreeable and line 7 ensures that $R_M = R_\tau$. By Theorem 1 the term M is a normal principal inhabitant of τ. □

Lemma 6 (Completeness of INH)**.** *Given a type τ, if there exists a normal principal inhabitant of τ, then there exists an accepting run of Algorithm 1 requiring at most polynomial space in the size of τ.*

Proof. Assume that τ has a normal principal inhabitant M. By Theorem 1 we have that τ is agreeable and there exists a normal principal inhabitant $M' \in \mathrm{Long}(\tau)$ such that $\mathcal{D} \triangleright \emptyset \vdash_{R_{M'}} M' : \varepsilon$ and $R_{M'} = R_\tau$. By induction on $\mathcal{D}$ there exists an accepting run $\mathcal{R}$ of Algorithm INH such that for each judgement $\Delta \vdash_{R_{M'}} L : \pi$ in $\mathcal{D}$ the run $\mathcal{R}$ invokes $\mathsf{AUX}(\tau, \mathrm{ran}(\Delta), \pi, R)$ where $R \subseteq R_{M'}$. Therefore, for each side condition $\pi' \; R_{M'} \; \pi''$ in $\mathcal{D}$ the corresponding invocation of AUX in line 7 ensures $\pi' \; R \; \pi''$. Overall, by Theorem 1 we have $R_\tau = R_{M'} = R$ and INH accepts.

Polynomial Space Requirement Arguments τ, $P \subseteq \mathrm{dom}(\tau)$, $\pi \in \mathrm{dom}(\tau)$, and $R \subseteq \mathrm{dom}(\tau)^2$ are polynomial in the size of τ. Since the above run $\mathcal{R}$ is accepting and there are no side-effects, there exists an accepting run $\mathcal{R}'$ that has no invocations of AUX with identical parameters along the recursive branches of AUX. Since P and R are non-decreasing along the recursive branches of AUX, the invocation stack of AUX in $\mathcal{R}'$ is of polynomial depth in size of τ. □

Lemma 7 ([37, Lemma 38])**.** *Principal inhabitation (Problem 2) is in* PSPACE.

Proof. By Lemma 5, Lemma 6 and the identity PSPACE=NPSPACE. □

2.1.3 Principal Inhabitation Lower Bound

In this section we illustrate a PSPACE lower bound for principal inhabitation [37, Section 6]. Unfortunately, the standard reduction [65] from quantified Boolean formulae to inhabitation in the simply typed λ-calculus does not carry over immediately (Example 12).

Example 12. *Consider the formula $\varphi = \exists p.\psi$, where $\psi = p \vee \neg p$. By the construction in [65] the formula φ is true iff the type $\tau = ((\alpha_p \to \alpha_\psi) \to \alpha_\varphi) \to ((\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi) \to (\alpha_p \to \alpha_\psi) \to (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi$ is inhabited in the simply typed λ-calculus. The only long normal inhabitants of τ are $\lambda x_1.\lambda x_2.\lambda y_1.\lambda y_2.x_1 \; (\lambda z.y_1 \; z)$ and $\lambda x_1.\lambda x_2.\lambda y_1.\lambda y_2.x_2 \; (\lambda z.y_2 \; z)$ for both of which τ is not principal. In fact, there is no normal principal inhabitant of τ.*

The inherent issue with the standard approach is that existential quantifiers and disjunctions may introduce unnecessary subformulae. This issue is solved by introducing additional subformulae that do not affect inhabitation.

Lemma 8. *[37, Lemma 42] Principal inhabitation (Problem 2) is* PSPACE*-hard.*

Given a type τ, let us illustrate the construction of τ^* [37, Section 6] such that τ is inhabited iff τ^* is principally inhabited. We reexamine in the following Example 13 the principally not inhabited type from the previous Example 12.

Example 13. *Let*

$$\begin{aligned}
\tau &= ((\alpha_p \to \alpha_\psi) \to \alpha_\varphi) \to ((\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi) \to \\
&\quad (\alpha_p \to \alpha_\psi) \to (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi \\
\sigma_x^y &= \alpha_x \to \alpha_y \to \alpha_y \text{ where } x, y \in \{p, \neg p, \psi, \varphi\} \\
\tau^* &= ((\alpha_p \to \alpha_{\neg_p} \to \alpha_\psi \to \alpha_\varphi \to \alpha_\varphi) \to \alpha_\varphi \to \alpha_\varphi) \to \\
&\quad \sigma_p^\varphi \to \sigma_{\neg p}^\varphi \to \sigma_\psi^\varphi \to \sigma_\varphi^p \to \sigma_\varphi^{\neg p} \to \sigma_\varphi^\psi \to \sigma_\varphi^\varphi \to (\alpha_\varphi \to \alpha_\varphi) \to \tau
\end{aligned}$$

The type τ^* is principally inhabited by the following λ-term M

$$
\begin{aligned}
M &= \lambda z x_p^\varphi x_{\neg p}^\varphi x_\psi^\varphi x_\varphi^p x_\varphi^{\neg p} x_\varphi^\psi x_\varphi^\varphi x w_1 w_2 w_3 w_4 . x\,(z\,F\,(x\,N)) \text{ where}\\
N &= w_1\,(\lambda z . w_3\ z) \text{ such that } \{w_1 : (\alpha_p \to \alpha_\psi) \to \alpha_\varphi, w_2 : (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi,\\
&\quad\ w_3 : \alpha_p \to \alpha_\psi, w_4 : \alpha_{\neg p} \to \alpha_\psi\} \vdash N : \alpha_\varphi \text{ is derivable}\\
F &= \lambda y_p y_{\neg p} y_\psi y_\varphi . x_\varphi^\varphi\,F_1\,(x_\varphi^\varphi\,F_p\,(x_\varphi^\varphi\,F_{\neg p}\,(x_\varphi^\varphi\,F_\psi\,(x_\varphi^\varphi\,G_1\,(x_\varphi^\varphi\,G_2\,G_{3,4})))))\\
F_1 &= x\,(x\,(x_\varphi^\varphi\,(x\,y_\varphi)\,(x\,y_\varphi))) \text{ ensuring } x : \alpha_\varphi \to \alpha_\varphi, x_\varphi^\varphi : \sigma_\varphi^\varphi, y_\varphi : \alpha_\varphi\\
F_p &= x_p^\varphi\,(x_\varphi^p\,y_\varphi\,(x_\varphi^p\,y_\varphi\,y_p))\,y_\varphi \text{ ensuring } x_p^\varphi : \sigma_p^\varphi, x_\varphi^p : \sigma_\varphi^p, y_p : \alpha_p\\
F_{\neg p} &= x_{\neg p}^\varphi\,(x_\varphi^{\neg p}\,y_\varphi\,(x_\varphi^{\neg p}\,y_\varphi\,y_{\neg p}))\,y_\varphi \text{ ensuring } x_{\neg p}^\varphi : \sigma_{\neg p}^\varphi, x_\varphi^{\neg p} : \sigma_\varphi^{\neg p}, y_{\neg p} : \alpha_{\neg p}\\
F_\psi &= x_\psi^\varphi\,(x_\varphi^\psi\,y_\varphi\,(x_\varphi^\psi\,y_\varphi\,y_\psi))\,y_\varphi \text{ ensuring } x_\psi^\varphi : \sigma_\psi^\varphi, x_\varphi^\psi : \sigma_\varphi^\psi, y_\psi : \alpha_\psi\\
G_1 &= w_1\,(\lambda r . x_\varphi^\psi\,(x_p^\varphi\,r\,y_\varphi)\,y_\psi) \text{ ensuring } w_1 : (\alpha_p \to \alpha_\psi) \to \alpha_\varphi\\
G_2 &= w_2\,(\lambda r . x_\varphi^\psi\,(x_{\neg p}^\varphi\,r\,y_\varphi)\,y_\psi) \text{ ensuring } w_2 : (\alpha_{\neg p} \to \alpha_\psi) \to \alpha_\varphi\\
G_{3,4} &= x_\psi^\varphi\,(w_3\,y_p)\,(x_\psi^\varphi\,(w_4\,y_{\neg p})\,y_\varphi) \text{ ensuring } w_3 : \alpha_p \to \alpha_\psi, w_4 : \alpha_{\neg p} \to \alpha_\psi
\end{aligned}
$$

In the above construction the term F is used solely to ensure that all equalities on atomic subformulae (cf. R_{τ^*}) are established (cf. R_M). In particular, $G_1, G_2, G_{3,4}$ establish the structure of the original premises of τ, including those which are not used to type N. Since additional premises in τ^* are intuitionistic theorems, if τ^* is inhabited, then so is τ.

Arguably, in the above example τ is of low functional order, and the illustrated technique might not work for arbitrary types. However, the construction in [65] shows that in order to reduce satisfiability of quantified Boolean formulae to simple type inhabitation, types of shape similar to Example 13 are sufficient [37, Problem 2].

Concluding Remarks

We believe that the described characterization of principality (Theorem 1) is of general interest because it separates conditions satisfied by terms from conditions required by types. Additionally, the described subformula calculus is not restricted to simply typed terms and can be used to inspect the structure of any λ-term in β-normal form. Therefore, there may be a deeper connection to intersection type systems (for an overview see [68]).

The lower bound construction implying PSpace-hardness of principal inhabitation may be useful to inspect principal inhabitation in the simply typed λI-calculus for which inhabitation is 2-ExpTime-complete [57].

Using quantified Boolean formulae allows us to bound the functional order of types that suffice for the lower bound construction. Possibly, the lower bound construction can be adapted to reduce simple type inhabitation to principal inhabitation directly.

2.2 Strict Intersection Type System

The Coppo-Dezani type assignment system [23], initiating the line of work known today as the intersection type discipline (for an overview see [4, Part 3] and [68]), can be considered the first intersection type system. Most importantly, in the Coppo-Dezani type assignment system each λ-term in β-normal form is typable, and typability characterizes exactly the strongly normalizing terms in the λI-calculus.

The Coppo-Dezani-Venneri type assignment system [25] (also presented as the strict intersection type system [68]) extends the Coppo-Dezani type assignment system by the universal type ω. Since any λ-term is typable by ω, this type system is closed under β-expansion. This allows to assign exactly the same types to β-equal λ-terms and inspect normalization properties (for an overview see [42]).

Later, the Barendregt-Coppo-Dezani type assignment system [3] (commonly referred to as *the* intersection type system) simplified type syntax, added an intersection type subtyping relation, and provided a natural filter model.

Our presentation follows the original intuition of Coppo and Dezani [23]. In particular, the type language of *intersection types* (Definition 12) extends simple types by allowing *sets* of types as premises of the arrow type constructor. We use the naming conventions introduced by van Bakel [68] distinguishing *strict types* and *intersection types*.

Definition 12 (Strict Types, $\mathbb{T}^{\cap}_{!}$, and Intersection Types, $\mathbb{T}^{\cap}_{\{!\}}$)**.**

$$\begin{aligned}
&\mathbb{T}^{\cap}_{!} \ni \phi, \psi ::= \alpha \mid \sigma \to \phi && (\textit{strict types})\\
&\mathbb{T}^{\cap}_{\{!\}} \ni \sigma, \tau ::= \{\phi_1, \ldots, \phi_n\} \textit{ where } n \geq 0 && (\textit{intersection types})\\
&\qquad \textit{where } \alpha \textit{ ranges over type variables } \mathbb{V}
\end{aligned}$$

We abbreviate the empty intersection type by $\omega \in \mathbb{T}^{\cap}_{\{!\}}$. The above stratification of intersection types, compared to Barendregt-Coppo-Dezani presentation [3], naturally *normalizes* the presentation in the sense of [48].

Historically (for an overview see [68]) an intersection type $\{\phi_1, \ldots, \phi_n\}$ is written as $\phi_1 \cap \ldots \cap \phi_n$ together with a remark that $\cap$ is treated as an associative, commutative, and idempotent type constructor. Arguably, the $\cap$ type constructor is more natural considering semantics of intersection types (especially in presence of union types). However, in this work we use intersection types in the original sense of Coppo and Dezani [23] as sets. Therefore, we use a similar presentation relying on inherent properties of sets, disambiguating strict types and singleton intersection types.

Modern presentations [17] also tend to distinguish so-called *set types* (our presentation above) and *multiset types* (premises of $\to$ are multisets, i.e. idempotency of $\cap$ is dropped).

Having the type ω at our disposal, we consider type environments to be total functions from term variables into intersection types that are almost everywhere ω except their finite domain, i.e. $x \in \mathrm{dom}(\Gamma)$ iff $\Gamma(x) \neq \omega$. This implies $x \notin \mathrm{dom}(\Gamma)$ iff $\Gamma = \Gamma, x : \omega$.

The rules of the *strict intersection type system* [68, Definition 5.1] (corresponding to the Coppo-Dezani-Venneri type assignment system [25]) are given in the following Definition 13.

Definition 13 (Strict Intersection Type System [68, Definition 5.1], $\vdash_{\lambda(\cap)}$)**.**

$$\frac{\Gamma, x : \sigma \vdash_{\lambda(\cap)} M : \phi}{\Gamma \vdash_{\lambda(\cap)} \lambda x.M : \sigma \to \phi}\ (\to\text{I}) \qquad \frac{\Gamma \vdash_{\lambda(\cap)} M : \sigma \to \phi \qquad \Gamma \vdash_{\lambda(\cap)} N : \sigma}{\Gamma \vdash_{\lambda(\cap)} M\ N : \phi}\ (\to\text{E})$$

$$\frac{\phi \in \sigma}{\Gamma, x : \sigma \vdash_{\lambda(\cap)} x : \phi}\ (\cap\text{E}) \qquad \frac{\Gamma \vdash_{\lambda(\cap)} M : \phi_1 \qquad \dots \qquad \Gamma \vdash_{\lambda(\cap)} M : \phi_n}{\Gamma \vdash_{\lambda(\cap)} M : \{\phi_1, \dots, \phi_n\}}\ (\cap\text{I})$$

Example 14. *While the λ-term $\lambda x.x\,x$ is not typable in the simply typed λ-calculus, it can be assigned the type $\{\{\alpha\} \to \beta, \alpha\} \to \beta$ in the strict intersection type system as follows. Let $\Gamma = \{x : \{\{\alpha\} \to \beta, \alpha\}\}$. We have*

$$\dfrac{\dfrac{\dfrac{\{\alpha\} \to \beta \in \{\{\alpha\} \to \beta, \alpha\}}{\Gamma \vdash_{\lambda(\cap)} x : \{\alpha\} \to \beta}\ (\cap\text{E}) \qquad \dfrac{\dfrac{\alpha \in \{\{\alpha\} \to \beta, \alpha\}}{\Gamma \vdash_{\lambda(\cap)} x : \alpha}\ (\cap\text{E})}{\Gamma \vdash_{\lambda(\cap)} x : \{\alpha\}}\ (\cap\text{I})}{\Gamma \vdash_{\lambda(\cap)} x\,x : \beta}\ (\to\text{E})}{\emptyset \vdash_{\lambda(\cap)} \lambda x.x\,x : \{\{\alpha\} \to \beta, \alpha\} \to \beta}\ (\to\text{I})$$

In the Context of Synthesis

Intersection types constitute a particularly interesting specification language for type-based code synthesis that can express higher-order tabular functional dependencies [54]. The universal type ω is useful to describe underspecified pieces of code.

Both [66] and [36] inspect inhabitation in intersection type systems without the type ω. Unfortunately, the strict intersection type system is not a conservative extension of the Coppo-Dezani type assignment system (Example 15). Therefore, it requires some work, in particular regarding a formalization in a proof assistant, to adapt existing results.

Example 15. *Let $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ and $M = (\lambda xy.y)\ \Omega$. We have that $\emptyset \vdash_{\lambda(\cap)} M : \alpha \to \alpha$. However, since Ω is not typable in the Coppo-Dezani type assignment system, neither is M.*

In this section we adapt the simplified proof from [40] to the strict system. Most importantly, we incorporate the type ω into the formalization [31] in the Coq[1] proof assistant of soundness and completeness of the underlying reduction.

In [31] the strict intersection type system (Definition 13) is formalized in `StrictDerivation.v` as the mutual inductive type

```
Inductive strict_derivation : environment → term → formula → Prop
with strict_derivation_cap : environment → term → formula' → Prop
```

having as type constructors exactly the above typing rules. Since the rules (∩E), (→E), and (→I) assign strict types (`formula`) and the rule (∩I) assigns intersection types (`formula'`), the above formalization is split accordingly into two inductive types.

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

The formalization uses lists as type environments instead of sets, which is immaterial because the typing system enjoys weakening (and permutation) of type environments as shown in

```
Theorem strict_weakening : ∀ (Γ Γ': environment) (M: term) (t: formula),
  strict_derivation Γ M t → well_formed_environment Γ' →
    (∀ (p : label * formula'),  In p Γ → In p Γ') →
    strict_derivation Γ' M t.
```

After Leivant [49] we define the *rank* of a type (rules are given in descending priority).

Definition 14 (Rank)**.**

$$\begin{aligned} \operatorname{rank}(\tau) &= 0 \text{ if } \tau \text{ is a simple type,} \\ &\quad \text{and otherwise} \\ \operatorname{rank}(\sigma \to \phi) &= \max\{1 + \operatorname{rank}(\sigma), \operatorname{rank}(\phi)\} \\ \operatorname{rank}(\{\phi_1, \ldots, \phi_n\}) &= \max\{1, \operatorname{rank}(\phi_1), \ldots, \operatorname{rank}(\phi_n)\} \end{aligned}$$

Let us focus our attention on the following intersection type inhabitation problem (Problem 3).

Problem 3 (Intersection Type Inhabitation, $\Gamma \vdash_{\lambda(\cap)} ? : \tau$)**.** *Given a type environment Γ and an intersection type τ, is there a λ-term M such that the judgement $\Gamma \vdash_{\lambda(\cap)} M : \tau$ is derivable?*

A thorough analysis of intersection type inhabitation showing its undecidability at rank 3 was done by Urzyczyn [66] and later simplified [36] and formalized [40] in the Coq proof assistant. This section outlines a proof of undecidability of inhabitation in the Coppo-Dezani-Venneri type assignment system. The proof is by reduction from a word problem for simple semi-Thue systems. Additionally, we outline the formalization [31] of soundness and completeness of the reduction in the Coq proof assistant considering the universal type ω.

Section Outline The remainder of this section outlines the proof and formalization of undecidability of intersection type inhabitation based on [40]. Section 2.2.1 recapitulates the notion of simple semi-Thue systems along with a particular undecidable word problem for such systems. This word problem is reduced to intersection type inhabitation in Section 2.2.2 referring to formalization of soundness and completeness in the Coq proof assistant.

Authorship Statement Since contributions presented in this section are part of joint work [40], this mandatory paragraph lists the following contributions attributed to the author.

- reduction from simple semi-Thue rewriting to intersection type inhabitation (Section 2.2.2) based on [66] including soundness (Theorem 3) and completeness (Theorem 4) proofs
- formalization of soundness and completeness results in the Coq proof assistant

2.2.1 Simple Semi-Thue Systems

In this section we recapitulate the notion of *simple semi-Thue systems* which are closely related to Turing machines, albeit much simpler to handle. Additionally, we refer to a particular undecidable word problem in such systems, which will be reduced to intersection type inhabitation in Section 2.2.2 (a similar problem is used in [66] as a starting point).

Definition 15 (Semi-Thue System)**.** *A* semi-Thue system $\mathcal{R}$ *over an alphabet* Σ *is a finite set of rules of shape* $v \to w$ *where* $v, w \in \Sigma^*$*. The relation* $\to$ *is extended to* $\to_{\mathcal{R}} \subseteq \Sigma^* \times \Sigma^*$ *by* $tvu \to_{\mathcal{R}} twu$*, if* $v \to w$ *where* $t, u \in \Sigma^*$*. The relation* $\twoheadrightarrow_{\mathcal{R}}$ *is the reflexive, transitive closure of* $\to_{\mathcal{R}}$*.*

Definition 16 (Simple Semi-Thue System)**.** *A semi-Thue system* $\mathcal{R}$ *over an alphabet* Σ*, where each rule has the form* $ab \to cd$ *for some* $a, b, c, d \in \Sigma$*, is called a* simple semi-Thue system*.*

The simple semi-Thue System rewriting relation is formalized in `SSTS.v` as the inductive type

```
Inductive rewrites_to (rs : list rule) : list nat → list nat → Prop
```

where words are represented by lists of natural numbers. Let us abbreviate $a \dots a \in \Sigma^n$ by a^n.

Problem 4 ($0^? \twoheadrightarrow_{\mathcal{R}} 1^?$)**.** *Given a simple semi-Thue system* $\mathcal{R}$ *over an alphabet* Σ *such that* $0, 1 \in \Sigma$*, is there an* $n \in \mathbb{N}$ *with* $n > 0$ *such that* $0^n \twoheadrightarrow_{\mathcal{R}} 1^n$*?*

Accordingly, the above decision problem is formalized in `SSTS.v` as

```
∃ (m : nat), rewrites_to rs (repeat 0 (1+m)) (repeat 1 (1+m))
```

where `repeat a (1+m)` is a list of `a`'s of length `1+m`.

Example 16. *Let* $\mathcal{R} = \{00 \to 11, 11 \to 00, 01 \to 10, 10 \to 01\}$ *be a simple semi-Thue system over the alphabet* $\Sigma = \{0, 1\}$*. Since* $\mathcal{R}$ *preserves binary parity we have* $0^n \twoheadrightarrow_{\mathcal{R}} 1^n$ *iff* n *is even.*

Lemma 9 ([40, Lemma 3.3])**.** $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$ *(Problem 4) is undecidable.*

Intuitively, simple semi-Thue systems are expressive enough to represent Turing machine computation. The main observation is that a Turing machine configuration (q, i, t) (where q is the current state, i is the current head position, and t is the current tape) can be represented by a word $t_1 \dots t_{i-1} \langle q, t_i \rangle t_{i+1} \dots t_{|t|}$ over an extended alphabet including pairs of states and symbols. Accordingly, a transition would affect exactly two neighboring symbols.

2.2.2 Reduction from Rewriting to Inhabitation

In this section we reduce simple semi-Thue system rewriting $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$ (Problem 4) to intersection type inhabitation $\Gamma \vdash_{\lambda(\cap)} ? : \tau$ (Problem 3) showing undecidability of the latter (Theorem 2).

Theorem 2. *Intersection type inhabitation (Problem 3) is undecidable.*

For the remainder of this section, let us fix a simple semi-Thue system $\mathcal{R}$ over the alphabet $\Sigma \subseteq \mathbb{V}$ such that $0, 1 \in \Sigma$ and $\blacktriangle, \bullet, *, \#, \$, l, r \in \mathbb{V} \setminus \Sigma$.

We define the following types of rank at most 2 and the type environment $\Gamma_{\blacktriangle}$, providing some intuition for their intended use.

$$\sigma_{\blacktriangle} = \{\{\#, \$\} \to \blacktriangle\}$$
initializes the first word to $\#\$$
$$\sigma_{*} = \Big\{\big\{\{\bullet\} \to *\big\} \to *, \big\{\{l\} \to *\big\} \to \#, \big\{\{r\} \to \#, \{\bullet\} \to \$\big\} \to \$\Big\}$$
expands the word $* \ldots * \#\$$ to $* \ldots * *\#\$$
$$\sigma_{0} = \big\{\{0\} \to *, \{0\} \to \#, \{1\} \to \$\big\}$$
ends the expansion phase by rewriting $* \ldots * \#\$$ to $0 \ldots 001$
$$\sigma_{1} = \{1\}$$
accepts the word $1 \ldots 1$
$$\sigma_{ab \to cd} = \big\{\{l\} \to \{c\} \to a, \{r\} \to \{d\} \to b\big\} \cup \big\{\{\bullet\} \to \{e\} \to e \mid e \in \Sigma\big\}$$
rewrites the word $vabw$ to $vcdw$
$$\Gamma_{\blacktriangle} = \{x_{\blacktriangle} : \sigma_{\blacktriangle}, x_{*} : \sigma_{*}, x_0 : \sigma_0, x_1 : \sigma_1\} \cup \{x_{ab \to cd} : \sigma_{ab \to cd} \mid ab \to cd \in \mathcal{R}\}$$

The above definitions are formalized in `Encoding.v`. In particular, $\Gamma_{\blacktriangle}$ corresponds to the environment `Γ_init ++ Γ_step rs`.

We will show that $\Gamma_{\blacktriangle} \vdash_{\lambda(\cap)} ? : \blacktriangle$ is equivalent to $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$, which is formalized [31] in `MainResult.v` as

```
Theorem correctness : ∀ (rs : list rule),
  (∃ (m : nat), rewrites_to rs (repeat 0 (1+m)) (repeat 1 (1+m))) ↔
  (∃ (N : term), normal_form N ∧
    strict_derivation (Γ_init ++ Γ_step rs) N (atom triangle)).
```

Similarly to the Coppo-Dezani type assignment system, it suffices to restrict inhabitants to β-normal forms [67, Theorem 2.15]. Since $\Gamma_{\blacktriangle}$ does not contain an occurrence of ω, the results from [40] remain applicable, albeit with a more intricate case analysis (cf. [31]).

As observed by Urzyczyn in [66], intersection type inhabitation amounts to solving a set of "parallel" judgement constraints

$$\Gamma_1 \vdash_{\lambda(\cap)} M : \phi_1, \ldots, \Gamma_n \vdash_{\lambda(\cap)} M : \phi_n \text{ with } \operatorname{dom}(\Gamma_i) = \operatorname{dom}(\Gamma_j) \text{ for } 1 \leq i, j \leq n$$

If we are able to establish a linear order on such parallel constraints, we can represent a word $v = a_1 \ldots a_n$ by derived types $\phi_1, \ldots, \phi_n$ and transition rules by assumptions in $\Gamma_1, \ldots, \Gamma_n$ (leading to the idea of *bus machines* in [66]). In presence of intersection introduction, we are able to extend the current word (leading to the idea of *expanding tape machines* in [66]).

First, we establish a linear order on parallel judgements in order to encode a word. For this purpose (and differently from [66]), we define the following types and the family of type environments Γ_i^n

$$\sigma_i^j = \begin{cases} \{l\} & \text{if } i = j \\ \{r\} & \text{if } i = j+1 \\ \{\bullet\} & \text{otherwise} \end{cases} \qquad \Gamma_i^n = \{y_j : \sigma_i^j \mid 1 \leq j < n\} \text{ for } 1 \leq i$$

By construction, the following Lemma 10 lets us locate "neighboring" judgements Γ_j^n and Γ_{j+1}^n.

Lemma 10. $\Gamma_{n+1}^n \vdash_{\lambda(\cap)} y_j : \bullet$ *iff* $\Gamma_j^n \vdash_{\lambda(\cap)} y_j : l$ *and* $\Gamma_{j+1}^n \vdash_{\lambda(\cap)} y_j : r$ *and* $\Gamma_i^n \vdash_{\lambda(\cap)} y_j : \bullet$ *for* $i < j$ *or* $i > j+1$.

Before we proceed to soundness and completeness of the reduction, let us illustrate in the following Example 17 in which way $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$ corresponds to $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} ? : \blacktriangle$.

Example 17. *Let* $\mathcal{R} = \{00 \to 1a, a0 \to 1b, b0 \to 11\}$ *be a simple semi-Thue system over the alphabet* $\Sigma = \{0, 1, a, b\}$. *We have* $0^4 \twoheadrightarrow_{\mathcal{R}} 1^4$ *because*

$$0000 \to_{\mathcal{R}} 1a00 \to_{\mathcal{R}} 11b0 \to_{\mathcal{R}} 1111$$

Let

$$N = x_{00\to 1a}\ y_1\ N' \qquad N' = x_{a0\to 1b}\ y_2\ N'' \qquad N'' = x_{b0\to 11}\ y_3\ x_1$$
$$M = x_\blacktriangle\ (x_*\ (\lambda y_1.x_*\ (\lambda y_2.x_*\ (\lambda y_3.x_0\ N))))$$
$$\Gamma_i = \Gamma_\blacktriangle \cup \Gamma_i^4 \text{ for } i = 1 \ldots 5$$

We have

$\Gamma_1 \vdash_{\lambda(\cap)} x_1 : 1$	$\Gamma_2 \vdash_{\lambda(\cap)} x_1 : 1$	$\Gamma_3 \vdash_{\lambda(\cap)} x_1 : 1$	$\Gamma_4 \vdash_{\lambda(\cap)} x_1 : 1$	$\Gamma_5 \vdash_{\lambda(\cap)} x_1 : 1$
$\Gamma_1 \vdash_{\lambda(\cap)} N'' : 1$	$\Gamma_2 \vdash_{\lambda(\cap)} N'' : 1$	$\Gamma_3 \vdash_{\lambda(\cap)} N'' : b$	$\Gamma_4 \vdash_{\lambda(\cap)} N'' : 0$	$\Gamma_5 \vdash_{\lambda(\cap)} N'' : 1$
$\Gamma_1 \vdash_{\lambda(\cap)} N' : 1$	$\Gamma_2 \vdash_{\lambda(\cap)} N' : a$	$\Gamma_3 \vdash_{\lambda(\cap)} N' : 0$	$\Gamma_4 \vdash_{\lambda(\cap)} N' : 0$	$\Gamma_5 \vdash_{\lambda(\cap)} N' : 1$
$\Gamma_1 \vdash_{\lambda(\cap)} N : 0$	$\Gamma_2 \vdash_{\lambda(\cap)} N : 0$	$\Gamma_3 \vdash_{\lambda(\cap)} N : 0$	$\Gamma_4 \vdash_{\lambda(\cap)} N : 0$	$\Gamma_5 \vdash_{\lambda(\cap)} N : 1$
$\Gamma_1 \vdash_{\lambda(\cap)} x_0\, N : *$	$\Gamma_2 \vdash_{\lambda(\cap)} x_0\, N : *$	$\Gamma_3 \vdash_{\lambda(\cap)} x_0\, N : *$	$\Gamma_4 \vdash_{\lambda(\cap)} x_0\, N : \#$	$\Gamma_5 \vdash_{\lambda(\cap)} x_0\, N : \$$
$\Gamma_\blacktriangle \vdash_{\lambda(\cap)} M : \blacktriangle$				

Read from bottom to top, after the initial expansion to $***\#\$$ *the assigned types are initialized to* 00001 *and replaced step by step according to* $\mathcal{R}$. *The term variables* y_1, y_2, y_3 *are used to locate particular rewriting steps.*

Soundness

In this paragraph we outline the proof that $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} ? : \blacktriangle$ implies $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$.

The following Lemma 11 shows that, once particular parallel constraints are established for a word v, we obtain $v \twoheadrightarrow_{\mathcal{R}} 1 \dots 1$.

Lemma 11. *Let $v = a_1 \dots a_n \in \Sigma^n$ where $n > 0$, and let $\Gamma_i = \Gamma_\blacktriangle \cup \Gamma_i^n$ for $1 \leq i$. If there exists a term N such that $\Gamma_i \vdash_{\lambda(\cap)} N : a_i$ for $1 \leq i \leq n$ and $\Gamma_{n+1} \vdash_{\lambda(\cap)} N : 1$, then $v \twoheadrightarrow_{\mathcal{R}} 1^n$.*

Proof. Induction on the size of N in β-normal form (cf. [40, Lemma 4.3]). □

The formalization of the above argumentation is found in `Soundness.v` as `Lemma soundness_step`.

It remains to show that the above parallel constraints are necessarily met. The following Lemma 12 shows that parallel constraints required in the above Lemma 11 are necessarily established.

Lemma 12. *If there exists $n > 0$ and a term N such that $\Gamma_\blacktriangle \cup \Gamma_i^n \vdash_{\lambda(\cap)} N : *$ for $1 \leq i < n$, $\Gamma_\blacktriangle \cup \Gamma_n^n \vdash_{\lambda(\cap)} N : \#$, and $\Gamma_\blacktriangle \cup \Gamma_{n+1}^n \vdash_{\lambda(\cap)} N : \$$, then there exists an $n' > 0$ and N' such that $\Gamma_\blacktriangle \cup \Gamma_i^{n'} \vdash_{\lambda(\cap)} N' : 0$ for $i = 1 \dots n'$, and $\Gamma_\blacktriangle \cup \Gamma_{n'+1}^{n'} \vdash_{\lambda(\cap)} N' : 1$.*

Proof. Induction on the size of N in β-normal form (cf. [40, Lemma 4.4]). □

The formalization of the above argumentation is found in `Soundness.v` as `Lemma soundness_expand`.

Using the above results we can show soundness of the described reduction in the following Theorem 3.

Theorem 3. *If $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} N : \blacktriangle$, then there exists an $n > 0$ such that $0^n \twoheadrightarrow_{\mathcal{R}} 1^n$.*

Proof. Wlog. N is in β-normal form. Since $\blacktriangle$ appears only in $\sigma_\blacktriangle$ in the type environment, we have $N = x_\blacktriangle\, M$ such that $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} M : \#$ and $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} M : \$$. Since $\Gamma_1^1 = \Gamma_2^1 = \emptyset$, we have $\Gamma_\blacktriangle \cup \Gamma_1^1 \vdash_{\lambda(\cap)} M : \#$ and $\Gamma_\blacktriangle \cup \Gamma_2^1 \vdash_{\lambda(\cap)} M : \$$. Therefore, by Lemma 12, there exists an $n' > 0$ and N' such that $\Gamma_\blacktriangle \cup \Gamma_i^{n'} \vdash_{\lambda(\cap)} N' : 0$ for $i = 1 \dots n'$, and $\Gamma_\blacktriangle \cup \Gamma_{n'+1}^{n'} \vdash_{\lambda(\cap)} N' : 1$. Therefore, by Lemma 11, we obtain $0^{n'} \twoheadrightarrow_{\mathcal{R}} 1^{n'}$. □

The above argumentation is formalized in `Soundness.v` as `Lemma soundness`.

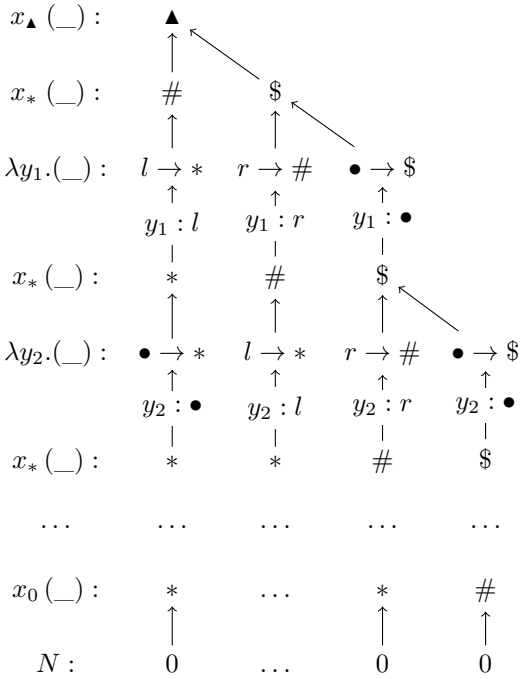
Completeness

In this paragraph we observe that a positive answer to $0^? \twoheadrightarrow_{\mathcal{R}} 1^?$ implies a positive answer to $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} ? : \blacktriangle$. Since any derivation in the Coppo-Dezani type assignment system directly translates to a derivation in the strict intersection type system, we obtain the following Theorem 4 as an immediate consequence of [40, Lemma 4.6].

Theorem 4. *If there exists an $n > 0$ such that $0^n \twoheadrightarrow_{\mathcal{R}} 1^n$, then there exists a term N such that $\Gamma_\blacktriangle \vdash_{\lambda(\cap)} N : \blacktriangle$.*

The proof of the above Theorem 4 is formalized in `Completeness.v` as `Lemma completeness` and is identical to the formalization of [40, Lemma 4.6].

Complementary to Example 17 we would like to visualize the construction diagrammatically. The expansion phase introduces neighbor information as types of y_i splitting the last constraint containing the symbol \$. Afterwards the word representation is initialized as $0 \dots 0$. The last constraint containing the symbol 1 does not have neighbor information, remaining unchanged.



Second, once expanded, the rewriting happens as follows ending in the representation of the word $1 \dots 1$ typed by x_1.

$x_{ab\to cd}\ y_j\ (\_):$	a_1	...	a_{j-1}	a	b	a_{j+1}	...	a_n	1
$N[x_1]:$	a_1	...	a_{j-1}	c	d	a_{j+1}	...	a_n	1
...	...	...	...	...	...	...	...	...	...
$x_1:$	1	...	1	1	1	1	...	1	1

Concluding Remarks

With some additional work, the Turing machine halting problem can be reduced to intersection type inhabitation directly [36] using the techniques outlined in this work and the encoding remarked in Section 2.2.1. The main advantage of the presented reduction in comparison to [66] is that it concisely encodes local computation without introducing additional machinery (such as expanding tape machines). This simplifies formalization of the reduction in a proof assistant. Additionally, this demonstrates techniques (such as parallel constraint ordering) and can be used to specify local computation at the level of types that can be used for type-based synthesis.

2.3 Dimensionally Bounded Intersection Type System

Since key decision problems (such as inhabitation) are undecidable in the Coppo-Dezani-Venneri type assignment system (cf. Section 2.2), it is natural to ask whether there are meaningful restrictions of the full system that exhibit decidable properties. In this section we give an overview over the line of work regarding so-called bounded-dimensional restrictions of intersection type systems [34, 38, 39], for which some key decision problems are decidable.

In the Context of Synthesis

A better understanding of decidable restrictions of intersection type systems leads to new algorithmic perspectives to parameterize inhabitant search, thus improving practical implementations.

As in Section 2.2, we use the strict intersection type system (Definition 13) for which the type language is stratified into strict types $\mathbb{T}_!^\cap \ni \phi, \psi ::= \alpha \mid \sigma \to \phi$ and intersection types $\sigma, \tau ::= \{\phi_1, \ldots, \phi_n\}$ where $n \geq 0$ (Definition 12).

Before we consider restrictions, let us introduce *elaborations* (Definition 17) as λ-terms in which every subterm is annotated by some intersection type. We refer to the individual annotations as *decorations*. Erasing decorations in a given elaboration $\mathbf{P}$ using $\lceil\cdot\rceil$ (Definition 18) results in a λ-term $M = \lceil\mathbf{P}\rceil$, and we say that $\mathbf{P}$ *elaborates* M. We abbreviate by $\mathbf{0}_M$ the unique elaboration of M in which every decoration is the empty intersection ω.

Definition 17 (Elaborations)**.**

$$\mathbf{P}, \mathbf{Q}, \mathbf{R} ::= x\langle\tau\rangle \mid (\lambda x.\mathbf{P})\langle\tau\rangle \mid (\mathbf{P}\ \mathbf{Q})\langle\tau\rangle$$

In the Context of Synthesis

Statically typed programming languages often embed type annotations into program text to guide the type checker. Elaborations extend λ-terms providing this capability.

Definition 18 (Erasure, $\lceil\cdot\rceil$)**.**

$$\lceil x\langle\tau\rangle\rceil = x \qquad \lceil(\lambda x.\mathbf{P})\langle\tau\rangle\rceil = \lambda x.\lceil\mathbf{P}\rceil \qquad \lceil(\mathbf{P}\ \mathbf{Q})\langle\tau\rangle\rceil = \lceil\mathbf{P}\rceil\ \lceil\mathbf{Q}\rceil$$

We write $\mathbf{P} : \tau$ if the outermost decoration of $\mathbf{P}$ is τ. Elaborations of the same term can be combined using point-wise set union on decorations (Definition 19). Whenever we refer to $\mathbf{P} \sqcup \mathbf{Q}$, the condition $\lceil\mathbf{P}\rceil = \lceil\mathbf{Q}\rceil$ is implicitly enforced.

Definition 19 (Elaboration Union, $\sqcup$)**.**

$$\begin{aligned} x\langle\sigma\rangle \sqcup x\langle\tau\rangle &= x\langle\sigma \cup \tau\rangle \\ (\lambda x.\mathbf{P})\langle\sigma\rangle \sqcup (\lambda x.\mathbf{Q})\langle\tau\rangle &= (\lambda x.\mathbf{P} \sqcup \mathbf{Q})\langle\sigma \cup \tau\rangle \\ (\mathbf{P}\ \mathbf{Q})\langle\sigma\rangle \sqcup (\mathbf{P}'\ \mathbf{Q}')\langle\tau\rangle &= ((\mathbf{P} \sqcup \mathbf{P}')\ (\mathbf{Q} \sqcup \mathbf{Q}'))\langle\sigma \cup \tau\rangle \end{aligned}$$

Clearly, $\sqcup$ is associative, commutative, and idempotent, and for any elaboration $\mathbf{P}$ we have $\mathbf{0}_{\lceil \mathbf{P} \rceil} \sqcup \mathbf{P} = \mathbf{P}$.

Using the above notions we define the *elaborated strict intersection type system* (Definition 20) using decorations to keep track of all types assigned to particular subterms (cf. [34, Section 3.1]).

Definition 20 (Elaborated Strict Intersection Type System, $\vdash_{\langle\lambda\rangle(\cap)}$)**.**

$$\frac{\phi \in \sigma}{\Gamma, x : \sigma \vdash_{\langle\lambda\rangle(\cap)} x\langle\{\phi\}\rangle : \phi}\ (\cap\mathrm{E}_{\langle\mathrm{S}\rangle})$$

$$\frac{\Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \sigma \to \phi \qquad \Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{Q} : \sigma}{\Gamma \vdash_{\langle\lambda\rangle(\cap)} (\mathbf{P}\ \mathbf{Q})\langle\{\phi\}\rangle : \phi}\ (\to\mathrm{E}_{\langle\mathrm{S}\rangle})$$

$$\frac{\Gamma, x : \sigma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \phi}{\Gamma \vdash_{\langle\lambda\rangle(\cap)} (\lambda x.\mathbf{P})\langle\{\sigma \to \phi\}\rangle : \sigma \to \phi}\ (\to\mathrm{I}_{\langle\mathrm{S}\rangle})$$

$$\frac{\Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P}_i : \phi_i \qquad \lceil \mathbf{P}_i \rceil = M \qquad (i = 1 \ldots n, n \geq 0)}{\Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{0}_M \sqcup \mathbf{P}_1 \sqcup \ldots \sqcup \mathbf{P}_n : \{\phi_1, \ldots, \phi_n\}}\ (\cap\mathrm{I}_{\langle\mathrm{S}\rangle})$$

Observe that the type system in [34, Section 3.1] does not include ω, therefore the rule $(\cap\mathrm{I}_{\langle\mathrm{S}\rangle})$ is restricted to $n \geq 1$, corresponding to the original Coppo-Dezani type asignment system [23].

Erasing all decorations in a $(\vdash_{\langle\lambda\rangle(\cap)})$-derivation results in a $(\vdash_{\lambda(\cap)})$-derivation and vice versa (Lemma 13).

Lemma 13. *$\Gamma \vdash_{\lambda(\cap)} M : \tau$ iff there exists an elaboration $\mathbf{P}$ such that $\lceil \mathbf{P} \rceil = M$ and $\Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \tau$*

We say an elaboration $\mathbf{P}$ is *well-typed* if $\Gamma \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \tau$ for some type environment Γ and some intersection type τ.

Example 18. *Let $\mathbf{P} = (\lambda x.x\langle\{\alpha, \beta\}\rangle)\langle\{\{\alpha\} \to \alpha, \{\beta\} \to \beta\}\rangle = \mathbf{0}_{\lambda x.x} \sqcup \mathbf{P}_1 \sqcup \mathbf{P}_2$, where $\mathbf{P}_1 = (\lambda x.x\langle\{\alpha\}\rangle)\langle\{\{\alpha\} \to \alpha\}\rangle$ and $\mathbf{P}_2 = (\lambda x.x\langle\{\beta\}\rangle)\langle\{\{\beta\} \to \beta\}\rangle$. The elaboration $\mathbf{P}$ is well-typed because we can derive*

$$\dfrac{\dfrac{\dfrac{\alpha \in \{\alpha\}}{\{x : \{\alpha\}\} \vdash_{\langle\lambda\rangle(\cap)} x\langle\{\alpha\}\rangle : \alpha}\ (\cap\mathrm{E}_{\langle\mathrm{S}\rangle})}{\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P}_1 : \{\alpha\} \to \alpha}\ (\to\mathrm{I}_{\langle\mathrm{S}\rangle}) \qquad \dfrac{\dfrac{\beta \in \{\beta\}}{\{x : \{\beta\}\} \vdash_{\langle\lambda\rangle(\cap)} x\langle\{\beta\}\rangle : \beta}\ (\cap\mathrm{E}_{\langle\mathrm{S}\rangle})}{\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P}_2 : \{\beta\} \to \beta}\ (\to\mathrm{I}_{\langle\mathrm{S}\rangle})}{\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{0}_{\lambda x.x} \sqcup \mathbf{P}_1 \sqcup \mathbf{P}_2 : \{\{\alpha\} \to \alpha, \{\beta\} \to \beta\}}\ (\cap\mathrm{I}_{\langle\mathrm{S}\rangle})$$

Cardinalities of individual decorations indicate how many distinct types are assigned to corresponding subterms anywhere in the derivation. Let us define the *max-norm* (Definition 21) of an elaboration as the maximal cardinality of its decorations.

Definition 21 (Max-Norm, $\|\cdot\|$)**.**

$$\begin{aligned} \|x\langle\tau\rangle\| &= |\tau| \\ \|(\lambda x.\mathbf{P})\langle\tau\rangle\| &= \max\{|\tau|, \|\mathbf{P}\|\} \\ \|(\mathbf{P}\ \mathbf{Q})\langle\tau\rangle\| &= \max\{|\tau|, \|\mathbf{P}\|, \|\mathbf{Q}\|\} \end{aligned}$$

Intuitively, the max-norm captures the "amount of intersection introduction" used to type the term. In the above Example 18 we can observe that $\|(\lambda x.x\langle\{\alpha,\beta\}\rangle)\langle\{\{\alpha\}\to\alpha,\{\beta\}\to\beta\}\rangle\| = 2$, which reflects the fact that intersection introduction has been used at least once. Additionally, the max-norm satisfies the following properties, making it a suitable measure.

$$\begin{aligned} \|\mathbf{P}\sqcup\mathbf{Q}\| &\leq \|\mathbf{P}\|+\|\mathbf{Q}\| && \text{(triangle inequality)}\\ \|\mathbf{P}\| &\geq 0 && \text{(non-negativity)}\\ \|\mathbf{P}\| &= 0 \text{ implies } \mathbf{P}=\mathbf{0}_{\lceil\mathbf{P}\rceil} \end{aligned}$$

In the Context of Synthesis

The max-norm of an elaboration measures the number of distinct features of the corresponding program (and its subprograms).

Restricting the max-norm provides a meaningful way to bound the system.

Definition 22 (Bounded-dimensional Strict Intersection Type System, $\vdash_d$)**.** *We write $\Gamma\vdash_d M:\tau$ if there exists an elaboration $\mathbf{P}$ such that $\Gamma\vdash_{\langle\lambda\rangle(\cap)}\mathbf{P}:\tau$, $\lceil\mathbf{P}\rceil = M$, and $\|\mathbf{P}\|\leq d$.*

We call the parameter d the *dimension* of $(\vdash_d)$. As an immediate consequence of Lemma 13, any $(\vdash_{\lambda(\cap)})$-derivation is captured in some dimension (Corollary 1).

Corollary 1. *$\Gamma\vdash_{\lambda(\cap)} M:\tau$ iff $\Gamma\vdash_d M:\tau$ for some $d\geq 0$.*

In the Context of Synthesis

The dimensional bound limits the number of distinct features (or, layers of refinement) that are considered simultaneously.

Capturing arbitrary large fragments of the full system, each dimensionally restricted fragment is, by itself, a meaningful type system satisfying subject reduction (Theorem 5).

Lemma 14 (Substitutivity under Non-Increasing Max-Norm)**.**
If $\Gamma,x:\sigma\vdash_{\langle\lambda\rangle(\cap)}\mathbf{P}:\tau$ and $\Gamma\vdash_{\langle\lambda\rangle(\cap)}\mathbf{Q}:\sigma$, then there exists an elaboration $\mathbf{R}$ such that $\Gamma\vdash_{\langle\lambda\rangle(\cap)}\mathbf{R}:\tau$, $\lceil\mathbf{R}\rceil = \lceil\mathbf{P}\rceil[x:=\lceil\mathbf{Q}\rceil]$, and $\|\mathbf{R}\|\leq\max\{\|\mathbf{P}\|,\|\mathbf{Q}\|\}$.

Proof. (Sketch) First, we show that $\Gamma\vdash_d\mathbf{Q}:\sigma_1\cup\ldots\cup\sigma_l$ can be decomposed into $\Gamma\vdash_d\mathbf{Q}_1:\sigma_1,\ldots,\Gamma\vdash_d\mathbf{Q}_l:\sigma_l$ such that $\mathbf{Q}=\mathbf{Q}_1\sqcup\ldots\sqcup\mathbf{Q}_l$ (similar to [34, Lemma 15]). The claim is obtained by showing that if $\Gamma,x:\sigma\vdash_d\mathbf{P}:\tau$ such that $x\langle\sigma_1\rangle,\ldots,x\langle\sigma_l\rangle$ are occurrences of x in $\mathbf{P}$, then replacing the occurrences $x\langle\sigma_i\rangle$ in $\mathbf{P}$ by $\mathbf{Q}_i$ for $i=1\ldots l$ results in the elaboration $\mathbf{R}$ with $\Gamma\vdash_d\mathbf{R}:\tau$ (similar to [34, Lemma 16]). □

Theorem 5 (Subject Reduction in Bounded Dimension [34, Theorem 19][2])**.**
If $\Gamma\vdash_d M:\tau$ and $M\to_\beta N$, then $\Gamma\vdash_d N:\tau$.

Proof. (Sketch) Contextual closure of Lemma 14. □

[2]The proof in [34] can be extended to encompass ω.

In the Context of Synthesis

Satisfying the subject reduction property, bounded-dimensional fragments constitute well-defined typed programming languages. In bounded dimension the type system can express interaction of feature vectors limited in size by the dimension.

Section Outline The remainder of this section outlines the line of work covering key decision problems (such as inhabitation, typability, type checking) in dimensionally bounded type systems [34, 38, 39].

First, the one-dimensional fragment is considered in Section 2.3.1 for which inhabitation is decidable and type checking is conjectured to be in NP.

In Section 2.3.2 the elaborated, relevant restriction of the strict intersection type system is described accompanied by the notion of elaboration compactness, which can be considered a stronger notion of irredundancy than relevance.

Sections 2.3.3 and 2.3.4 give an overview over the properties of inhabitation, typability, and type checking in bounded dimension.

Section 2.3.5 briefly sketches a variant of the notion of dimension in which intersection is non-idempotent and inhabitation is decidable.

Authorship Statement Since contributions presented in this section are part of joint work [34, 38, 39], this mandatory paragraph lists the following contributions attributed to the author.

- the relevant elaborated system, $\mathfrak{R}$ (Definition 25)
- relevant filtrations (Definition 28) and associated properties required for basis construction
- analysis of upper and lower bounds of typability and type checking in bounded-dimension (Section 2.3.4) and non-idempotent dimension (Section 2.3.5)

2.3.1 One-Dimensional Fragment

The one-dimensional restriction ($\vdash_1$) of the strict intersection type system disallows any meaningful use of the intersection introduction rule ($\cap\mathrm{I}_{\langle\mathrm{S}\rangle}$). Such a restriction has been considered under the name *explicit intersection* [56] and has several key properties.

Unsurprisingly, any simply typed λ-term M can be assigned the corresponding strict type in ($\vdash_1$). Therefore, from the perspective of *reduction complexity*, λ-terms typable in ($\vdash_1$) may give rise to β-reduction sequences of non-elementary length (Example 19).

Example 19. *Let* $\mathsf{c}_2 = \lambda f x.f\ (f\ x)$, $\mathsf{I} = \lambda x.x$, *and* $M_n = \underbrace{\mathsf{c}_2 \ldots \mathsf{c}_2}_{n\ times}\ \mathsf{I}\ \mathsf{I}$.
We have $\emptyset \vdash_1 M_n : \{\alpha \to \alpha\}$ *for all* $n \geq 0$, *and* $M_n \twoheadrightarrow_\beta \underbrace{\mathsf{I}\ (\mathsf{I}\ (\ldots (\mathsf{I}}_{m\ times}\ \mathsf{I})\ldots)) \twoheadrightarrow_\beta \mathsf{I}$
where m *is non-elementary in* n.

Under closer inspection of principal type schemes [24, Definition 8], any λ-term (resp. approximant in the sense of [1]) in β-normal (resp. $\beta\bot$-normal) form can be assigned its principal type in ($\vdash_1$) (Example 20).

Example 20. *Let* $\phi = \big\{\{\alpha\} \to \beta, \alpha\big\} \to \beta$, *which is the principal type of* $\lambda x.x\ x$ *(cf. [68, Definition 10.6]), and let* $\mathbf{P} = \big(\lambda x.(x\langle\{\{\alpha\} \to \beta\}\rangle\ x\langle\{\alpha\}\rangle)\langle\{\beta\}\rangle\big)\langle\{\phi\}\rangle$.
We have $\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \{\phi\}$, $\lceil\mathbf{P}\rceil = \lambda x.x\ x$, *and* $\|\mathbf{P}\| = 1$. *Therefore, we have* $\emptyset \vdash_1 \lambda x.x\ x : \{\phi\}$.

> In the Context of Synthesis
>
> In the one-dimensional fragment of the strict intersection type system any normal form has a meaningful specification (its principal type).

Types that can be assigned in ($\vdash_1$) are bounded neither by rank (Example 21) nor by cardinality of occurring intersection types (Example 22).

Example 21. *Let* $\phi_0 = \big\{\{\alpha\} \to \beta, \alpha\big\} \to \beta$ *and* $\phi_{n+1} = \big\{\{\phi_n\} \to \gamma_n\big\} \to \gamma_n$ *for* $n \geq 0$. *Let* $M_0 = \lambda x.x\ x$ *and* $M_{n+1} = \lambda y_n.y_n\ M_n$ *for* $n \geq 0$. *Inductively, we have* $\emptyset \vdash_1 M_n : \{\phi_n\}$ *while* ϕ_n *is of rank* $2n + 2$ *for* $n \geq 0$.

Example 22. *Let* $\phi_0 = \alpha_0$ *and* $\phi_{n+1} = \alpha_n \to \alpha_{n+1}$ *for* $n \geq 0$. *Let* $M_0 = x$ *and* $M_{n+1} = x\ M_n$ *for* $n \geq 0$. *Inductively, we have*
$\emptyset \vdash_1 \lambda x.M_n : \{\{\phi_0, \ldots, \phi_n\} \to \phi_n\}$ *while* $|\{\phi_0, \ldots, \phi_n\}| = n + 1$ *for* $n \geq 0$.

Inhabitation in the corresponding explicit intersection type system without subtyping is, similarly to the simply typed system, PSpace-complete [56]. Since the empty intersection type ω can be assigned to any λ-term, typability in bounded dimension is trivial. If ω is excluded, then typability in ($\vdash_1$) is in PSpace [38]. Since the lower bound construction uses the fixed dimension of four [38, Proposition 20], one-dimensional typability (without ω) may be of lower complexity.

The complexity of type checking in ($\vdash_1$) (Problem 5) appears to be not yet studied. We conjecture that type checking in ($\vdash_1$) is NP-complete (Conjecture 1).

Problem 5 (One-dimensional Type Checking)**.** *Given a type environment* Γ*, a* λ*-term* M *and an intersection type* τ*, does* $\Gamma \vdash_1 M : \tau$ *hold?*

Conjecture 1. *One-dimensional type checking (Problem 5) is* NP*-complete.*

Our reasoning is as follows. The NP lower bound follows analogically to the construction in the proof of [38, Proposition 29] by reduction from monotone one-in-three-3SAT. As for the NP upper bound, we expect the approach in [38, Algorithm $\mathscr{S}$] to be applicable in order to non-deterministically reduce type checking to syntactic unification. Complexity in higher dimension arises from decomposing elaborations in order to invert the intersection introduction rule. In ($\vdash_1$) this is not required because decorations are of cardinality at most 1.

In the Context of Synthesis

The one-dimensional fragment of the strict intersection type system is, due to its expressiveness and decidability of the corresponding inhabitation problem, an interesting candidate for practical type-based synthesis from scratch. Additionally, normal principal inhabitants can be reconstructed from their corresponding principal type [39, Theorem 6.5] in dimension one.

2.3.2 Relevant Restriction and Compactness

Relevance is a broadly studied concept in logic (resp. type theory) by which a proof (resp. type derivation) is required to contain only those assumptions that are relevant to reach its conclusion. For example, in the simply typed λ-calculus we can derive $\emptyset \vdash_{\lambda(\to)} \lambda yx.x : \beta \to \alpha \to \alpha$ although the particular type assumption $y : \beta$ is not relevant to type $\lambda x.x$. In fact, we even *have* to assign some particular type to y. Having the universal intersection type ω at our disposal which inherently carries no particular type information, i.e. $\Gamma(y) = \omega$ implies $y \notin \mathrm{dom}(\Gamma)$, we can derive $\emptyset \vdash_{\lambda(\cap)} \lambda yx.x : \omega \to \{\alpha\} \to \alpha$. Still, $(\vdash_{\lambda(\cap)})$ admits weakening, i.e. $\Gamma \vdash_{\lambda(\cap)} M : \tau$ and $\Gamma \subseteq \Gamma'$ imply $\Gamma' \vdash_{\lambda(\cap)} M : \tau$, which admits unnecessary assumptions.

The *relevant intersection type system* (Definition 23) is designed to capture exactly the relevant assumptions. Let us define the union[3] of type environments pointwise, i.e. $(\Gamma_1 \cup \Gamma_2)(x) = \Gamma_1(x) \cup \Gamma_2(x)$.

Definition 23 (Relevant Intersection Type System [68, Definition 10.1], $\vdash_{\mathrm{R}}$)**.**

$$\frac{}{\{x : \{\phi\}\} \vdash_{\mathrm{R}} x : \phi}\ (\mathrm{Ax}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{R}} M : \sigma \to \phi \qquad \Gamma_2 \vdash_{\mathrm{R}} N : \sigma}{\Gamma_1 \cup \Gamma_2 \vdash_{\mathrm{R}} M\ N : \phi}\ (\to\mathrm{E})$$

$$\frac{\Gamma, x : \sigma \vdash_{\mathrm{R}} M : \phi}{\Gamma \vdash_{\mathrm{R}} \lambda x.M : \sigma \to \phi}\ (\to\mathrm{I}) \qquad \frac{\Gamma_1 \vdash_{\mathrm{R}} M : \phi_1 \qquad \dots \qquad \Gamma_n \vdash_{\mathrm{R}} M : \phi_n}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash_{\mathrm{R}} M : \{\phi_1, \ldots, \phi_n\}}\ (\cap\mathrm{I})$$

Observe that the rule $(\to\mathrm{I})$ includes the case $x \notin \mathrm{dom}(\Gamma)$ in which we have $\Gamma = \Gamma, x : \omega$.

Example 23. *We have $\emptyset \vdash_{\mathrm{R}} \lambda yx.x : \omega \to \{\alpha\} \to \alpha$, observing that $\emptyset = \emptyset, y : \omega$, by the following derivation*

$$\dfrac{\dfrac{\dfrac{}{\{x : \{\alpha\}\} \vdash_{\mathrm{R}} x : \alpha}\ (\mathrm{Ax})}{\emptyset \vdash_{\mathrm{R}} \lambda x.x : \{\alpha\} \to \alpha}\ (\to\mathrm{I})}{\emptyset \vdash_{\mathrm{R}} \lambda yx.x : \omega \to \{\alpha\} \to \alpha}\ (\to\mathrm{I})$$

However, $\emptyset \nvdash_{\mathrm{R}} \lambda yx.x : \{\beta\} \to \{\alpha\} \to \alpha$ because the type assumption $y : \{\beta\}$ is not used.

The above relevant intersection type system enjoys subject reduction (consequence of [68, Lemma 10.2]) and can be considered the core of the most prominent intersection type assignment systems (e.g. the Barendregt-Coppo-Dezani type assignment system [3], cf. [68, Section 6.2 and Theorem 10.5]). Accordingly, it is directly related to the strict intersection type system (Definition 13) via the essential subtyping relation (Definition 24).

Definition 24 (Essential Subtyping [68, Definition 4.1 (iii)], $\leq_{\mathrm{E}}$)**.** *The relation $\leq_E$ is the least pre-order on $\mathbb{T}^{\cap}_{\{!\}}$ satisfying*

$$\{\phi_1, \ldots, \phi_n\} \leq_E \{\phi_i\} \text{ for } i \in \{1, \ldots, n\}$$
$$\tau \leq_E \{\phi_i\} \text{ for } i = 1 \ldots n \text{ implies } \tau \leq_E \{\phi_1, \ldots, \phi_n\}$$
$$\tau \leq_E \sigma \text{ and } \{\phi\} \leq_E \{\psi\} \text{ implies } \{\sigma \to \phi\} \leq_E \{\tau \to \psi\}$$

[3] If we would have used the $\cap$ type constructor, we would rather define intersections of type environments [68, Definition 3.5]. This however is confusing (and sometimes ambiguous) if type environments (and/or intersections) are treated as sets.

We extend $\leq_{\mathrm{E}}$ pointwise to type environments. Clearly, any derivable relevant judgement can also be derived in the strict intersection type system. The converse requires weakening the assumptions and strengthening the derived type using $\leq_{\mathrm{E}}$ (Lemma 15).

Lemma 15. *If $\Gamma \vdash_{\lambda(\cap)} M : \tau$, then there exists a type environment Γ' and a type τ' such that $\Gamma' \vdash_{\mathrm{R}} M : \tau'$, $\Gamma \leq_E \Gamma'$, and $\tau' \leq_E \tau$.*

Proof. (Sketch) Derivations in $(\vdash_{\lambda(\cap)})$ are conservative wrt. the so-called essential intersection type system [68, Definition 4.3]. We obtain the result by [68, Theorem 10.5]. □

Similarly to the elaborated strict intersection type system (Definition 20) we could extend the above Definition 23 to elaborations (cf. [39, Figure 1]). However, a relevant derivation captures as type assumption $x : \sigma$ in the type environment exactly the strict types $\phi \in \sigma$ occurring as $\{x : \{\phi\}\} \vdash_{\mathrm{R}} x : \phi$ in the derivation. Therefore, the type σ coincides with the set of strict types appearing in decorations of x in the elaboration. In fact, for a relevant system, elaborations contain sufficient information to reconstruct the corresponding type environment. We use this observation in Definition 25 to give a more concise presentation of an elaborated relevant strict intersection type system as the set $\mathfrak{R}$ of *relevant elaborations*.

Let us denote by $\mathcal{T}(\mathbf{P})$ the union of all decorations in $\mathbf{P}$, and by $\mathcal{T}_x(\mathbf{P})$ the union of all decorations of occurrences of the free variable x in $\mathbf{P}$

$$\mathcal{T}(\mathbf{P}) = \{\phi \in \mathbb{T}^{\cap}_{!} \mid \phi \in \tau \text{ for some decoration } \tau \text{ in } \mathbf{P}\}$$
$$\mathcal{T}_x(\mathbf{P}) = \{\phi \in \mathbb{T}^{\cap}_{!} \mid \phi \in \tau \text{ for some subterm } x\langle\tau\rangle \text{ in } \mathbf{P}\} \text{ where } x \in \mathrm{FV}(\mathbf{P})$$

Definition 25 (Relevant Elaborations, $\mathfrak{R}$)**.**

$$\frac{}{x\langle\{\phi\}\rangle \in \mathfrak{R}}\ (\mathrm{Ax}_{\mathfrak{R}}) \qquad \frac{\mathbf{P} \in \mathfrak{R} \qquad \mathbf{P} : \{\phi\}}{(\lambda x.\mathbf{P})\langle\{\mathcal{T}_x(\mathbf{P}) \to \phi\}\rangle \in \mathfrak{R}}\ (\to\mathrm{I}_{\mathfrak{R}})$$

$$\frac{\mathbf{P} \in \mathfrak{R} \quad \mathbf{P} : \{\sigma \to \phi\} \qquad \mathbf{Q} \in \mathfrak{R} \quad \mathbf{Q} : \sigma}{(\mathbf{P}\ \mathbf{Q})\langle\{\phi\}\rangle \in \mathfrak{R}}\ (\to\mathrm{E}_{\mathfrak{R}})$$

$$\frac{\mathbf{P}_i \in \mathfrak{R} \quad \mathbf{P}_i : \{\phi_i\} \quad \lceil\mathbf{P}_i\rceil = M \qquad (i = 1 \ldots n, n \geq 0)}{\mathbf{0}_M \sqcup \mathbf{P}_1 \sqcup \ldots \sqcup \mathbf{P}_n \in \mathfrak{R}}\ (\cap\mathrm{I}_{\mathfrak{R}})$$

Observe that for any term M using the rule $(\cap\mathrm{I}_{\mathfrak{R}})$ and $n = 0$ we have $\mathbf{0}_M \in \mathfrak{R}$.

We denote by $\Gamma_{\mathbf{P}}$ the type environment defined by collecting decorations of corresponding free variables in a given elaboration $\mathbf{P}$

$$\Gamma_{\mathbf{P}} = \{x : \mathcal{T}_x(\mathbf{P}) \mid x \in \mathrm{FV}(\lceil\mathbf{P}\rceil)\}$$

As indicated above, we can reconstruct from a given elaboration in $\mathfrak{R}$ the corresponding relevant type environment (Lemma 16). Conversely, any relevant derivation corresponds to some elaboration in $\mathfrak{R}$ (Lemma 16).

Lemma 16. *$\Gamma \vdash_{\mathrm{R}} M : \tau$ iff there exists an elaboration $\mathbf{P} \in \mathfrak{R}$ such that $\Gamma_{\mathbf{P}} = \Gamma$, $\lceil\mathbf{P}\rceil = M$, and $\mathbf{P} : \tau$.*

The following Example 24 shows a non-relevant elaboration that would otherwise be well-typed. Complementarily, Example 25 shows a corresponding relevant elaboration.

Example 24. *Let* $\phi = \{\beta\} \to \beta$, $\mathbf{P} = \big(\lambda x.(\lambda y.y\langle\{\beta\}\rangle)\langle\{\phi\}\rangle\big)\langle\{\{\alpha\} \to \phi\}\rangle$, $\mathbf{Q} = z\langle\{\alpha\}\rangle$, *and* $\mathbf{R} = (\mathbf{P}\,\mathbf{Q})\langle\{\phi\}\rangle$. *In a non-relevant setting, the elaboration* $\mathbf{R}$ *is well-typed. However,* $\mathbf{R} \notin \mathfrak{R}$ *since in the abstraction* x *is assigned a non-empty intersection type* $\{\alpha\}$*, which is not used in* $\mathbf{P}$.

Example 25. *Let* $\phi = \{\beta\} \to \beta$, $\mathbf{P} = \big(\lambda x.(\lambda y.y\langle\{\beta\}\rangle)\langle\{\phi\}\rangle\big)\langle\{\omega \to \phi\}\rangle$, $\mathbf{Q} = z\langle\omega\rangle$, $\mathbf{R} = (\mathbf{P}\,\mathbf{Q})\langle\{\phi\}\rangle$. *We have* $\mathbf{R} \in \mathfrak{R}$ *and* $\Gamma_{\mathbf{R}} \vdash_R \lceil\mathbf{R}\rceil : \{\phi\}$ *where* $\Gamma_{\mathbf{R}} = \{z : \omega\} = \emptyset$ *and* $\lceil\mathbf{R}\rceil = (\lambda xy.y)\,z$.

In the Context of Synthesis

Since relevance ensures that any unused assumption is typed by ω, any argument that is decorated by ω in a relevant elaboration can be considered dead code.

Compactness and Filtrations

Somewhat surprisingly, relevant typing still may contain superfluous type information (Example 26).

Example 26. *Let* $\phi = \{\alpha\} \to \alpha$. *We have* $\emptyset \vdash_{\mathrm{R}} \lambda x.x : \{\phi\} \to \phi$, *and accordingly* $\mathbf{P} = (\lambda x.x\langle\{\phi\}\rangle)\langle\{\{\phi\} \to \phi\}\rangle \in \mathfrak{R}$. *However,* ϕ *contains unnecessary structure for the particular type derivation. Accordingly, the strict type* α *is not decorating in* $\mathbf{P}$.

Inspecting a relevant elaboration $\mathbf{P} \in \mathfrak{R}$ such that $\mathbf{P} : \tau$, one can observe that if a strict subformula of τ is not decorating in $\mathbf{P}$, then its structure is not necessary for the particular derivation. Let us say that an elaboration $\mathbf{P}$ is *tight*, if $\mathcal{T}(\mathbf{P})$ is closed under strict subformulae (Definition 26). If the elaboration is relevant and tight, then we call it *compact* (Definition 27).

Definition 26 (Tightness)**.** *We say* $\mathbf{P}$ *is* tight*, if for all* $\phi \in \mathcal{T}(\mathbf{P})$ *and all* ψ *that are strict subformulae of* ϕ *we have* $\psi \in \mathcal{T}(\mathbf{P})$.

Definition 27 (Compactness)**.** *We say* $\mathbf{P}$ *is* compact*, if* $\mathbf{P} \in \mathfrak{R}$ *and* $\mathbf{P}$ *is tight.*

Relevance and tightness are orthogonal notions (Examples 26, 27, 28, and 29).

Example 27. *Let* $\phi = \{\alpha\} \to \{\beta\} \to \alpha$. *We have* $\emptyset \vdash_{\lambda(\cap)} \lambda xy.x : \phi$. *Accordingly for* $\mathbf{P} = \big(\lambda x.(\lambda y.x\langle\{\alpha\}\rangle)\langle\{\{\beta\} \to \alpha\}\rangle\big)\langle\{\phi\}\rangle\big)$ *we have* $\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \phi$. *The elaboration* $\mathbf{P}$ *is not relevant (the assumption* β *is not required), and not tight (the strict subformula* β *is not decorating).*

Example 28. *We have* $\emptyset \vdash_{\mathrm{R}} \lambda x.x : \{\alpha\} \to \alpha$. *Accordingly, for the elaboration* $\mathbf{P} = (\lambda x.x\langle\{\alpha\}\rangle)\langle\{\{\alpha\} \to \alpha\}\rangle$ *we have that* $\mathbf{P}$ *is both relevant and tight. Therefore,* $\mathbf{P}$ *is compact.*

Example 29. *Let* $\phi = \{\alpha\} \to \{\alpha\} \to \alpha$. *We have* $\emptyset \vdash_{\lambda(\cap)} \lambda xy.x : \phi$. *Accordingly for* $\mathbf{P} = \big(\lambda x.(\lambda y.x\langle\{\alpha\}\rangle)\langle\{\{\alpha\} \to \alpha\}\rangle\big)\langle\{\phi\}\rangle\big)$ *we have* $\emptyset \vdash_{\langle\lambda\rangle(\cap)} \mathbf{P} : \phi$. *The elaboration* $\mathbf{P}$ *is not relevant (the second assumption* α *is not required), however it is tight (all strict subformulae* $\{\alpha, \{\alpha\} \to \alpha, \phi\}$ *are decorating).*

In the Context of Synthesis

Compact, well-typed elaborations constitute pieces of code that are annotated by exactly their specification.

The most important feature of a tight elaboration $\mathbf{P}$ of a λ-term M is that the cardinality of (arbitrary nested) intersection types is bounded by the number of decorating types. Simultaneously, the number of decorating types is directly limited by the size of M and the max-norm of $\mathbf{P}$. This allows us (Section 2.3.4) to decide typability and type checking in bounded dimension.

A particularly important instrument developed throughout the line of work of [34, 38, 39] is the notion of *filtrations*. Intuitively, for a set $X \subseteq \mathbb{T}^\cap_!$ of strict types the filtration $\mathscr{F}_X$ removes subformulae of its input type that are, in a sense, not supported by X. In practice, X is chosen to be the set of decorating types of a given elaboration in order to establish tightness.

Definition 28 (Relevant Filtration $\mathscr{F}_X$)**.** *Let $X \subseteq \mathbb{T}^\cap_!$ be a set of strict types. We define the* filtration $\mathscr{F}_X : \mathbb{T}^\cap_! \to \mathbb{T}^\cap_!$ *(and tacitly lift it to $\mathbb{T}^\cap_{\{!\}}$) by*

$$\mathscr{F}_X(\beta) = \beta$$

$$\mathscr{F}_X(\sigma \to \psi) = \begin{cases} \mathscr{F}_X(\sigma) \to \mathscr{F}_X(\psi) & \text{if } \sigma \cup \{\sigma \to \psi, \psi\} \subseteq X \\ \alpha_{\sigma\to\psi} & \text{otherwise} \end{cases}$$

$$\mathscr{F}_X(\{\phi_1, \dots, \phi_n\}) = \{\mathscr{F}_X(\phi_1), \dots, \mathscr{F}_X(\phi_n)\}$$

We define $\mathscr{F}_X(\mathbf{P})$ pointwise on decorations and $\mathscr{F}_X(\Gamma)$ pointwise on type environments. Let us revisit Example 26 applying a filtration in the following Example 30.

Example 30. *Let $\phi = \{\alpha\} \to \alpha$, $\mathbf{P} = (\lambda x.x\langle\{\phi\}\rangle)\langle\{\{\phi\} \to \phi\}\rangle \in \mathfrak{R}$, and $X = \mathcal{T}(\mathbf{P}) = \{\phi, \{\phi\} \to \phi\}$. We have*

$$\mathscr{F}_X(\mathbf{P}) = (\lambda x.x\langle\{\alpha_\phi\}\rangle)\langle\{\{\alpha_\phi\} \to \alpha_\phi\}\rangle \in \mathfrak{R}$$

Observe that $\mathscr{F}_X(\mathbf{P})$ is compact (tight and relevant), $\mathscr{F}_X(\mathbf{P})$ has the same max-norm as $\mathbf{P}$, and the strict substitution $S(\alpha_\phi) = \phi$ reverts the filtration, i.e. $S(\mathscr{F}_X(\mathbf{P})) = \mathbf{P}$.

The properties in the above Example 30 are systematic. In particular, filtrations preserve relevance (Lemma 17), ensure tightness (Lemma 18), and do not increase norm (Lemma 19).

Lemma 17 ([39, Lemma 5.15])**.** *For $\mathbf{P} \in \mathfrak{R}$ we have $\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P}) \in \mathfrak{R}$.*

Proof. We show a stronger claim by induction on derivation depth: for any $X \supseteq \mathcal{T}(\mathbf{P})$ we have $\mathscr{F}_X(\mathbf{P}) \in \mathfrak{R}$.

Case ($\mathrm{Ax}_\mathfrak{R}$)**:** We have $\mathbf{P} = x\langle\{\phi\}\rangle$, therefore $\mathscr{F}_X(\mathbf{P}) = x\langle\{\mathscr{F}_X(\phi)\}\rangle \in \mathfrak{R}$.

Case ($\to\mathrm{I}_\mathfrak{R}$)**:** We have $\mathbf{P} = (\lambda x.\mathbf{Q})\langle\{\mathcal{T}_x(\mathbf{Q}) \to \psi]\}\rangle$ such that $\mathbf{Q} \in \mathfrak{R}$ and $\mathbf{Q} : \{\psi\}$. Observing that $\{\mathcal{T}_x(\mathbf{Q}) \to \psi, \psi\} \cup \mathcal{T}_x(\mathbf{Q}) \subseteq \mathcal{T}(\mathbf{P}) \subseteq X$, we have

$$\mathscr{F}_X(\mathbf{P}) = (\lambda x.\mathscr{F}_X(\mathbf{Q}))\langle\{\mathscr{F}_X(\mathcal{T}_x(\mathbf{Q})) \to \mathscr{F}_X(\psi)\}\rangle$$

Since $\mathscr{F}_X(\mathcal{T}_x(\mathbf{Q})) = \mathcal{T}_x(\mathscr{F}_X(\mathbf{Q}))$, and $\mathscr{F}_X(\mathbf{Q}) \in \mathfrak{R}$ due to the induction hypothesis, we obtain $\mathscr{F}_X(\mathbf{P}) \in \mathfrak{R}$ by rule ($\to\mathrm{I}_\mathfrak{R}$).

Case ($\to\mathrm{E}_{\mathfrak{R}}$)**:** We have $\mathbf{P} = (\mathbf{Q}\,\mathbf{R})\langle\{\psi\}\rangle$ such that $\mathbf{Q}, \mathbf{R} \in \mathfrak{R}$, $\mathbf{Q} : \{\sigma \to \psi\}$ and $\mathbf{R} : \sigma$ for some σ. Observing that $\sigma \cup \{\sigma \to \psi, \psi\} \subseteq \mathcal{T}(\mathbf{P}) \subseteq X$, we have

$$\mathscr{F}_X(\mathbf{P}) = (\mathscr{F}_X(\mathbf{Q})\ \mathscr{F}_X(\mathbf{R}))\langle\{\mathscr{F}_X(\psi)\}\rangle$$

with $\mathscr{F}_X(\mathbf{Q}) : \{\mathscr{F}_X(\sigma) \to \mathscr{F}_X(\psi)\}$ and $\mathscr{F}_X(\mathbf{R}) : \mathscr{F}_X(\sigma)$. By the induction hypothesis $\mathscr{F}_X(\mathbf{Q}), \mathscr{F}_X(\mathbf{R}) \in \mathfrak{R}$, therefore $\mathscr{F}_X(\mathbf{P}) \in \mathfrak{R}$ by rule ($\to\mathrm{E}_{\mathfrak{R}}$).

Case ($\cap\mathrm{I}_{\mathfrak{R}}$)**:** We have $\mathbf{P} = \mathbf{0}_M \sqcup \bigsqcup_{i=1}^{n} \mathbf{P}_i$ such that $\mathbf{P}_i \in \mathfrak{R}$ and $\mathbf{P}_i : \{\psi_i\}$ for $i = 1 \ldots n$. Since filtration is applied pointwise, we have $\mathscr{F}_X(\mathbf{P}) = \mathscr{F}_X(\mathbf{0}_M \sqcup \bigsqcup_{i=1}^{n} \mathbf{P}_i) = \mathbf{0}_M \sqcup \bigsqcup_{i=1}^{n} \mathscr{F}_X(\mathbf{P}_i)$ and $\mathscr{F}_X(\mathbf{P}_i) : \{\mathscr{F}_X(\psi_i)\}$ for $i = 1 \ldots n$. By the induction hypothesis $\mathscr{F}_X(\mathbf{P}_i) \in \mathfrak{R}$ for $i = 1 \ldots n$, therefore $\mathscr{F}_X(\mathbf{P}) \in \mathfrak{R}$ by rule ($\cap\mathrm{I}_{\mathfrak{R}}$). □

Although the above proof of Lemma 17 is by routine induction on derivation depth, it underlines why exactly the particular definition of filtration is used for the relevant system. The results in [38, 37, 39] all use slightly different notions of filtrations to take into account the different properties of the underlying type system in each case.

Lemma 18 ([39, Lemma 5.16])**.** *For $\mathbf{P} \in \mathfrak{R}$ the elaboration $\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})$ is tight.*

Lemma 19 ([39, Lemma 5.17])**.** *For $\mathbf{P} \in \mathfrak{R}$ we have $\|\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})\| \leq \|\mathbf{P}\|$.*

Combining the above Lemmas 17, 18, and 19 in the following Corollary 2 establishes compactness under non-increasing norm.

Corollary 2. *For $\mathbf{P} \in \mathfrak{R}$ the elaboration $\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})$ is compact and we have $\|\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})\| \leq \|\mathbf{P}\|$.*

A key property that was first observed in [39] is that a filtration may be reversed by a single strict substitution (Lemma 20).

Lemma 20 ([39, Lemma 5.19])**.** *For $\mathbf{P} \in \mathfrak{R}$, there exists a strict substitution S such that $S(\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})) = \mathbf{P}$.*

Proof. (Sketch) Let $\{\alpha_{\phi_1}, \ldots, \alpha_{\phi_m}\}$ be the additional type variables introduced in $\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\mathbf{P})$. Define $S(\alpha_{\phi_i}) = \phi_i$ for $i = 1 \ldots m$. For $\phi \in \mathcal{T}(\mathbf{P})$ one shows $S(\mathscr{F}_{\mathcal{T}(\mathbf{P})}(\phi)) = \phi$ by induction in the size of the syntax tree of ϕ. □

Using filtrations allows us to focus on compact elaborations (e.g. for type inference) for which type structure is bounded by dimension and term size. Conveniently, any relevant elaboration can be constructed from a compact one by means of a single strict substitution. We will use this property in Section 2.3.4 to inspect decidability of type checking in bounded dimension.

In the Context of Synthesis

Applying a filtration to a program containing specification annotations may result in a more concise specification of the underlying functionality. Although principal type information can be considered the most concise specification, it may be difficult to compute.

2.3.3 Inhabitation in Bounded Dimension

Since bounded dimensional fragments restrict the "amount of intersection introduction", one could hope that inhabitation in bounded dimension (Problem 6) would be decidable. Unfortunately, it is not (Theorem 6).

Problem 6 (Bounded-dimensional Inhabitation, $\Gamma \vdash_d ? : \tau$)**.** *Given a type environment Γ, an intersection type τ and a dimensional parameter d, is there a λ-term M such that $\Gamma \vdash_d M : \tau$ holds?*

Theorem 6 (cf. [34, Theorem 28])**.** *Bounded-dimensional inhabitation (Problem 6) is undecidable.*

Proof. (Sketch) Inspecting the proof of undecidability of intersection type inhabitation in Section 2.2.2 reveals that only subformulae of the given type environment Γ and input type τ are assigned to subterms of the inhabitant (cf. subformula property [3, Lemma 4.5]). Therefore, the maximal cardinality of any decoration in the elaboration (hence, its max-norm) of the inhabitant is bounded by the number of subformulae in the input. Choosing the dimensional parameter accordingly, we can adjust the proof of Theorem 2 in the bounded dimensional setting. □

In the Context of Synthesis

Unfortunately, bounding the number of considered orthogonal features does not immediately result in a tractable synthesis approach.

In [34] the corresponding undecidability result is obtained directly from normalization and subformula properties. However, in the presence of the empty intersection type ω, we do not necessarily have the normalization property (Example 31).

Example 31. *Let $\Omega = (\lambda x.x\ x)\ (\lambda x.x\ x)$ and $M = x\ \Omega$. Although M cannot be reduced to a β-normal form, we have $\{x : \omega \to \alpha\} \vdash_1 M : \alpha$.*

As observed in Section 2.3.1, inhabitation in $(\vdash_1)$ is decidable. This raises the (open) question of decidability of inhabitation in $(\vdash_d)$ for fixed d (Problem 7). We conjecture that there exists $d > 1$ such that inhabitation in $(\vdash_d)$ is undecidable (Conjecture 2).

Problem 7 (Fixed-dimensional Inhabitation)**.** *Let $d \geq 0$. Given a type environment Γ and an intersection type τ, is there a λ-term M such that $\Gamma \vdash_d M : \tau$ holds?*

Conjecture 2. *There exists $d > 1$ such that fixed-dimensional inhabitation (Problem 7) is undecidable.*

Our reasoning is as follows. Using techniques from Section 2.2.2 we might represent a universal Turing machine as a type environment Γ. If we are able to represent inputs for that Turing machine as a type τ without arbitrarily increasing dimension necessary to decide inhabitation, then a similar argumentation as in Section 2.2.2 might apply. This seems feasible because of the apparent connection between the number of distinct types on the right-hand sides of

parallel constraints (not the actual number of constraints, cf. Example 17) and the dimension of the inhabitant. The fixed dimension d would be dictated by the size of the universal Turing machine as well as the dimensional increase required to represent arbitrary inputs.

Let us conclude this section by observing that *principal inhabitation* (given a type environment Γ and an intersection type τ, does there exist a $\lambda\bot$-term M in $\beta\bot$-normal form such that (Γ, τ) is the principal pair of M in the sense of [24]?) is decidable in polynomial time [39, Theorem 6.5]. Surprising at first glance, this fact is due to severe restrictions on the structure of types that are principally inhabited (cf. *weakly balanced* [39, Definition 6.1]), allowing for a direct inversion of the principal pair generation procedure [24]. The following Examples 32 and 33 illustrate that the restricted shape of principal types dictates the structure of the corresponding λ-term.

Example 32. *Let $\phi = \{\{\alpha\} \to \beta, \{\beta\} \to \gamma\} \to \{\alpha\} \to \gamma$. Each type variable occurs in ϕ at most once positively and at most once negatively (cf.* weakly balanced *[39, Definition 6.1]). The only λ-term in β-normal form typed by $\{\phi\}$ is its principal inhabitant, the second Church-numeral, $\lambda f x.f\,(f\,x)$. Since principal type derivations do not require intersection introduction, we also have $\emptyset \vdash_1 \lambda f x.f\,(f\,x) : \{\phi\}$.*

Example 33. *Let $\phi = \{\alpha\} \to \alpha$ and $\psi = \{\phi\} \to \phi$. Although the intersection type $\{\psi\}$ is inhabited by any Church-numeral, it is not principally inhabited by a λ-term in β-normal form because the type variable α occurs more than two times in ψ (cf. [24, Definition 8]).*

In the Context of Synthesis

Polynomial time decidability of principal inhabitation hints at the fact that principal types precisely (in fact, uniquely) specify their corresponding principal inhabitant. As a result, principal types do not alleviate description complexity of desired inhabitants.

Despite its restrictive nature, principal inhabitation can be combined with a type inference algorithm [39, Algorithm 2] in order to construct approximants of a given λ-term which is not necessarily in β-normal form. In fact, any approximant of a given term can be discovered using this approach in suitable dimension [39, Theorem 7.6]. Let us illustrate this approach in the following Example 34.

Example 34. *Let $M = \lambda xy.x\,y\,x$, $N = \lambda x.x\,x$, $\phi = \{\alpha\} \to \omega \to \beta$, and $\tau = \{\{\phi, \alpha\} \to \beta\}$. In dimension one we may infer [39, Algorithm 2] that $\emptyset \vdash_1 M\,N : \tau$. In fact, $M\,N \twoheadrightarrow_\beta \lambda y.y\,y\,N$ which has $F = \lambda y.y\,y\,\bot$ as a direct approximant. Additionally, τ is principally inhabited by F. Therefore, F can be constructed from τ by [39, Algorithm 1]. As a result, we discovered an approximant of the λ-term $(M\,N)$ using type inference in bounded dimension combined with principal inhabitation.*

2.3.4 Typability, Type Checking, and Bases

As observed in the previous Section 2.3.3, even in bounded dimension inhabitation is undecidable. Interestingly, the dual problem of typability (Problem 8) behaves differently in bounded dimension. Of course, having the intersection type ω at our disposal, any λ-term can be typed by ω in dimension zero. We could disallow ω, which results in an elaborated variant of the original Coppo-Dezani type assignment system [23]. Typability under this restriction is PSPACE-complete [38, Theorem 25]. The upper bound is shown algorithmically by a non-deterministic reduction to a type unification problem on type constraints. The lower bound is shown by observing that the standard reduction from quantified Boolean formula satisfiability can be performed in dimension four.

Problem 8 (Bounded-dimensional Typability, $? \vdash_d M : ?$)**.** *Given a dimensional parameter d and a λ-term M, is there a type environment Γ and an intersection type τ such that $\Gamma \vdash_d M : \tau$ holds?*

Compared to the simply typed λ-calculus, intersection type assignment systems have a more intricate theory of principality [24]. In particular, it involves considering the set of approximants of a given term in order to construct a principal pair from which all typings of the given term can be reached using chains of certain operations (type expansions, liftings, and substitutions). The operation of type expansion, that imitates intersection introduction, is particularly complex (for a survey see [19]). However, if we are interested in typings of a term in bounded dimension, we do not need the full power of type expansions. In fact, the theory of principality in bounded dimension can be presented algebraically in terms of computable bases of an abstract vector space (more precisely, semimodule) [39]. In particular, given a λ-term M and a dimension parameter d we can algorithmically construct [39, Theorem 7.4] a unique modulo renaming, finite basis $\mathcal{B}_{M,d} \subseteq \mathbb{T}_!^\cap$ such that the strict span of $\mathcal{B}_{M,d}$ contains exactly those strict types that can be relevantly assigned to M in dimension d [39, Theorem 5.20 (1)]. A strict span of a set of strict types contains exactly all strict instances of those types [39, Definition 4.8]. Additionally, the span of $\mathcal{B}_{M,d}$, where intersection plays the role of vector addition, contains all types that can be relevantly assigned to M in dimension d [39, Theorem 5.20 (2)].

In the Context of Synthesis

The finite basis for a given program in a given dimension, similar to a principal type, contains necessary information to obtain any strict type relevantly assignable to the given program in the given dimension.
In contrast to a principal type, the basis can be constructed algorithmically for a program which is not in normal form. Additionally, types are obtained by means of substitution and do not require (chains of) expansions.

Let us compare the notion of principality and the notion of basis in the following Example 35. Intuitively, the basis spans bounded dimensional typings of a term without using type expansion.

Example 35. *Let* $M = \lambda x.x\,x$*, and*

$$\phi = \{\{\alpha\} \to \beta, \alpha\} \to \beta$$
$$\psi = \{\{\alpha, \beta\} \to \gamma, \alpha, \beta\} \to \gamma$$
$$\tau = \{\{\omega \to \xi_3\} \to \xi_3, \{\{\xi_1, \xi_2\} \to \xi_3, \xi_1, \xi_2\} \to \xi_3\} \text{ for some } \xi_1, \xi_2, \xi_3 \in \mathbb{T}^{\cap}_{!}$$

We have $\emptyset \vdash_{\mathrm{R}} M : \phi$*,* $\emptyset \vdash_{\mathrm{R}} M : \psi$*,* $\emptyset \vdash_{\mathrm{R}} M : \tau$*, and* $\emptyset \vdash_2 M : \tau$*.*

The strict type ϕ *is the principal type of* M *in the sense of [24]. In order to obtain* τ *from the principal type* ϕ*, we require two type expansions followed by a lifting and a substitution.*

In dimension 2 *the basis of* M *is* $\mathcal{B}_{M,\,2} = \{\psi\}$*. The type* τ *is obtained by the linear combination* $\tau = S(\psi) \cap T(\psi)$ *where* $S = \{\alpha \mapsto \omega, \beta \mapsto \omega, \gamma \mapsto \xi_3\}$ *and* $T = \{\alpha \mapsto \xi_1, \beta \mapsto \xi_2, \gamma \mapsto \xi_3\}$*.*

Spanning assignable types of a given λ-term M by linear combinations of basis types can be seen as a generalization of principality in the simply typed λ-calculus where the basis consist exactly of the principal type of M that can be scaled by instantiation.

We would like to conclude this section with a conjecture that type checking in bounded dimension is decidable (Conjecture 3) because for a given term its basis is finite and computable.

Problem 9 (Bounded-dimensional Type Checking, $\Gamma \vdash_d M : \tau$?)**.** *Given a dimensional parameter* d*, a type environment* Γ*, a* λ*-term* M*, and an intersection type* τ *does* $\Gamma \vdash_d M : \tau$ *hold?*

Conjecture 3. *Bounded-dimensional type checking (Problem 9) is decidable.*

Our argument is as follows. Let us first consider the strict scenario, i.e. $\Gamma \vdash_d M : \{\psi\}$. If $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$, let $N = \lambda x_1 \ldots x_n.M$. It suffices to decide whether $\emptyset \vdash_d N : \{\sigma_1 \to \ldots \to \sigma_n \to \psi\}$ holds. Using the basis construction theorem [39, Theorem 7.4] and Algorithm $\mathscr{S}(N, d)$ [39, Algorithm 2] we can compute the finite basis $\mathcal{B}_{N,d}$ spanning types of N. Using Lemma 15, it suffices to decide whether there exists a $\phi \in \mathcal{B}_{N,d}$ such that $S(\phi) \leq_{\mathrm{E}} \sigma_1 \to \ldots \to \sigma_n \to \psi$ for some strict substitution S. This corresponds to essential intersection type matching and, conceivably, is decidable (cf. [28]). However, the type structure and subtyping rules studied in [28] slightly differ from $\leq_{\mathrm{E}}$. Next, let us consider the general case, i.e. $\Gamma \vdash_d M : \tau$. Unfortunately, it does not suffice to verify that $\Gamma \vdash_d M : \phi_i$ holds for $i = 1 \ldots m$ where $\tau = \{\phi_1, \ldots, \phi_m\}$ because, introducing an intersection, the required dimension may increase. Therefore, we have to follow a different approach. Similarly to the strict case, we can use Algorithm $\mathscr{S}(N, d)$ [39, Algorithm 2] to inspect all compact elaborations of N in dimension d. Using intersection type matching, invertibility and monotonicity of filtrations (Lemma 20 and Corollary 2) we may discover a compact filtration that, possibly weakened wrt. $\leq_{\mathrm{E}}$, witnesses the checked type.

2.3.5 Non-idempotent Restriction

Restricting intersection types to be non-idempotent, i.e. multisets, is a common approach to characterize quantitative features of λ-terms [47, 18], and is closely related to linear logic [26]. For example, in system $\mathcal{M}$ of [16] the second Church-numeral $\mathsf{c}_2 = \lambda f x.f\,(f\,x)$ can be typed by $[[\alpha] \to \alpha, [\alpha] \to \alpha] \to [\alpha] \to \alpha$, where $[\cdot]$ denotes a multiset. Since the term variable f appears twice in c_2, it has to assume two copies of the type $[\alpha] \to \alpha$. This approach considers *types as resources* that are consumed upon using corresponding assumptions.

Interestingly, non-idempotent intersection type systems have a decidable inhabitation problem [16] because the type, due to linearity, restricts the size of subterms of inhabitants. Still, non-idempotent intersection type systems characterize normalization properties of λ-terms (for an overview see [18]). Therefore, typability is undecidable, even under this restriction.

In the remainder of this paragraph we outline a different, non-linear use of non-idempotent intersection types, namely in the case of bounded dimension. Compared to the mentioned non-idempotent intersection type systems, the quantitative nature of multisets is not used to impose linearity, but instead the dimensional bound. Intuitively, the former approach uses non-idempotency to count the number of times term variables are used in the term. The latter approach uses non-idempotency to count how many times individual subterms are typed (possibly by the same type). As in the idempotent scenario, intersection introduction (as opposed to terms or types) is treated as a resource.

In the following Examples 36 and 37 we use the strict intersection type system (Definition 13) to provide an intuition for the notion of multiset dimension.

Example 36. *Let* $\mathsf{c}_2 = \lambda f x.f\,(f\,x)$, $\phi = \{\alpha\} \to \alpha$, $\Gamma = \{f : \{\phi\}, x : \{\alpha\}\}$, *and consider the following derivation*

$$\dfrac{\dfrac{\dfrac{\phi \in \Gamma(f)}{\Gamma \vdash_{\lambda(\cap)} f : \phi}\,(\cap\mathrm{E}) \qquad \dfrac{\dfrac{\alpha \in \Gamma(x)}{\Gamma \vdash_{\lambda(\cap)} x : \alpha}\,(\cap\mathrm{E})}{\Gamma \vdash_{\lambda(\cap)} x : \{\alpha\}}\,(\cap\mathrm{I})}{\Gamma \vdash_{\lambda(\cap)} f\,x : \alpha}\,(\to\mathrm{E})}{\Gamma \vdash_{\lambda(\cap)} f\,(f\,x) : \alpha}\,(\to\mathrm{I})}{\{f : \{\phi\}\} \vdash_{\lambda(\cap)} \lambda x.f\,(f\,x) : \{\alpha\} \to \alpha}\,(\to\mathrm{I})}{\emptyset \vdash_{\lambda(\cap)} \lambda f x.f\,(f\,x) : \{\phi\} \to \{\alpha\} \to \alpha}\,(\to\mathrm{I})$$

In the above derivation each subterm of c_2 *is typed exactly once (the intersection introduction rule is used only to embed strict types into intersection types). Therefore,* c_2 *can be typed by* $[[\alpha] \to \alpha] \to [\alpha] \to \alpha$ *in bounded multiset dimension* 1.

Example 37. *Let* $\mathsf{c}_2 = \lambda f x.f\,(f\,x)$, $\phi = \{\alpha, \beta\} \to \alpha$, $\psi = \{\alpha, \beta\} \to \beta$, $\Gamma = \{f : \{\phi, \psi\}, x : \{\alpha, \beta\}\}$, *and consider the following derivation*

$$\dfrac{\dfrac{\dfrac{\dfrac{\phi \in \Gamma(f)}{\Gamma \vdash_{\lambda(\cap)} f : \phi}\,(\cap\mathrm{E}) \qquad \dfrac{\mathcal{D}_1 \rhd \Gamma \vdash_{\lambda(\cap)} f\,x : \alpha \qquad \mathcal{D}_2 \rhd \Gamma \vdash_{\lambda(\cap)} f\,x : \beta}{\Gamma \vdash_{\lambda(\cap)} f\,x : \{\alpha, \beta\}}\,(\cap\mathrm{I})}{\Gamma \vdash_{\lambda(\cap)} f\,(f\,x) : \alpha}\,(\to\mathrm{E})}{\{f : \{\phi, \psi\}\} \vdash_{\lambda(\cap)} \lambda x.f\,(f\,x) : \{\alpha, \beta\} \to \alpha}\,(\to\mathrm{I})}{\emptyset \vdash_{\lambda(\cap)} \lambda f x.f\,(f\,x) : \{\phi, \psi\} \to \{\alpha, \beta\} \to \alpha}\,(\to\mathrm{I})$$

where the derivation $\mathcal{D}_1$ *is*

$$\dfrac{\dfrac{\phi \in \Gamma(f)}{\Gamma \vdash_{\lambda(\cap)} f : \phi}\,(\cap\mathrm{E}) \qquad \dfrac{\dfrac{\alpha \in \Gamma(x)}{\Gamma \vdash_{\lambda(\cap)} x : \alpha}\,(\cap\mathrm{E}) \qquad \dfrac{\beta \in \Gamma(x)}{\Gamma \vdash_{\lambda(\cap)} x : \beta}\,(\cap\mathrm{E})}{\Gamma \vdash_{\lambda(\cap)} x : \{\alpha, \beta\}}\,(\cap\mathrm{I})}{\Gamma \vdash_{\lambda(\cap)} f\, x : \alpha}\,(\to\mathrm{E})$$

and the derivation $\mathcal{D}_2$ *is analogous to* $\mathcal{D}_1$ *using* ψ *instead of* ϕ*. Although the term variable* x *appears only once in* c_2*, it is typed by each* α *and by* β *in each* $\mathcal{D}_1$ *and in* $\mathcal{D}_2$*. Therefore, a multiset dimension of at least* 4 *is required to type* c_2 *by* $[[\alpha, \beta] \to \alpha, [\alpha, \beta] \to \beta] \to [\alpha, \beta, \alpha, \beta] \to \alpha$*.*

The multiset-dimensional type system is introduced in [34, Section 3.2] and enjoys the subject reduction property [34, Theorem 18]. Surprisingly, despite its non-linearity inhabitation in the bounded dimensional non-idempotent system is decidable, and ExpSpace-complete. The upper bound is shown algorithmically [34, Proposition 32]. More precisely, while dimension in the idempotent setting bounds the number of distinct types any subterm is assigned in a type derivation, non-idempotent dimension bounds the number of times any subterm is typed (possibly by the same type) in a type derivation. Therefore, we can bound the number of parallel judgment constraints (in the sense of [66]) that need to be considered during an inhabitant search, leading to a termination argument. The lower bound is shown by observing that the ExpSpace-complete decision problem of rank 2 intersection type inhabitation [66, Theorem 9] requires at most linear non-idempotent dimension. This observation not only establishes the lower bound, but shows that wrt. inhabitation the notion of bounded non-idempotent dimension vastly generalizes the rank 2 restriction.

Type checking in bounded non-idempotent dimension is NP-hard (even in fixed dimension 1) [38, Proposition 29] and typability is in NP [38, Proposition 32]. The former result is shown by reduction from monotone one-in-three-3SAT. The latter result is shown algorithmically by adapting the typability algorithm from the idempotent setting.

Concluding Remarks

Bounded-dimensional intersection type systems provide a way to systematically restrict and approximate their corresponding unbounded counterpart. While key decision problems such as inhabitation, typability and type checking are undecidable in most intersection type systems, some are decidable in bounded dimension. Most importantly, the theory of intersection type principality is algebraically accessible in bounded dimension by means of finite bases (not involving type expansions). Additionally, bounded dimensional non-idempotent intersection type calculi provide a further restriction (and enjoy decidable inhabitation) while not imposing linearity on typed terms.

Areas of general interest with currently open research questions include semantics as well as reduction complexity of bounded-dimensional fragments.

Chapter 3

Combinatory Logic

Combinatory logic, pioneered by Moses Schönfinkel in 1920s [58], only allows terms to consist of applications of select *combinators* (Schönfinkel's *Bausteine*). Depending on the choice of allowed combinators (the *basis*), combinatory logic is as expressive as the λ-calculus. However, in context of program synthesis we are also interested in less expressive bases that exactly cover the given domain of interest.

For a more unified syntax, we call λ-terms without abstractions *combinatory terms* (Definition 29), denoted by F, G, H. We call term variables x appearing in combinatory terms *combinators*.

Definition 29 (Combinatory Terms)**.** $F, G ::= x \mid (F\ G)$

Clearly, combinatory terms are of shape $(x\ F_1 \dots F_n)$ for some $n \geq 0$. To mitigate extensive use of parentheses in combinatory terms we use the left-associative *pipe* metaoperator $\succ$ defined as $F \succ G = (G\ F)$. For example, $G_3\ (G_2\ (G_1\ F)) = F \succ G_1 \succ G_2 \succ G_3$. The *size* of a combinatory term is the number of leaves in its syntax tree (Definition 30).

Definition 30 (Size)**.** $\mathrm{size}(x) = 1$ *and* $\mathrm{size}(F\ G) = \mathrm{size}(F) + \mathrm{size}(G)$.

Since their initial appearance in Schönfinkel's seminal paper [58], particular combinators such as S or K are associated with specific notions of combinatory term reduction. This allows to construct a model of computation (without using abstraction) equivalent in expressiveness to the λ-calculus. Additionally, combinators may be equipped with type schemes in order to construct type systems that expose similar features (e.g. the Curry-Howard isomorphism) as typed λ-calculi [61].

In this chapter we focus our attention on inhabitation in typed combinatory logic (given a set of typed combinators and a type, is there a combinatory term that can be a assigned the given type?). Specifically, we neither fix any particular basis nor impose any particular notion of combinatory term reduction. Relying on combinatory terms (Definition 29) as the term language we will vary the type language (simple types in Section 3.1 and intersection types in Section 3.2) together with corresponding typing rules.

In the Context of Synthesis

Typed combinators represent existing library components that expose properties specified by their types. Simple types can be used to specify (higher-order) functional dependencies whereas intersection types can encode (higher-order) tabular specification wrt. multiple feature dimensions. A positive answer to an inhabitation query $\Gamma \vdash ? : \tau$ is a combinatory term F such that $\Gamma \vdash F : \tau$ is derivable in the underlying type system. The term F represents the synthesized program that satisfies the specification represented by τ.

Chapter Outline In this chapter we inspect properties of two distinct type systems having combinatory terms as their term language.

First, in Section 3.1 we consider the simply typed combinatory logic. We show that inhabitation (given a simple type τ and a simple type environment Γ, is there a combinatory term typable by τ in Γ?) in this type system is undecidable [35]. Additionally, we inspect inhabitation in the simply typed combinatory logic considering a restricted class of type environments. We show that inhabitation is undecidable even considering type environments containing types that are derivable from the axiom $\alpha \to \beta \to \alpha$ in a Hilbert-style calculus, and types that are principal for some λ-terms in β-normal form.

Second, in Section 3.2 we consider bounded combinatory logic with intersection types [30] where type instantiation is bounded by some syntactic measure. We show that inhabitation in combinatory logic with intersection types where instantiation is bounded by the functional order o and functional arity a is $(o + 2)$-ExpTime complete.

Third, in Section 3.3 we inspect properties of intersection type subtyping [3] used in combinatory logic with intersection types, showing three results. First, it is decidable in quadratic time whether two given types are in a subtyping relation. Second, the corresponding matching problem (one of the given types may be instantiated) is not fixed-parameter tractable in the number of type variables [33]. Third, the corresponding unification problem (both given types may be instantiated) is ExpTime-hard [33].

3.1 Simply Typed Combinatory Logic

Simple types (Definition 4) serve as the type language of *simply typed combinatory logic* ($\vdash_{\mathcal{C}(\to)}$) for which the rules (Ax) and ($\to$E) are given in the following Definition 31.

Definition 31 (Simply Typed Combinatory Logic ($\vdash_{\mathcal{C}(\to)}$))**.**

$$\frac{S \text{ is a substitution}}{\Gamma, x : \sigma \vdash_{\mathcal{C}(\to)} x : S(\sigma)}\ (\text{Ax}) \qquad \frac{\Gamma \vdash_{\mathcal{C}(\to)} F : \sigma \to \tau \qquad \Gamma \vdash_{\mathcal{C}(\to)} G : \sigma}{\Gamma \vdash_{\mathcal{C}(\to)} F\ G : \tau}\ (\to\text{E})$$

Observe that the above definition is *relativized* to arbitrary type environments (or, *bases*) Γ whereas the term "simply typed combinatory logic" commonly refers to a fixed basis containing the combinators K of type $\alpha \to \beta \to \alpha$ and S of type $(\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$. We will use *subintuitionistic* bases to inspect decidability of inhabitation "below" S and K in terms of derivability.

Naturally, we have the following generation lemma (Lemma 21).

Lemma 21. *If $\Gamma \vdash_{\mathcal{C}(\to)} x\ F_1 \dots F_n : \tau$, then there exists a substitution S such that $S(\Gamma(x)) = \sigma_1 \to \dots \to \sigma_n \to \tau$ for some $n \in \mathbb{N}$, $\sigma_1, \dots, \sigma_n \in \mathbb{T}^{\to}$ and $\Gamma \vdash_{\mathcal{C}(\to)} F_i : \sigma_i$ holds for $i = 1 \dots n$.*

The decision problem of inhabitation in ($\vdash_{\mathcal{C}(\to)}$) (Problem 10), abbreviated by $\Gamma \vdash_{\mathcal{C}(\to)} ? : \tau$, is whether there exists a combinatory term typable by a given type τ in the given type environment Γ.

Problem 10 (Inhabitation in ($\vdash_{\mathcal{C}(\to)}$), $\Gamma \vdash_{\mathcal{C}(\to)} ? : \tau$)**.** *Given a type environment Γ and a type τ, is there a combinatory term F such that $\Gamma \vdash_{\mathcal{C}(\to)} F : \tau$ is derivable?*

Example 38. *Let $\sigma = (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$, $\tau = \alpha \to \beta \to \alpha$, $\rho_1 = \alpha \to (\alpha \to \alpha) \to \alpha$, $\rho_2 = \alpha \to \alpha \to \alpha$, and $\Gamma = \{\mathsf{S} : \sigma, \mathsf{K} : \tau\}$. The question $\Gamma \vdash_{\mathcal{C}(\to)} ? : \alpha \to \alpha$ has the combinatory term $\mathsf{S}\,\mathsf{K}\,\mathsf{K}$ as a positive answer, shown by the following derivation*

$$\frac{\mathcal{D} \triangleright \Gamma \vdash_{\mathcal{C}(\to)} \mathsf{S}\,\mathsf{K} : \rho_2 \to \alpha \to \alpha \qquad \dfrac{\{\beta \mapsto \alpha\}}{\Gamma \vdash_{\mathcal{C}(\to)} \mathsf{K} : \rho_2}\ (\text{Ax})}{\Gamma \vdash_{\mathcal{C}(\to)} \mathsf{S}\,\mathsf{K}\,\mathsf{K} : \alpha \to \alpha}\ (\to\text{E})$$

where the derivation $\mathcal{D}$ is

$$\frac{\dfrac{\{\beta \mapsto \alpha \to \alpha, \gamma \mapsto \alpha\}}{\Gamma \vdash_{\mathcal{C}(\to)} \mathsf{S} : \rho_1 \to \rho_2 \to \alpha \to \alpha}\ (\text{Ax}) \qquad \dfrac{\{\beta \mapsto \alpha \to \alpha\}}{\Gamma \vdash_{\mathcal{C}(\to)} \mathsf{K} : \rho_1}\ (\text{Ax})}{\Gamma \vdash_{\mathcal{C}(\to)} \mathsf{S}\,\mathsf{K} : \rho_2 \to \alpha \to \alpha}\ (\to\text{E})$$

In the Context of Synthesis

The above Example 38 shows that the black-box library components S and K specified functionally by their assigned types can be applicatively composed to satisfy the functional specification $\alpha \to \alpha$.

Section Outline The remainder of this section outlines the contribution showing that inhabitation in the simply typed combinatory logic (even under several restrictions for considered type environments) is undecidable [35]. To start with, the Hilbert-style calculus, that is related to the simply typed combinatory logic via the Curry-Howard correspondence, is introduced in Section 3.1.1. Next, the problem of recognizing principal axiomatizations of formulae derivable from each of the axioms $\alpha \to \beta \to \alpha$ (Section 3.1.2), $\alpha \to \alpha$ (Section 3.1.3), $\alpha \to \beta \to \beta$ (Section 3.1.4) is investigated via inhabitation in the simply typed combinatory logic. As a result, recognizing principal axiomatizations of $\alpha \to \beta \to \alpha$ is shown to be undecidable (Theorem 7). Complementarily, it is shown that recognizing principal axiomatizations of $\alpha \to \alpha$ (resp. $\alpha \to \beta \to \beta$) is decidable in linear time (Theorem 8 (resp. Theorem 9)).

Authorship Statement Since contributions presented in this section are part of joint work [35], this mandatory paragraph lists the following contributions attributed to the author.

- reduction from the Post correspondence problem to inhabitation in the simply typed combinatory logic
- formalization of the reduction in the Lean proof assistant
- decidability of recognizing axiomatizations of $\alpha \to \alpha$ (resp. $\alpha \to \beta \to \beta$)

3.1.1 Hilbert-Style Calculus

Let us identify propositional implicational *axioms* (sometimes called *formulae*) with simple types ($\mathbb{T}^{\to}$, Definition 4) and denote finite sets of axioms by Δ. The rules (Ax) and ($\to$E) of the *Hilbert-style calculus* ($\vdash_{\mathcal{H}}$) are given in the following Definition 32, which is in direct Curry-Howard correspondence with ($\vdash_{\mathcal{C}(\to)}$).

Definition 32 (Hilbert-Style Calculus ($\vdash_{\mathcal{H}}$))**.**

$$\frac{S \text{ is a substitution}}{\Delta, \sigma \vdash_{\mathcal{H}} S(\sigma)}\ (\text{Ax}) \qquad \frac{\Delta \vdash_{\mathcal{H}} \sigma \to \tau \qquad \Delta \vdash_{\mathcal{H}} \sigma}{\Delta \vdash_{\mathcal{H}} \tau}\ (\to\text{E})$$

The set of *derivable* formulae is denoted by $[\Delta]_{\mathcal{H}} = \{\tau \in \mathbb{T}^{\to} \mid \Delta \vdash_{\mathcal{H}} \tau\}$. We say Δ_1 *axiomatizes* $[\Delta_2]_{\mathcal{H}}$ if $[\Delta_1]_{\mathcal{H}} = [\Delta_2]_{\mathcal{H}}$. Clearly, $[\Delta_1]_{\mathcal{H}} = [\Delta_2]_{\mathcal{H}}$ iff $\Delta_1 \vdash_{\mathcal{H}} \tau$ for all $\tau \in \Delta_2$ and $\Delta_2 \vdash_{\mathcal{H}} \sigma$ for all $\sigma \in \Delta_1$. For brevity, we say Δ axiomatizes σ if $[\Delta]_{\mathcal{H}} = [\{\sigma\}]_{\mathcal{H}}$.

Example 39. *For $\Delta = \{\alpha \to \beta \to \alpha, (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\}$ the set of formulae $[\Delta]_{\mathcal{H}}$ contains exactly the intuitionistic propositional implicational theorems.*

We say that an axiom σ is *principal* if there exists a λ-term M in β-normal form such that σ is the principal type of M in the simply typed λ-calculus. Principality of axioms is decidable [15] and PSpace-complete (Section 2.1).

In the Context of Synthesis

Axioms that are not principal, e.g. $\alpha \to \alpha \to \alpha$, could be considered "artificial" since they have no "naturally" associated implementation.

Derivability in the Hilbert-style calculus is equivalent to inhabitation in the simply typed combinatory logic (Lemma 22).

Lemma 22. *We have $\tau \in [\{\sigma_1, \ldots, \sigma_n\}]_{\mathcal{H}}$ iff the inhabitation problem instance $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \vdash_{\mathcal{C}(\to)} ? : \tau$ has a solution.*

In the Context of Synthesis

Derivability in Hilbert-style calculi from principal axioms is directly related to program synthesis from a collection of existing program pieces (principal inhabitants of the given axioms). The derived formula corresponds to desired properties of the synthesized result.

The problem of derivability in the Hilbert-style calculus was posed by Tarski in 1946 and has subsequently been studied by several authors including Linial and Post [50] in 1949 who have shown undecidability of the problem of recognizing axiomatizations of classical propositional logic. Zolin [70] provides a good overview of further developments regarding this problem.

3.1.2 Recognizing Axiomatizations of $\alpha \to \beta \to \alpha$

In this section we give an overview over the proof in [35] showing that recognizing principal axiomatizations of the Hilbert-style calculus containing only the axiom $\alpha \to \beta \to \alpha$ is undecidable (Theorem 7). Via the Curry-Howard correspondence this result translates to the area of composition synthesis (Corollary 3).

If one is not interested in principality of axioms, then this result follows directly from the recent work by Bokov [12].

Theorem 7. *Given principal axioms $\sigma_1, \ldots, \sigma_n$ such that $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \ldots n$, it is undecidable whether $\{\sigma_1, \ldots, \sigma_n\} \vdash_{\mathcal{H}} \alpha \to \beta \to \alpha$.*

Corollary 3. *Given λ-terms $M_1, \ldots, M_n$ in β-normal form with principal types $\sigma_1, \ldots, \sigma_n$ in the simply typed λ-calculus such that $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \ldots n$, it is undecidable whether there is an applicative composition of $M_1, \ldots, M_n$ typable by $\alpha \to \beta \to \alpha$.*

In the Context of Synthesis

In the above Corollary 3 the types $\sigma_1, \ldots, \sigma_n$ are natural specifications of associated terms $M_1, \ldots, M_n$ and $\alpha \to \beta \to \alpha$ is a goal specification. Deriving $\alpha \to \beta \to \alpha$ from $\sigma_1, \ldots, \sigma_n$ naturally corresponds to finding a composition of given terms satisfying the goal specification.

Theorem 7 is proven by reduction from the Post correspondence problem (Problem 11), which is known for its simplicity and undecidability [53].

Post Correspondence Problem

Problem 11 (Post Correspondence Problem, PCP)**.** *Given pairs of words $(v_1, w_1), \ldots, (v_k, w_k)$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ such that $v_i \neq \epsilon \neq w_i$ for $i = 1 \ldots k$ (where ϵ is the empty word), is there an index sequence $i_1, \ldots, i_n$ for some $n > 0$ such that $v_{i_1} v_{i_2} \ldots v_{i_n} = w_{i_1} w_{i_2} \ldots w_{i_n}$?*

Lemma 23 ([53])**.** *The Post correspondence problem (Problem 11) is undecidable.*

Corollary 4. *Given pairs of words $(v_1, w_1), \ldots, (v_k, w_k)$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ such that $\epsilon \neq v_i \neq w_i \neq \epsilon$ for $i = 1 \ldots k$ it is undecidable whether there exists an index sequence $i_1, \ldots, i_n$ such that $v_1 v_{i_1} v_{i_2} \ldots v_{i_n} = w_1 w_{i_1} w_{i_2} \ldots w_{i_n}$.*

Usually, the Post correspondence problem is approached constructively, i.e. start with some given pair of words, then iteratively append corresponding suffixes, and finally test for equality. The approach taken here is, in a sense, "deconstructive". In particular, we start from an arbitrary pair of equal words, then iteratively remove corresponding suffixes, and finally test whether the resulting pair is given. While the former approach requires an equality test for arbitrarily large structures as the final operation, the final operation of the latter approach can be bounded. The following Definition 33 and Lemma 24 capture the outlined iterative deconstruction.

Definition 33. *Given a set* $\mathsf{PCP} = \{(v_1, w_1), \ldots, (v_k, w_k)\}$ *of pairs of words over the alphabet* $\{\mathtt{a}, \mathtt{b}\}$ *we define for* $n \geq 0$ *the set* PCP_n *of pairs of words as*

$$\mathsf{PCP}_0 = \{(v, v) \mid v \in \{\mathtt{a}, \mathtt{b}\}^*\}$$
$$\mathsf{PCP}_{n+1} = \{(v, w) \mid \exists i \in \{1, \ldots, k\}.(vv_i, ww_i) \in \mathsf{PCP}_n\}$$

Lemma 24. *Let* $n \geq 0$ *and* $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. *We have* $vv_{i_1}v_{i_2}\ldots v_{i_n} = ww_{i_1}w_{i_2}\ldots w_{i_n}$ *for some index sequence* $i_1, \ldots, i_n$ *iff* $(v, w) \in \mathsf{PCP}_n$.

Proof. Routine induction on n. □

In sum, it is undecidable whether the prefix (v_1, w_1) is in PCP_n for some $n \geq 0$ (Corollary 5).

Corollary 5. *Given a set* $\mathsf{PCP} = \{(v_1, w_1), \ldots, (v_k, w_k)\}$ *of pairs of words over the alphabet* $\{\mathtt{a}, \mathtt{b}\}$ *such that* $\epsilon \neq v_i \neq w_i \neq \epsilon$ *for* $i = 1 \ldots k$ *it is undecidable whether there exists an* $n \geq 0$ *such that* $(v_1, w_1) \in \mathsf{PCP}_n$.

Example 40. *Let* $\mathsf{PCP} = \{(v_1, w_1), (v_2, w_2), (v_3, w_3)\}$ *where* $v_1 = \mathtt{a}$, $w_1 = \mathtt{ab}, v_2 = \mathtt{baa}, w_2 = \mathtt{a}, v_3 = \mathtt{a}, w_3 = \mathtt{aa}$. *For the index sequence* $i_1 = 1$, $i_2 = 2$, $i_3 = 3$ *we have* $v_{i_1}v_{i_2}v_{i_3} = \mathtt{abaaa} = w_{i_1}w_{i_2}w_{i_3}$.

Complementarily, we have $(\mathtt{abaaa}, \mathtt{abaaa}) \in \mathsf{PCP}_0$, $(\mathtt{abaa}, \mathtt{aba}) \in \mathsf{PCP}_1$, *and* $(v_1, w_1) = (\mathtt{a}, \mathtt{ab}) \in \mathsf{PCP}_2$.

Proof of Theorem 7

We will show undecidability of recognizing principal axiomatizations of the formula $\alpha \to \beta \to \alpha$ by reduction from the Post correspondence problem using Corollary 5.

Let us fix $\mathsf{PCP} = \{(v_1, w_1), \ldots, (v_k, w_k)\}$ pairs of words over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ such that $\epsilon \neq v_i \neq w_i \neq \epsilon$ for $i = 1 \ldots k$. Our goal is to construct principal axioms $\sigma_1, \ldots, \sigma_l$ such that $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \ldots l$ and $\{\sigma_1, \ldots, \sigma_l\} \vdash_{\mathcal{H}} \alpha \to \beta \to \alpha$ is equivalent to $(v_1, w_1) \in \mathsf{PCP}_n$ for some $n \geq 0$. In this subsection we outline the construction of axioms $\sigma_1, \ldots, \sigma_l$. Principality of $\sigma_1, \ldots, \sigma_l$ is addressed in [35, Section 3] using the Haskell[1] programming language. Additionally, the reduction is formalized using the Lean[2] proof assistant and is part of the main contribution in [35].

We need to represent words, pairs and suffixing. Let us fix a unique type variable $\bullet$. For a word $v \in \{\mathtt{a}, \mathtt{b}\}^*$ we define its representation as $[v] = \bullet \cdot v$ where the operation $\cdot$ is defined as

$$\sigma \cdot \epsilon = \sigma \qquad \sigma \cdot w\mathtt{a} = (\bullet \to \bullet) \to (\sigma \cdot w) \qquad \sigma \cdot w\mathtt{b} = (\bullet \to \bullet \to \bullet) \to (\sigma \cdot w)$$

We represent a pair of types σ, τ as

$$\langle \sigma, \tau \rangle = (\bullet \to \bullet \to \bullet) \to (\sigma \to \tau \to \bullet) \to (\bullet \to \sigma) \to (\bullet \to \tau) \to \bullet \to \bullet \to \bullet$$

The formula $[v]$ contains only $\bullet$ as atoms. Additionally, we have $[v] \cdot w = [vw]$, and representations of two distinct words are not unifiable (Lemma 25).

[1] `https://www.haskell.org/`
[2] `https://leanprover.github.io/`

Lemma 25 ([35, Lemma 19])**.** *Let $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. If $[v]$ and $[w]$ are unifiable, then $v = w$.*

Additionally, for any types σ, τ we have that $\langle \sigma, \tau \rangle$ is derivable from $\alpha \to \beta \to \alpha$ (Lemma 27).

Lemma 26. *Let σ, τ be types. If $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \tau$, then $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma \to \tau$.*

Proof. Use the rule ($\to$E) with the premises $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \tau \to \sigma \to \tau$ and $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \tau$. □

Lemma 27. *Let σ, τ be types. We have $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \langle \sigma, \tau \rangle$.*

Proof. By iterative application of Lemma 26 starting with the judgement $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \bullet \to \bullet \to \bullet$. □

Finally, we define a type environment Γ of $k + 2$ combinators typed by principal axioms

$$\begin{aligned}\Gamma = &\{x : \langle \alpha, \alpha \rangle, z : \langle [v_1], [w_1] \rangle \to \bullet \to \alpha \to \bullet\} \\ &\cup \{y_i : \langle \alpha \cdot v_i, \beta \cdot w_i \rangle \to \langle \alpha, \beta \rangle \mid 1 \leq i \leq k\}\end{aligned}$$

In the Context of Synthesis

The above collection of components Γ specifies at the level of types the construction of the sets PCP_i for $i \geq 0$. The component x constructs arbitrary pairs of equal elements, and the components y_i remove suffixes (v_i, w_i) of a given pair of words for $i = 1 \dots k$. Therefore, typable applicative compositions $x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n}$ construct elements of PCP_n for $n \geq 0$.

Due to Lemma 27, each axiom in $\lceil \Gamma \rceil$ is derivable from $\alpha \to \beta \to \alpha$.

Having established all prerequisite definitions, we now proceed with our main reduction. The following Lemma 28 establishes a connection between elements $(v, w) \in \mathsf{PCP}_n$ and inhabitants of $\langle [v], [w] \rangle$.

Lemma 28. *Let S be a substitution and let $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$.*
If $\Gamma \vdash_{\mathcal{C}(\to)} x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n} : S(\langle [v], [w] \rangle)$ for some index sequence $i_1 \dots i_n$, then $(v, w) \in \mathsf{PCP}_n$.

Proof. Induction on n.

Basis Step: $\Gamma \vdash_{\mathcal{C}(\to)} x : S(\langle [v], [w] \rangle)$ implies $S([v]) = S([w])$. By Lemma 25 we obtain $v = w$.

Inductive Step: Assume $\Gamma \vdash_{\mathcal{C}(\to)} x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n} \succ y_l : S(\langle [v], [w] \rangle)$ for some index sequence $i_1, \dots, i_n, l$. We necessarily have $\Gamma \vdash_{\mathcal{C}(\to)} y_l : \sigma \to S(\langle [v], [w] \rangle)$ and $\Gamma \vdash_{\mathcal{C}(\to)} x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n} : \sigma$ for some type σ. Additionally, $\sigma \to S(\langle [v], [w] \rangle) = T(\langle \alpha \cdot v_l, \beta \cdot w_l \rangle \to \langle \alpha, \beta \rangle)$ for some substitution T, which implies $S([v]) = T(\alpha)$, $S([w]) = T(\beta)$ and $S(\bullet) = T(\bullet)$. Therefore, $T(\alpha \cdot v_l) = S([vv_l])$ and $T(\beta \cdot w_l) = S([ww_l])$. As a result, we have $\sigma = T(\langle \alpha \cdot v_l, \beta \cdot w_l \rangle) = S(\langle [vv_l], [ww_l] \rangle)$.

By the induction hypothesis $(vv_l, ww_l) \in \mathsf{PCP}_n$, which implies $(v, w) \in \mathsf{PCP}_{n+1}$. □

Let us define $\mathfrak{n} \in \mathbb{N} \cup \{\infty\}$ as either the minimal size of a combinatory term typable in Γ by $\sigma \to \sigma \to \sigma$ or as ∞ if no such term exists.

$$\mathfrak{n} = \min\{\text{size}(F) \mid \Gamma \vdash_{\mathcal{C}(\to)} F : \sigma \to \sigma \to \sigma \text{ for some type } \sigma\}$$

Intuitively, a "small", i.e. of size less than $\mathfrak{n}$, derivation of an instance of $\langle [v_1], [w_1] \rangle$ contains no derivation of an instance of $\bullet \to \bullet \to \bullet$. Due to our pair encoding, which has as its first argument the type $\bullet \to \bullet \to \bullet$, we are able to severely restrict the shape of the minimal derivation of $\langle [v_1], [w_1] \rangle$ (Lemma 29).

Lemma 29 ([35, Lemma 23])**.** *If* $\Gamma \vdash_{\mathcal{C}(\to)} F : S(\langle \sigma, \tau \rangle)$ *for some substitution* S *such that* $\text{size}(F) < \mathfrak{n}$*, then* $F = x \succ y_{i_1} \succ \ldots \succ y_{i_m}$ *for some (possibly empty) index sequence* $i_1, \ldots, i_m$*.*

By an exhaustive case analysis, for which the pedantic supervision of a proof assistant shows its strength, we have that if $\alpha \to \beta \to \alpha$ is derivable, then there is a small derivation of an instance of $\langle [v_1], [w_1] \rangle$ (Lemma 30).

Lemma 30 ([35, Lemma 24])**.** *If* $\lceil \Gamma \rceil \vdash_{\mathcal{H}} \alpha \to \beta \to \alpha$*, then* $\Gamma \vdash_{\mathcal{C}(\to)} F : S(\langle [v_1], [w_1] \rangle)$ *for some substitution* S *and combinatory term* F *such that* $\text{size}(F) < \mathfrak{n}$*.*

By construction, elements $(v, w) \in \mathsf{PCP}_n$ are associated with terms of type $\langle [v], [w] \rangle$ (Lemma 31).

Lemma 31 ([35, Lemma 25])**.** *Let* $v, w \in \{\mathsf{a}, \mathsf{b}\}^*$*. If* $(v, w) \in \mathsf{PCP}_n$*, then* $\Gamma \vdash_{\mathcal{C}(\to)} x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n} : \langle [v], [w] \rangle$ *for some index sequence* $i_1 \ldots i_n$*.*

Finally, we obtain the following key Lemma 32 which relates membership of (v_1, w_1) in some PCP_n and derivability of $\alpha \to \beta \to \alpha$ from the axioms $\lceil \Gamma \rceil$ concluding the proof of Theorem 7.

Lemma 32. *We have* $\lceil \Gamma \rceil \vdash_{\mathcal{H}} \alpha \to \beta \to \alpha$ *iff* $(v_1, w_1) \in \mathsf{PCP}_n$ *for some* $n \geq 0$*.*

Proof. $\Longrightarrow$: Assume $\lceil \Gamma \rceil \vdash_{\mathcal{H}} \alpha \to \beta \to \alpha$. By Lemma 30 we have $\Gamma \vdash_{\mathcal{C}(\to)} F : S(\langle [v_1], [w_1] \rangle)$ for some substitution S and combinatory term F with $\text{size}(F) < \mathfrak{n}$. By Lemma 29 we have $F = x \succ y_{i_1} \succ y_{i_2} \succ \ldots \succ y_{i_n}$ for some index sequence $i_1 \ldots i_n$. Finally, by Lemma 28 we have $(v_1, w_1) \in \mathsf{PCP}_n$.

$\Longleftarrow$: Assume $(v_1, w_1) \in \mathsf{PCP}_n$. By Lemma 31 we have $\Gamma \vdash_{\mathcal{C}(\to)} F : \langle [v_1], [w_1] \rangle$ for some term F. Using an appropriate substitution, we obtain $\Gamma \vdash_{\mathcal{C}(\to)} z\, F : \alpha \to \beta \to \alpha$. □

In sum, Theorem 7 is shown applying the above Lemma 32 to the construction in Corollary 4. For formal proofs of the above Lemmas in the Lean proof assistant see [35].

It is noteworthy that the condition $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \dots n$ in Theorem 7 is decidable in linear time [35, Section 3]. The key observation is that any formula derivable from $\alpha \to \beta \to \alpha$ is a prefixed instance of $\alpha \to \beta \to \alpha$ (Lemma 33). This contrasts PSPACE-completeness of intuitionistic implicational provability [62] and undecidability of relativized derivability in logic fragments "weaker" that $\alpha \to \beta \to \alpha$ (Theorem 7).

Lemma 33 ([35, Lemma 27])**.**
$[\{\alpha \to \beta \to \alpha\}]_{\mathcal{H}} = \{\sigma_1 \to \ldots \to \sigma_n \to \tau \to \sigma_n \mid n \geq 1 \text{ and } \sigma_1, \ldots, \sigma_n, \tau \in \mathbb{T}^{\to}\}$

Proof. (Sketch)

$\supseteq$: n-fold use of Lemma 26 starting with $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_n \to \tau \to \sigma_n$.

$\subseteq$: Assume $\{x : \alpha \to \beta \to \alpha\} \vdash_{\mathcal{C}(\to)} F : \sigma$ such that size(F) is minimal. The claim is shown by induction on size(F) using (Lemma 21). □

It remains open, whether there is a fixed principal basis derivable from the axiom $\alpha \to \beta \to \alpha$ that exposes an undecidable derivability problem (Conjecture 4).

Conjecture 4. *There exist principal axioms $\sigma_1, \ldots, \sigma_n$ such that we have $\{\alpha \to \beta \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \dots n$ and it is undecidable whether $\{\sigma_1, \ldots, \sigma_n\} \vdash_{\mathcal{H}} \tau$ holds given a formula τ.*

Our reasoning is as follows. It is possible to axiomatize arbitrary PCP instances using "weak", principal axioms. We may consider a particular PCP instance that encodes an universal Turing machine. The claim may be shown by representing Turing machine inputs via the given type τ.

In the Context of Synthesis

In the above reasoning the library components would correspond to building blocks of an universal Turing machine specified by simple types. We conjecture that typable applicative compositions of those components would correspond to (partial) runs of this machine.

3.1.3 Recognizing Axiomatizations of $\alpha \to \alpha$

In this section, we record that axiomatizations of the Hilbert-style calculus containing only the axiom $\alpha \to \alpha$ are recognizable in linear time. The key observation is that one cannot meaningfully compose axioms that are instances of $\alpha \to \alpha$. Therefore, the only derivable formulae are instances of the given axioms (Lemma 34).

Lemma 34 ([35, Lemma 28])**.** *Given $\sigma_1, \ldots, \sigma_n \in \mathbb{T}^{\to}$ we have*

$$[\{\sigma_1 \to \sigma_1, \ldots, \sigma_n \to \sigma_n\}]_{\mathcal{H}} = \bigcup_{i=1}^{n} \{S(\sigma_i \to \sigma_i) \mid S \text{ is a substitution}\}$$

Proof. (Sketch)

$\supseteq$: holds by instantiation of $\sigma_i \to \sigma_i$ for $i = 1 \ldots n$.

$\subseteq$: Let $\Gamma = \{x_1 : \sigma_1 \to \sigma_1, \ldots, x_n : \sigma_n \to \sigma_n\}$. Assume $\Gamma \vdash_{\mathcal{C}(\to)} F : \sigma$ such that size(F) is minimal. The claim is shown by case analysis using (Lemma 21) and minimality of size(F). □

Corollary 6. $[\{\alpha \to \alpha\}]_{\mathcal{H}} = \{\tau \to \tau \mid \tau \in \mathbb{T}^{\to}\}$.

Lemma 35. *If $[\Delta]_{\mathcal{H}} = [\{\alpha \to \alpha\}]_{\mathcal{H}}$, then $\beta \to \beta \in \Delta$ for some $\beta \in \mathbb{V}$.*

Proof. Since $[\{\alpha \to \alpha\}]_{\mathcal{H}} \supseteq [\Delta]_{\mathcal{H}}$ implies $[\{\alpha \to \alpha\}]_{\mathcal{H}} \supseteq \Delta$ we have $\Delta = \{\sigma_1 \to \sigma_1, \ldots, \sigma_n \to \sigma_n\}$ for some $\sigma_1, \ldots, \sigma_n \in \mathbb{T}^{\to}$ by Corollary 6. By Lemma 34 we obtain $[\Delta]_{\mathcal{H}} = \bigcup_{i=1}^{n} \{S(\sigma_i \to \sigma_i) \mid S \text{ is a substitution}\}$. Due to $[\{\alpha \to \alpha\}]_{\mathcal{H}} \subseteq [\Delta]_{\mathcal{H}}$ we obtain $\alpha \to \alpha = S(\sigma_i \to \sigma_i)$ for some $i \in \{1, \ldots, n\}$ and some substitution S, which holds iff $\sigma_i \to \sigma_i = \beta \to \beta$ for some $\beta \in \mathbb{V}$. □

Corollary 7. *We have $[\Delta]_{\mathcal{H}} = [\{\alpha \to \alpha\}]_{\mathcal{H}}$ iff $\Delta \subseteq \{\sigma \to \sigma \mid \sigma \in \mathbb{T}^{\to}\}$ and $\beta \to \beta \in \Delta$ for some $\beta \in \mathbb{V}$.*

As a result of Corollary 7, recognizing axiomatizations of $\alpha \to \alpha$ is decidable in linear time.

Theorem 8. *Given principal axioms $\sigma_1, \ldots, \sigma_n$ such that $\{\alpha \to \alpha\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \ldots n$, it is decidable in linear time whether $\{\sigma_1, \ldots, \sigma_n\} \vdash_{\mathcal{H}} \alpha \to \alpha$.*

3.1.4 Recognizing Axiomatizations of $\alpha \to \beta \to \beta$

In this section, we extend linear time recognizability to axiomatizations of the Hilbert-style calculus containing only the axiom $\alpha \to \beta \to \beta$. This is surprising in light of Theorem 7 showing undecidability in case of $\alpha \to \beta \to \alpha$. However, similarly to $\alpha \to \alpha$, meaningful logical compositions of instances of $\alpha \to \beta \to \beta$ are limited (Lemma 36).

Lemma 36 ([35, Lemma 32])**.**
$[\{\alpha \to \beta \to \beta\}]_{\mathcal{H}} = \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}^{\to}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}^{\to}\}$

Proof. (Sketch)

$\supseteq$: Instantiation resp. derivability of $\alpha \to \alpha$.

$\subseteq$: Assume $\{x : \alpha \to \beta \to \beta\} \vdash_{\mathcal{C}(\to)} F : \sigma$ such that size(F) is minimal. The claim is shown by case analysis using (Lemma 21) and minimality of size(F). □

Using the above Lemma 36, we can characterize axiomatizations of $\alpha \to \beta \to \beta$ syntactically (Lemma 37).

Lemma 37. *We have* $[\Delta]_{\mathcal{H}} = [\{\alpha \to \beta \to \beta\}]_{\mathcal{H}}$ *iff* $\gamma \to \delta \to \delta \in \Delta$ *for some* $\gamma, \delta \in \mathbb{V}$ *and* $\Delta \subseteq \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}^{\to}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}^{\to}\}$.

Proof. $\Longleftarrow$: $\sigma \to \tau \to \tau$ and $\tau \to \tau$ are derivable from $\gamma \to \delta \to \delta$ for any $\sigma, \tau \in \mathbb{T}^{\to}$.

$\Longrightarrow$: Due to $[\{\alpha \to \beta \to \beta\}]_{\mathcal{H}} \supseteq \Delta$ by Lemma 36 we have that $\Delta \subseteq \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}^{\to}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}^{\to}\}$. From $[\{\alpha \to \beta \to \beta\}]_{\mathcal{H}} \subseteq [\Delta]_{\mathcal{H}}$ we obtain $\Delta \vdash_{\mathcal{H}} \alpha \to \beta \to \beta$. By case analysis (similar to the proof of Lemma 36) the minimal derivation of $\Delta \vdash_{\mathcal{H}} \alpha \to \beta \to \beta$ is an instantiation of some $\sigma \to \tau \to \tau \in \Delta$, i.e. $\alpha \to \beta \to \beta = S(\sigma \to \tau \to \tau)$ for some substitution S. Therefore, $\sigma \to \tau \to \tau = \gamma \to \delta \to \delta$ for some $\gamma, \delta \in \mathbb{V}$. □

As a result of the above Lemma 37, recognizing axiomatizations of $\alpha \to \beta \to \beta$ is decidable in linear time.

Theorem 9. *Given principal axioms* $\sigma_1, \ldots, \sigma_n$ *such that* $\{\alpha \to \beta \to \beta\} \vdash_{\mathcal{H}} \sigma_i$ *for* $i = 1 \ldots n$*, it is decidable in linear time whether* $\{\sigma_1, \ldots, \sigma_n\} \vdash_{\mathcal{H}} \alpha \to \beta \to \beta$.

One reason why recognizing axiomatizations of $\alpha \to \alpha$ and $\alpha \to \beta \to \beta$ is trivial is that the set of minimal proofs in the corresponding calculi is finite, which is not the case for $\alpha \to \beta \to \alpha$.

In the Context of Synthesis

Contrarily to $\alpha \to \beta \to \alpha$, formulae "below" $\alpha \to \beta \to \beta$ (or, $\alpha \to \alpha$) do not seem to constitute an expressive specification language.

Concluding Remarks

The presented construction has two distinct benefits compared to the reduction from the halting problem for two-counter automata to inhabitation in simply typed combinatory logic in [54] with regard to composition synthesis. First, we use simple types as the underlying type language whereas in [54] the construction additionally requires type constants that are not subject to parametricity. Second, the restriction to principal axioms directly establishes a relationship to code fragments whereas in [54] the axioms are not intuitionistic theorems.

Considering axiomatizations that are principal provides an additional twist to the Curry-Howard isomorphism. In particular, it is insightful to inspect compositions of principal inhabitants of axioms $\sigma_1, \ldots, \sigma_l$ constructed in Section 3.1.2. On the one hand, for particular PCP instances solvability is derived from the axioms $\sigma_1, \ldots, \sigma_l$ logically. On the other hand, in [35, Section 3] the constructed λ-term (actually, Haskell program) not only corresponds to the proof that a given PCP instance is solvable, but actually constructs the solution computationally. However, it is unclear whether this phenomenon is systematic or coincidental.

One of the main contributions in [35] is the formalization of the reduction using the Lean proof assistant under the banner of "type theory inside type theory". We hope that this reduction can be embedded in the larger framework of computational reductions in Coq [41] already containing a collection of formalized reductions that are used in undecidability results.

It remains open whether, similarly to [60], there is a fixed principal basis "below" $\alpha \to \beta \to \alpha$ that exposes an undecidable derivability problem. If so, it would be interesting to inspect the principal inhabitants of the basis axioms.

3.2 Combinatory Logic with Intersection Types

As we have seen in Section 3.1, even in simple types, inhabitation in combinatory logic is undecidable. This remains true [27], even for the fixed type environment of S, K with corresponding types, if we use intersection types as the underlying type language and add corresponding intersection introduction and elimination rules. Therefore, several restrictions of combinatory logic with intersection types have been considered [55, 30]. One such restriction is the *bounded combinatory logic* [30] in which instantiation is restricted to types having a certain *level*. The level of a type is the depth of its syntax tree wrt. the arrow type constructor. If type instantiation is restricted to level at most k, then the decision problem of type inhabitation in combinatory logic with intersection types is $(k+2)$-ExpTime-complete [30].

In this section we refine the notion of level into the combination of functional *order* and functional *arity*. As a result, we obtain a fine grained analysis of inhabitation complexity showing that functional order (and not functional arity) contributes to the iterated exponential complexity.

Intersection types with constants $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ (Definition 34) serve as the type language of bounded combinatory logic.

Definition 34 (Intersection Types with Constants, $\mathbb{T}^{\cap}_{\mathbb{C}_0}$)**.**

$$\mathbb{T}^{\cap}_{\mathbb{C}_0} \ni \sigma, \tau ::= \alpha \mid a \mid \omega \mid \sigma \to \tau \mid \sigma \cap \tau$$
$$\textit{where } \alpha \textit{ ranges over type variables } \mathbb{V} \textit{ and}$$
$$a \textit{ ranges over type constants } \mathbb{C}_0$$

As is usual, $\to$ associates to the right, and $\cap$ binds more strongly than $\to$. A type $\sigma \cap \tau$ is said to have σ and τ as *components*.

Types in $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ differ from types in $\mathbb{T}^{\cap}_{\{!\}}$ in three aspects. First, $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ encompasses type constants a motivated by existence of certain primitive types that are not subject to parametricity in existing programming languages. Second, the special constant ω represents the universal type and corresponds to the empty intersection in $\mathbb{T}^{\cap}_{\{!\}}$. Third, $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ does not stratify types by allowing intersections on the right-hand side of the arrow. This allows for an exponentially more concise presentation of types as we introduce the *intersection type subtyping* (Definition 35) relation.

Definition 35 (Intersection Type Subtyping [3], $\leq$)**.** *The relation $\leq$ is the least preorder (reflexive and transitive relation) over $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ such that*

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau,$$
$$(\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2,$$
$$\textit{if } \sigma \leq \tau_1 \textit{ and } \sigma \leq \tau_2 \textit{ then } \sigma \leq \tau_1 \cap \tau_2,$$
$$\textit{if } \sigma_2 \leq \sigma_1 \textit{ and } \tau_1 \leq \tau_2 \textit{ then } \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$$

Type equality, written $\sigma = \tau$, holds when $\sigma \leq \tau$ and $\tau \leq \sigma$ hold, thereby making $\leq$ a partial order over $\mathbb{T}^{\cap}_{\mathbb{C}_0}$. To avoid confusion, we use $\equiv$ for syntactic identity. Observe that in [3] types do not include type constants. Since subtyping does not distinguish between type constants and type variables, we can treat them uniformly. Under above type equality intersection is associative, commutative and

idempotent. Therefore, we write $\bigcap_{i=1}^{n} \sigma_i$ for corresponding nested intersections, and for ω if $n = 0$.

Combinatory logic with intersection types (Definition 36) is an extension of simply typed combinatory logic (Definition 31) while combinatory terms (Definition 29) remain the underlying term language. The additional rules ($\cap$I) and ($\leq$) serve as introduction and elimination rules for the intersection type constructor.

Definition 36 (Combinatory Logic with Intersection Types, $\vdash_{\mathcal{C}(\cap)}$)**.**

$$\frac{S \text{ is a substitution}}{\Gamma, x : \sigma \vdash_{\mathcal{C}(\cap)} x : S(\sigma)}\ (\text{Ax}) \qquad \frac{\Gamma \vdash_{\mathcal{C}(\cap)} F : \sigma \to \tau \qquad \Gamma \vdash_{\mathcal{C}(\cap)} G : \sigma}{\Gamma \vdash_{\mathcal{C}(\cap)} F\ G : \tau}\ (\to\text{E})$$

$$\frac{\Gamma \vdash_{\mathcal{C}(\cap)} F : \sigma \qquad \Gamma \vdash_{\mathcal{C}(\cap)} F : \tau}{\Gamma \vdash_{\mathcal{C}(\cap)} F : \sigma \cap \tau}\ (\cap\text{I}) \qquad \frac{\Gamma \vdash_{\mathcal{C}(\cap)} F : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash_{\mathcal{C}(\cap)} F : \tau}\ (\leq)$$

In the Context of Synthesis

Similarly to program synthesis in typed λ-calculi (Chapter 2), intersection types provide a richer specification language in comparison to simple types. Conceivably, the intersection type specification of an universal Turing machine M could be

$\tau \equiv \texttt{TuringMachine} \cap (\texttt{TuringMachine} \to \texttt{String} \to \texttt{Integer} \to \texttt{Bool})$

Using intersection type subtyping we can derive

$\{M : \tau\} \vdash_{\mathcal{C}(\cap)} M : \texttt{TuringMachine}$ and
$\{M : \tau\} \vdash_{\mathcal{C}(\cap)} M : \texttt{TuringMachine} \to \texttt{String} \to \texttt{Integer} \to \texttt{Bool}$

Therefore, $\{M : \tau\}$ states two properties. First, M is itself a Turing machine. Second, given a Turing machine, an input string, and some number of steps, M returns either true of false (for example depending on the simulated behavior for the given input after the given number of steps). Such a specification is neither in scope of simple types nor do the two intersection components refine (in the sense of [52]) a simple type.

Similarly to the simply typed scenario, combinatory logic with intersection types where the type environment contains correspondingly typed S and K combinators is directly related to λ-calculus with intersection types [27], in particular the Barendregt-Coppo-Dezani type assignment system [3], for which inhabitation is undecidable [66].

In search of fragments exposing decidable inhabitation we explore restrictions of ($\vdash_{\mathcal{C}(\cap)}$). In the following, we restrict instantiation wrt. functional *order* (Definition 38) and functional *arity* (Definition 39). The notions of order and arity refine the notion of level (Definition 37) by inspecting nesting of the arrow type constructor to the left and to the right separately.

Definition 37 (Level)**.**

$$\begin{aligned}\operatorname{level}(\alpha) = \operatorname{level}(a) = \operatorname{level}(\omega) &= 0\\ \operatorname{level}(\sigma \to \tau) &= \max(1 + \operatorname{level}(\sigma), 1 + \operatorname{level}(\tau))\\ \operatorname{level}(\sigma \cap \tau) &= \max(\operatorname{level}(\sigma), \operatorname{level}(\tau))\end{aligned}$$

Definition 38 (Order)**.**

$$\begin{aligned}\operatorname{order}(\alpha) = \operatorname{order}(a) = \operatorname{order}(\omega) &= 0\\ \operatorname{order}(\sigma \to \tau) &= \max(1 + \operatorname{order}(\sigma), \operatorname{order}(\tau))\\ \operatorname{order}(\sigma \cap \tau) &= \max(\operatorname{order}(\sigma), \operatorname{order}(\tau))\end{aligned}$$

Definition 39 (Arity)**.**

$$\begin{aligned}\operatorname{arity}(\alpha) = \operatorname{arity}(a) = \operatorname{arity}(\omega) &= 0\\ \operatorname{arity}(\sigma \to \tau) &= \max(\operatorname{arity}(\sigma), 1 + \operatorname{arity}(\tau))\\ \operatorname{arity}(\sigma \cap \tau) &= \max(\operatorname{arity}(\sigma), \operatorname{arity}(\tau))\end{aligned}$$

The notion of order and arity is tacitly extended to substitutions as follows

$$\operatorname{order}(S) = \max\{\operatorname{order}(S(\alpha)) \mid \alpha \in \mathbb{V}\} \quad \operatorname{arity}(S) = \max\{\operatorname{arity}(S(\alpha)) \mid \alpha \in \mathbb{V}\}$$

The rules of *bounded combinatory logic* ($\vdash_{(o,a)}$) (cf. [30, Figure 1]) are given in the following Definition 40 where the parameter o limits instantiation order and the parameter a limits instantiation arity.

Definition 40 (Bounded Combinatory Logic, $\vdash_{(o,a)}$)**.**

$$\frac{S \text{ is a substitution} \quad \operatorname{order}(S) \leq o \quad \operatorname{arity}(S) \leq a}{\Gamma, x : \sigma \vdash_{(o,a)} x : S(\sigma)}\ (\mathrm{Ax})$$

$$\frac{\Gamma \vdash_{(o,a)} F : \sigma \to \tau \qquad \Gamma \vdash_{(o,a)} G : \sigma}{\Gamma \vdash_{(o,a)} F\ G : \tau}\ (\to\mathrm{E})$$

$$\frac{\Gamma \vdash_{(o,a)} F : \sigma \qquad \Gamma \vdash_{(o,a)} F : \tau}{\Gamma \vdash_{(o,a)} F : \sigma \cap \tau}\ (\cap\mathrm{I}) \qquad \frac{\Gamma \vdash_{(o,a)} F : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash_{(o,a)} F : \tau}\ (\leq)$$

Naturally, the decision problem of inhabitation in ($\vdash_{(o,a)}$) (Problem 12) amounts to existence of a combinatory term typable by the given type in the given environment under order and arity restrictions.

Problem 12 (Inhabitation in ($\vdash_{(o,a)}$), $\Gamma \vdash_{(o,a)} ? : \tau$)**.** *Let o and a be two natural numbers. Given a type environment Γ and a type τ, is there a combinatory term F such that $\Gamma \vdash_{(o,a)} F : \tau$ is derivable?*

Example 41. *Let $o = a = 0$, and*

$$\begin{aligned}\sigma &\equiv (0 \to 1) \cap (1 \to 0)\\ \tau &\equiv (0 \to 0) \cap (1 \to 1)\\ \rho_0 &\equiv (0 \to 1) \to (1 \to 0) \to (0 \to 0)\\ \rho_1 &\equiv (1 \to 0) \to (0 \to 1) \to (1 \to 1)\\ \Gamma &= \{s : \sigma, b : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma\}\end{aligned}$$

The question $\Gamma \vdash_{(o,a)} ? : \tau$ *has the combinatory term* $b\,s\,s$ *as a positive answer, shown by the following derivation*

$$\dfrac{\dfrac{\mathcal{D} \triangleright \Gamma \vdash_{(o,a)} b : \sigma \to \sigma \to \tau \qquad \dfrac{}{\Gamma \vdash_{(o,a)} s : \sigma}\,(\mathrm{Ax})}{\Gamma \vdash_{(o,a)} b\,s : \sigma \to \tau}\,(\to\mathrm{E}) \qquad \dfrac{}{\Gamma \vdash_{(o,a)} s : \sigma}\,(\mathrm{Ax})}{\Gamma \vdash_{(o,a)} b\,s\,s : \tau}\,(\to\mathrm{E})$$

where the derivation $\mathcal{D}$ *is given by*

$$\dfrac{\dfrac{\dfrac{\{\alpha \mapsto 0, \beta \mapsto 1, \gamma \mapsto 0\}}{\Gamma \vdash_{(o,a)} b : \rho_0}\,(\mathrm{Ax}) \qquad \dfrac{\{\alpha \mapsto 1, \beta \mapsto 0, \gamma \mapsto 1\}}{\Gamma \vdash_{(o,a)} b : \rho_1}\,(\mathrm{Ax})}{\Gamma \vdash_{(o,a)} b : \rho_0 \cap \rho_1}\,(\cap\mathrm{I}) \qquad (\star)}{\Gamma \vdash_{(o,a)} b : \sigma \to \sigma \to \tau}\,(\leq)$$

and $(\star)$ *is the side condition* $\rho_0 \cap \rho_1 \leq \sigma \to \sigma \to \tau$.

In the Context of Synthesis

In the above Example 41 the type σ (resp. τ) specifies the behavior of the successor (resp. identity) function in a binary field. Complementary, the schematic specification $(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ describes the behavior of functional composition that can be instantiated to ρ_0 and ρ_1 using substitutions of order and arity 0. Therefore, the question $\Gamma \vdash_{(o,a)} ? : \tau$ corresponds to a search for compositions of the binary successor function that result in the binary identity function.

Section Outline The remainder of this section inspects combinatory logic with intersection types where instantiation is bounded by functional order o and functional arity $a \geq 1$. Specifically, in Section 3.2.1 inhabitation in this type system is shown to be $(o + 2)$-ExpTime-complete. The upper bound (Theorem 10) is shown by adapting the alternating decision procedure from [30]. The lower bound (Theorem 11) is shown by observing that the reduction from alternating space Turing machines in [30] already shows the stronger result. Additionally, in Section 3.2.2 we consider bounded combinatory logic without the intersection introduction rule for which inhabitation is conjectured to be of lower complexity.

Authorship Statement The inspection of inhabitation complexity in the combinatory logic with intersection types where instantiation is bounded by functional order and functional arity is attributed to the author, and is not part of previously published work.

3.2.1 Inhabitation with Bounded Order and Arity

Inhabitation in bounded combinatory logic with instantiation restricted to level at most k is known to be $(k+2)$-ExpTime-complete [30, Theorem 24].

In this section we refine this result showing that iterated exponential complexity depends on functional order and not functional arity. In particular, we show that inhabitation in $(\vdash_{(o,a)})$ (Problem 12) is in $(o+2)$-ExpTime, and it is $(o+2)$-ExpTime-hard even for the fixed functional arity of 1.

Let $\exp_k : \mathbb{N} \to \mathbb{N}$ be the iterated exponential function, defined as

$$\exp_0(n) = n \qquad \exp_{k+1}(n) = 2^{\exp_k(n)}$$

We will use the following inequality in order to simplify terms containing iterated exponentiation.

Lemma 38. *For $k \geq 0$, $m \geq 1$, and $n \geq 2$ we have*

$$m \cdot \exp_{k+1}(n) \leq \exp_{k+1}(n + \frac{m}{2^k})$$

Proof. Induction on k using the inequality $y + 1 \leq 2^y$ for $y \in \mathbb{N}$. □

Restricting intersection types syntactically it is wrong to assume that intersection type equality (defined via subtyping) preserves syntax-oriented measures. However, intersection type subtyping is non-structural (see also Section 3.3.3) and generally does neither preserve functional order nor functional arity (Example 42).

Example 42. *Let $\sigma \equiv \omega \to \alpha$ and $\tau \equiv \sigma \cap ((\alpha \to \alpha \to \alpha) \to \alpha)$. Since $\alpha \to \alpha \to \alpha \leq \omega$ we have $\sigma \leq (\alpha \to \alpha \to \alpha) \to \alpha$. Therefore, we have $\sigma = \tau$. However,* $\mathrm{order}(\sigma) = 1 \neq 2 = \mathrm{order}(\tau)$ *and* $\mathrm{arity}(\sigma) = 1 \neq 2 = \mathrm{arity}(\tau)$.

Fortunately, the notion of *organized types* (Definition 42), which is based on a notion of *paths* (Definition 41), respects order and arity (Lemma 40) under type *organization* (Lemma 39).

Definition 41 (Paths, $\mathbb{P}^{\cap}_{\mathbb{C}_0}$)**.** $\mathbb{P}^{\cap}_{\mathbb{C}_0} \ni \pi ::= \alpha \mid a \mid \sigma \to \pi$

Definition 42 (Organized Type)**.** *We say a type $\sigma \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$ is* organized*, if $\sigma \equiv \bigcap_{i=1}^{n} \pi_i$ for some $n \geq 0$ and paths $\pi_1 \dots \pi_n$.*

Lemma 39 (Type Organization [30, Lemma 1])**.** *For any type $\sigma \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$, an organized type $\overline{\sigma} \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$ such that $\sigma = \overline{\sigma}$ can be computed in polynomial time by*

$$\overline{\alpha} \equiv \alpha \quad \overline{a} \equiv a \quad \overline{\omega} \equiv \omega \quad \overline{\sigma \cap \tau} \equiv \overline{\sigma} \cap \overline{\tau} \quad \overline{\sigma \to \tau} \equiv \bigcap_{i \in I} (\sigma \to \pi_i) \text{ where } \overline{\tau} \equiv \bigcap_{i \in I} \pi_i$$

Lemma 40. *For any type $\sigma \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$ we have* $\mathrm{order}(\sigma) = \mathrm{order}(\overline{\sigma})$ *and* $\mathrm{arity}(\sigma) = \mathrm{arity}(\overline{\sigma})$.

Upper Bound

The exact methodology based on the alternating decision procedure in [30, Section 3] applies for the refined analysis. The only difference is that the alternating $(k+1)$-ExpSpace inhabitation Algorithm [30, Figure 2] has to consider the set of substitutions of bounded order and arity (instead of bounded level). Analogous to [30, Lemma 13], we need to inspect the number and size of types over n constants and variables having bounded order and arity. Since order and arity are preserved by type organization (Lemma 40), it suffices to consider only organized types.

Let $\mathbb{A} \subseteq \mathbb{C}_0 \cup \mathbb{V}$ be a non-empty, finite set of *atoms*. We are interested in in the number and size of types $\sigma \in \mathbb{T}^{\cap}_{(\mathbb{A},o,a)}$ where σ is organized, σ contains constants and variables from $\mathbb{A}$, $\mathrm{order}(\sigma) \leq o$, and $\mathrm{arity}(\sigma) \leq a$. In particular, we have

$$\begin{aligned}
\mathbb{T}^{\cap}_{(\mathbb{A},o,a)} &= \{ \bigcap_{\pi \in P} \pi \mid P \subseteq \mathbb{P}^{\cap}_{(\mathbb{A},o,a)} \} \subseteq \mathbb{T}^{\cap}_{\mathbb{C}_0} \\
\mathbb{P}^{\cap}_{(\mathbb{A},o,a)} &= \{ \sigma_1 \to \ldots \to \sigma_l \to \epsilon \mid \epsilon \in \mathbb{A}, l \leq a, \sigma_i \in \mathbb{T}^{\cap}_{(\mathbb{A},o-1,a+1-i)} \text{ for } i = 1 \ldots l \} \\
&\subseteq \mathbb{P}^{\cap}_{\mathbb{C}_0}
\end{aligned}$$

First, we show that the cardinality of $\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}$ modulo subtyping equality is bounded by an $(o+1)$-times iterated exponential (Lemma 41).

Lemma 41. $|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}| \leq \exp_{o+1}((a+3)(|\mathbb{A}|+2))$.

Proof. For brevity, let $m = |\mathbb{A}|$. For $o = 0$ or $a = 0$ the set $\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}$ contains only intersections of atoms. Therefore, we have

$$|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}| = |\{ \bigcap_{\epsilon \in P} \epsilon \mid P \subseteq \mathbb{A} \}| = 2^m \leq \exp_{o+1}((a+3)(m+2))$$

Additionally, for $o \geq 1$ and $a \geq 1$ we have

$$|\mathbb{P}^{\cap}_{(\mathbb{A},o,a)}| \leq \sum_{l=0}^{a} |\mathbb{A}| \cdot |\mathbb{T}^{\cap}_{(\mathbb{A},o-1,a)}|^l \leq 2m |\mathbb{T}^{\cap}_{(\mathbb{A},o-1,a)}|^a \qquad (\star)$$

For $o, a \geq 1$ we show by induction on o the following stronger claim

$$|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}| \leq \exp_{o+1}((a+1)m + \sum_{i=0}^{o-2} \frac{a+m}{2^i})$$

Basis Step ($o = 1$):

$$\begin{aligned}
\mathbb{T}^{\cap}_{(\mathbb{A},1,a)}| = |\{ \bigcap_{\pi \in P} \pi \mid P \subseteq \mathbb{P}^{\cap}_{(\mathbb{A},1,a)} \}| &\overset{(\star)}{\leq} \exp_1(2m |\mathbb{T}^{\cap}_{(\mathbb{A},0,a)}|^a) \\
&\leq \exp_1(2m(2^m)^a) = \exp_2(ma + \log_2(2m)) \leq \exp_2((a+1)m)
\end{aligned}$$

Inductive Step:

$$\begin{aligned}
|\mathbb{T}^{\cap}_{(\mathbb{A},o+1,a)}| &= |\{\bigcap_{\pi\in P}\pi \mid P \subseteq \mathbb{P}^{\cap}_{(\mathbb{A},o+1,a)}\}| \overset{(\star)}{\leq} \exp_1(2m|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}|^a) \\
&\overset{\text{IH}}{\leq} \exp_1(2m(\exp_{o+1}((a+1)m + \sum_{i=0}^{o-2}\frac{a+m}{2^i}))^a) \\
&= \exp_1(2m(\exp_1(a\exp_o((a+1)m + \sum_{i=0}^{o-2}\frac{a+m}{2^i}))) \\
&\overset{\text{Lem. 38}}{\leq} \exp_{o+2}((a+1)m + \sum_{i=0}^{o-2}\frac{a+m}{2^i} + \frac{a}{2^{o-1}} + \frac{2m}{2^o}) \\
&= \exp_{o+2}((a+1)m + \sum_{i=0}^{o-1}\frac{a+m}{2^i})
\end{aligned}$$

Overall, we obtain

$$\begin{aligned}
|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}| &\leq \exp_{o+1}((a+1)m + \sum_{i=0}^{o-2}\frac{a+m}{2^i}) \\
&\leq \exp_{o+1}((a+1)m + 2(a+m)) \leq \exp_{o+1}((a+3)(m+2)) \qquad \square
\end{aligned}$$

Next, we are interested in the size of types in $\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}$. Let $|\tau|$ denote the number of nodes in the syntax tree of τ, and let $\text{size}(o,a)$ denote the maximal size of minimal representations (under subtype equality) of types with bounded order o and bounded arity a over atoms $\mathbb{A}$

$$\text{size}(o,a) = \max\{\min\{|\tau| \mid \tau = \sigma\} \mid \sigma \in \mathbb{T}^{\cap}_{(\mathbb{A},o,a)}\}$$

The following Lemma 42 shows that $\text{size}(o,a)$ is bounded by an o-times iterated exponential.

Lemma 42. $\text{size}(o,a) \leq \exp_o((a+3)(|\mathbb{A}|+3))$.

Proof. For brevity, let $m = |\mathbb{A}|$. For $o = 0$ or $a = 0$ the set $\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}$ contains only intersections of atoms. Therefore, $\text{size}(0,a) = \text{size}(o,0) \leq 2m$.

For $o, a \geq 1$ we proceed by induction on o. By construction we have

$$\begin{aligned}
\text{size}(o,a) &\leq |\mathbb{P}^{\cap}_{(\mathbb{A},o,a)}| \cdot (a(\text{size}(o-1,a)+1)+1) \\
&\leq 2m|\mathbb{T}^{\cap}_{(\mathbb{A},o-1,a)}|^a \cdot a(\text{size}(o-1,a)+2)
\end{aligned}$$

Additionally, we have $\log_2(n+1) \leq n$ for $n \in \mathbb{N}$.

Basis Step $(o = 1)$**:**

$$\begin{aligned}
\text{size}(1,a) &\leq 2m|\mathbb{T}^{\cap}_{(\mathbb{A},0,a)}|^a \cdot a(\text{size}(0,a)+2) = 2m2^{am} \cdot a(2m+2) \\
&= \exp_1(am + \log_2(2am(2(m+1)))) \leq \exp_1((a+3)(m+3))
\end{aligned}$$

Inductive Step:

$$\begin{aligned}
\text{size}(o+1,a) &\leq 2m|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}|^a \cdot a(\text{size}(o,a)+2) \\
&\overset{\text{IH}}{\leq} 2am|\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}|^a(\exp_o((a+3)(m+3))+2) \\
&\leq |\mathbb{T}^{\cap}_{(\mathbb{A},o,a)}|^a(\exp_{o+1}(a+2+m+2+1+1+a+(m-1))) \\
&\overset{\text{Lem. 41}}{\leq} \left((\exp_{o+1}((a+3)(m+2)))^a\right)^2 \\
&\leq \exp_{o+1}((a+3)(m+2)+a+1) \\
&\leq \exp_{o+1}((a+3)(m+3))
\end{aligned}$$

□

Finally, we can follow the proof of [30, Theorem 14] bounding inhabitant search space (Theorem 10).

Theorem 10. *Inhabitation in* $(\vdash_{(o,a)})$ *(Problem 12) is in* $(o+2)$-ExpTime.

Proof. More precisely, inhabitation in $(\vdash_{(o,a)})$ is in $\text{DTime}(\exp_{o+2}(p(a,n)))$ for some bivariate polynomial p. We use the alternating inhabitation procedure [30, Figure 2] adjusting the number (Lemma 41) and size (Lemma 42) of types modulo subtyping bounded by order and arity. The result follows analogously to the proof of [30, Theorem 14] by the following relationships

$$\begin{aligned}
\text{ASpace}(f(n)) &= \text{DTime}(2^{\mathcal{O}(f(n))}) \\
\text{DTime}(2^{\mathcal{O}(\exp_m(f(n)))}) &\subseteq \text{DTime}(\exp_{m+1}(\mathcal{O}(f(n))))
\end{aligned}$$

□

Let us conclude the upper bound discussion with a more high-level argument. In [55] it is shown that inhabitation in combinatory logic with intersection types without instantiation is ExpTime-complete. We can internalize instantiation in bounded combinatory logic by a priori intersecting all instances (finite in size by Lemma 41 and Lemma 42) of types containing type variables. Therefore, by blowing up the input we end up in a decidable, non-schematic scenario.

Lower Bound

The lower bound construction for inhabitation in level-bounded combinatory logic [30, Section 5] directly reduces alternating space Turing machine computation to inhabitant search. Upon closer inspection, this construction is adequate to show that inhabitation in $(\vdash_{(o,1)})$ (Problem 12) is $(o+2)$-ExpTime-hard. Key to the reduction are the following three aspects, which we examine in detail.

First, an encoding of quaternary predicates $F(\tau_1, \tau_2, \tau_3, \tau_4)$ which ensures that only predicate arguments are in range of level-bounded substitutions and not predicate expressions themselves.

$$F^{[1]} \equiv F \qquad F^{[i+1]} \equiv F^{[i]} \to F \qquad \Omega_\tau \equiv (\tau \to \tau) \to \tau \to \tau$$
$$F(\tau_1, \tau_2, \tau_3, \tau_4) \equiv (((F^{[k]} \to \Omega_{\tau_1}) \to \Omega_{\tau_2}) \to \Omega_{\tau_3}) \to \Omega_{\tau_4}$$

Since $\mathrm{level}(F(\tau_1, \tau_2, \tau_3, \tau_4)) \geq \mathrm{order}(F(\tau_1, \tau_2, \tau_3, \tau_4)) > k$, the same argument holds in the setting of bounded order.

Second, a number encoding $\langle n \rangle$ that respects the level (in our case order and arity) bound to address individual cells of the Turing machine tape.

$$\mathcal{N}_0 = \{\bigcap_{i=1}^{n} (b_i)_i \mid b_i \in \{0,1\}\} \text{ where } 0_i, 1_i \in \mathbb{C}_0 \text{ for } i = 1 \ldots n$$
$$\mathcal{N}_{j+1} = \{ \bigcap_{\sigma \in \mathcal{N}_j} (\sigma \to b_\sigma) \mid b_\sigma \in \{0,1\} \text{ for } \sigma \in \mathcal{N}_j\}$$
$$[\![\bigcap_{i=1}^{n} (b_i)_i]\!]_0 = \sum_{i=1}^{n} b_i 2^{i-1} \qquad [\![\bigcap_{\sigma \in \mathcal{N}_j} (\sigma \to b_\sigma)]\!]_{j+1} = \sum_{\sigma \in \mathcal{N}_j} b_\sigma 2^{[\![\sigma]\!]_j}$$

For each $i \in \{0, \ldots, \exp_{k+1}(n) - 1\}$ there is a unique $\sigma \in \mathcal{N}_j$ such that $[\![\sigma]\!]_k = i$. Defining $\langle i \rangle_k \equiv \sigma$ we have $\mathrm{level}(\langle i \rangle_k) = \mathrm{order}(\langle i \rangle_k) = k$ and $\mathrm{arity}(\langle i \rangle_k) \leq 1$. Therefore, under the order of o and arity of 1 restriction, we may instantiate type variables by tape addresses covering $(o+1)$-iterated exponential space.

Third, Turing machine configuration (state p, tape content v, head position h) representation based on an intersection of instances of the type $\mathsf{Cell}(\alpha, \beta, \gamma, \delta)$, where α is substituted by $v_i \in \mathbb{C}_0$, β is substituted by $q \in \mathbb{C}_0$, γ is substituted by $\langle h \rangle_k \in \mathcal{N}_k$, and δ is substituted by $\langle i \rangle_k \in \mathcal{N}_k$. According to the above above number encoding analysis, the more restricted bound of order $o = k$ and arity 1 suffices to represent Turing machine configurations.

Overall, we observe that [30, Section 5] shows $(o+2)$-ExpTime-hardness of inhabitation in $(\vdash_{(o,1)})$, and, a fortiori in $(\vdash_{(o,a)})$ for $a \geq 1$ (Theorem 11).

Theorem 11. *Inhabitation in* $(\vdash_{(o,a)})$ *(Problem 12) is* $(o+2)$-ExpTime-*hard for* $a \geq 1$.

Clearly, for $a = 0$ the above lower bound construction cannot hold because types of arity 0 are necessarily of order 0. By Theorem 10, inhabitation in $(\vdash_{(0,0)})$ is in 2-ExpTime.

3.2.2 Combinatory Logic without Intersection Introduction

In some practical scenarios the intersection introduction rule is not necessary. Therefore, complexity of inhabitation in the fragment of $(\vdash_{(o,a)})$ (Definition 40) without the rule ($\cap$I) is of interest. Upon closer examination, the methodology in [30] can be adapted to show $(o+1)$-ExpTime completeness for this fragment (Conjecture 5).

Conjecture 5. *Inhabitation in* $(\vdash_{(o,a)})$ *for arity* $a \geq 1$ *without the* ($\cap$I) *rule is* $(o+1)$-ExpTime-*complete .*

Our reasoning is as follows. For an upper bound, we may use a similar alternating space procedure with the restriction that only one instance (instead of intersection of arbitrary many) of combinators typings needs to be chosen. This directly reduces the iterated exponential space requirements exactly by one exponential iteration.

For a lower bound, instead of relying on intersection introduction to capture contents of tape cells, we may represent the whole tape content using the iterated exponential number encoding.

Although the corresponding complexity proofs are similar, the question regarding a *direct* translation between $(\vdash_{(o,a)})$ and $(\vdash_{(o+1,a)})$ without the rule ($\cap$I) remains open.

Concluding Remarks

Viewing inhabitant search as execution semantics of a logic programming language, it is essential to know its exact expressiveness in terms of complexity. However, in practice even for functional order of zero inhabitation in bounded combinatory logic is intractable due to its 2-ExpTime-hardness. Therefore, practical implementations (cf. Chapter 4) rely on different and often incomplete (wrt. particular complexity) restrictions.

We consider it worthwhile to explore novel restrictions of combinatory logic with intersection types that treat intersection introduction as a resource similarly to bounded dimensional calculi (Section 2.3). Interestingly, disallowing intersection introduction, which is the most extreme manifestation of this approach, appears to be of well-defined complexity under the order and arity restriction.

3.3 Intersection Type Subtyping

Intersection type subtyping (Definition 35) is an important component in intersection typed λ-calculi [3] as well as intersection typed combinatory logic [27]. Since it is also a key component in bounded combinatory logic (Definition 40, rule ($\leq$)), results on decidability of the subtyping relation (Problem 13) and associated algorithmics are collected in this section.

Problem 13. *(Intersection Type Subtyping, $\sigma \leq \tau$?) Given $\sigma, \tau \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$, does $\sigma \leq \tau$ hold?*

In the Context of Synthesis

Intersection type subtyping can be understood as specification specialization. For example, the component *cl2fh* in [54] that converts real valued temperature from degree Celsius to degree Fahrenheit can be specified by the type $\tau \equiv (\texttt{Real} \to \texttt{Real}) \cap (\texttt{Celsius} \to \texttt{Fahrenheit})$. By intersection type subtyping we have $\tau \leq (\texttt{Real} \cap \texttt{Celsius}) \to (\texttt{Real} \cap \texttt{Fahrenheit})$. Therefore, we may also use the component *cl2fh* at its more special specification.

Intersection type subtyping (Problem 13) is known to be decidable [44] via a normalization argument (that may incur an exponential blow-up). A polynomial time decision procedure based on type normalization (Definition 39) with a quartic upper bound is developed in [55]. Alternatively, a polynomial time decision procedure based on rewriting with a quintic upper bound is given in [63].

Since combinatory logic is per se schematic, we are also interested in matching, satisfiability and unification problems that arise from allowing instantiation of type variables in intersection type subtyping. We hope that a better algorithmic understanding of intersection type subtyping and related problems has a direct impact on type-based synthesis.

Section Outline The remainder of this section inspects complexity of three problems associated with intersection type subtyping.

First, in Section 3.3.1 an algorithm to decide intersection type subtyping in quadratic time (Theorem 12) is given. Second, in Section 3.3.2 we outline fixed-parameter intractability of intersection type matching wrt. the number of type variables (Theorem 13). Third, in Section 3.3.3 we show an ExpTime lower bound for intersection type unification (Theorem 14) by reduction from two player tiling games.

Authorship Statement Since contributions presented in this section are part of joint work [33], this mandatory paragraph lists the following contributions attributed to the author.

- quadratic time upper bound to decide intersection type subtyping
- fixed-parameter intractability of intersection type matching
- ExpTime lower bound for intersection type unification

3.3.1 Deciding Intersection Type Subtyping in Quadratic Time

Common to previous approaches [44, 55, 63] to decide intersection type subtyping (Problem 13) is that they rely on (partial) normalization, and therefore have to address a potentially exponential number of sub-instances either by memoization [55] or dynamic programming [63].

In this paragraph we describe a different algorithmic approach to decide subtyping with a quadratic upper bound. We show that direct implementation of the so-called *beta-soundness* property (Lemma 43, inspected in detail in [3, Lemma 2.4.2]) does not incur an exponential number of recursive calls.

Lemma 43 (Beta-Soundness [3, Lemma 2.4.2])**.** *Given* $\sigma = \bigcap_{i\in I}(\sigma_i \to \tau_i) \cap \bigcap_{j\in J} a_j \cap \bigcap_{k\in K} \alpha_k$*, we have*

(*i*) *If* $\sigma \leq a$ *for some* $a \in \mathbb{C}_0$*, then* $a \equiv a_j$ *for some* $j \in J$*.*

(*ii*) *If* $\sigma \leq \alpha$ *for some* $\alpha \in \mathbb{V}$*, then* $\alpha \equiv \alpha_k$ *for some* $k \in K$*.*

(*iii*) *If* $\sigma \leq \sigma' \to \tau' \neq \omega$ *for some* $\sigma', \tau' \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$*, then* $I' = \{i \in I \mid \sigma' \leq \sigma_i\} \neq \emptyset$ *and* $\bigcap_{i\in I'} \tau_i \leq \tau'$*.*

Corollary 8. *Given a path* $\pi \in \mathbb{P}$ *and types* σ, τ*, we have* $\sigma \cap \tau \leq \pi$ *iff* $\sigma \leq \pi$ *or* $\tau \leq \pi$*.*

We give the following Algorithm SUB to decide intersection type subtyping.

Algorithm 3 Recursive Algorithm SUB deciding $\sigma \leq \tau$

Input: intersection types $\sigma, \tau \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$
Output: **true** iff $\sigma \leq \tau$
if $\tau \equiv \omega$ **or** $\sigma \equiv \alpha \equiv \tau$ **or** $\sigma \equiv a \equiv \tau$ **then**
  return true
else if $\tau \equiv \tau_1 \cap \tau_2$ **then**
  if $\mathsf{SUB}(\sigma, \tau_1)$ **and** $\mathsf{SUB}(\sigma, \tau_2)$ **then**
    return true
  end if
else if $\sigma \equiv \sigma_1 \cap \sigma_2$ **and** $(\tau \equiv \alpha$ **or** $\tau \equiv a)$ **then**
  if $\mathsf{SUB}(\sigma_1, \tau)$ **or** $\mathsf{SUB}(\sigma_2, \tau)$ **then**
    return true
  end if
else if $\tau \equiv \tau_1 \to \tau_2$ **then**
  if $\mathsf{SUB}(\mathsf{AUX}(\sigma, \tau_1), \tau_2)$ **then**
    return true
  end if
end if
return false

Algorithm 4 Recursive Algorithm AUX auxiliary to SUB (Algorithm 3)

1: *Input:* intersection types $\sigma, \tau \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$
2: *Output:* intersection of targets of arrows $\sigma_1 \to \sigma_2$ in σ such that $\tau \leq \sigma_1$
3: **if** $\sigma \equiv \sigma_1 \to \sigma_2$ **then**
4: **if** $\mathsf{SUB}(\tau, \sigma_1)$ **then**
5: **return** σ_2
6: **end if**
7: **else if** $\sigma \equiv \sigma_1 \cap \sigma_2$ **then**
8: **return** $\mathsf{AUX}(\sigma_1, \tau) \cap \mathsf{AUX}(\sigma_2, \tau)$
9: **end if**
10: **return** ω

Algorithm SUB is correct (Lemma 44) and terminates in quadratic time (Lemma 45).

Lemma 44. *Algorithm $\mathsf{SUB}(\sigma, \tau)$ (Algorithm 3) returns* **true** iff $\sigma \leq \tau$ holds.

Proof. Algorithm $\mathsf{SUB}(\sigma, \tau)$ directly implements beta-soundness (Lemma 43) where the auxiliary Algorithm AUX collects arrow targets indexed in the set I' (Lemma 43 (*iii*)). Additionally the principle $\sigma \leq \tau_1 \cap \tau_2$ iff $\sigma \leq \tau_1$ and $\sigma \leq \tau_2$ is used in lines 5–8. The only case which is not covered by Lemma 43 (*iii*) is $\tau \equiv \tau_1 \to \tau_2 = \omega$. In this case we have $\tau_2 = \omega$, therefore $\mathsf{SUB}(\mathsf{AUX}(\sigma, \tau_1), \tau_2)$ also correctly succeeds regardless of the type $\mathsf{AUX}(\sigma, \tau_1)$. □

Lemma 45. *Algorithm $\mathsf{SUB}(\sigma, \tau)$ (Algorithm 3) terminates in time $\mathcal{O}(|\sigma||\tau|)$.*

Proof. Let $T(\sigma, \tau)$ be the running time of $\mathsf{SUB}(\sigma, \tau)$ and $T'(\sigma, \tau)$ the running time of $\mathsf{AUX}(\sigma, \tau)$. Let $a \in \mathbb{N}$ be total amount of constant running time of computation anywhere in SUB and AUX. We show $T(\sigma, \tau) \leq a|\sigma||\tau| = \mathcal{O}(|\sigma||\tau|)$ and $T'(\sigma, \tau) \leq a|\sigma||\tau| = \mathcal{O}(|\sigma||\tau|)$ by induction on $|\sigma| + |\tau|$.

Basis Step: For $\sigma, \tau \in \{\omega\} \cup \mathbb{C}_0 \cup \mathbb{V}$ we have
$T(\sigma, \tau) = \mathcal{O}(1)$ and $T'(\sigma, \tau) = \mathcal{O}(1)$. Therefore, $T(\sigma, \tau) \leq a \leq a|\sigma||\tau|$ and $T'(\sigma, \tau) \leq a \leq a|\sigma||\tau|$.

Inductive Step $T'(\sigma, \tau)$:

Case $\sigma \equiv \sigma_1 \to \sigma_2$: We have $T'(\sigma, \tau) = \mathcal{O}(1) + T(\tau, \sigma_1)$, therefore

$$T'(\sigma, \tau) \overset{(\mathrm{IH})}{\leq} a + a|\tau||\sigma_1| \leq a + a(|\sigma| - 1)|\tau| \leq a|\sigma||\tau|$$

Case $\sigma \equiv \sigma_1 \cap \sigma_2$: We have $T'(\sigma, \tau) = \mathcal{O}(1) + T'(\sigma_1, \tau) + T'(\sigma_2, \tau)$, therefore

$$T'(\sigma, \tau) \overset{(\mathrm{IH})}{\leq} a + a|\sigma_1||\tau| + a|\sigma_2||\tau| = a + a(|\sigma| - 1)|\tau| \leq a|\sigma||\tau|$$

Otherwise: We have $T'(\sigma, \tau) = \mathcal{O}(1)$, therefore $T'(\sigma, \tau) \leq a \leq a|\sigma||\tau|$.

Inductive Step $T(\sigma, \tau)$:

Case $\tau \equiv \tau_1 \cap \tau_2$: We have $T(\sigma, \tau) = \mathcal{O}(1) + T(\sigma, \tau_1) + T(\sigma, \tau_2)$, therefore

$$T(\sigma, \tau) \overset{(\mathrm{IH})}{\leq} a + a|\sigma||\tau_1| + a|\sigma||\tau_2| = a + a|\sigma|(|\tau| - 1) \leq a|\sigma||\tau|$$

Case $\sigma \equiv \sigma_1 \cap \sigma_2$, $\tau \in \mathbb{C}_0 \cup \mathbb{V}$**:** We have $T(\sigma,\tau) = \mathcal{O}(1) + T(\sigma_1,\tau) + T(\sigma_2,\tau)$, therefore

$$T(\sigma,\tau) \overset{\text{(IH)}}{\leq} a + a|\sigma_1||\tau| + a|\sigma_2||\tau| = a + a(|\sigma| - 1)|\tau| \leq a|\sigma||\tau|$$

Case $\tau \equiv \tau_1 \to \tau_2$**:** By routine induction we have $|\mathsf{AUX}(\sigma,\tau_1)| \leq |\sigma|$. Additionally, $T(\sigma,\tau) = \mathcal{O}(1) + T'(\sigma,\tau_1) + T(\mathsf{AUX}(\sigma,\tau_1),\tau_2)$, therefore

$$\begin{aligned} T(\sigma,\tau) &\overset{\text{(IH)}}{\leq} a + a|\sigma||\tau_1| + a|\mathsf{AUX}(\sigma,\tau_1)||\tau_2| \\ &\leq a + a|\sigma||\tau_1| + a|\sigma||\tau_2| = a + a|\sigma|(|\tau| - 1) \leq a|\sigma||\tau| \end{aligned}$$

Otherwise: We have $T(\sigma,\tau) = \mathcal{O}(1)$, therefore $T(\sigma,\tau) \leq a \leq a|\sigma||\tau|$. □

Overall, Algorithm SUB can be used to decide intersection type subtyping in quadratic time (Theorem 12).

Theorem 12. *Intersection type subtyping $\sigma \leq \tau$? (Problem 13) is decidable in time $\mathcal{O}(|\sigma||\tau|)$.*

Proof. By Lemma 44 and Lemma 45. □

A similar argument to the above that formally proves (in the Coq proof assistant) a quadratic upper bound on the number of recursive calls to decide intersection type subtyping is given in [11].

It remains an open problem whether intersection type subtyping is PTIME-complete (Conjecture 6). Intuitively, beta-soundness (Lemma 43 (*iii*)) could enable sharing in the sense of Boolean circuits for a lower bound. This intuition is strengthened by the fact that type normalization, i.e. recursive organization, eliminates sharing incurring an exponential blow-up.

Conjecture 6. *Intersection type subtyping $\sigma \leq \tau$? (Problem 13) is* PTIME*-complete.*

Since intersection type subtyping is related to intuitionistic linear logic ($\multimap$ corresponds to $\to$ and & corresponds to $\cap$)[3] as well as minimal relevant logic [69, Section 3], further study could provide insights into complexity of provability in these logic fragments.

[3]Communicated by Olivier Laurent to the author in 2015.

3.3.2 Intractability of Intersection Type Matching

Intersection type matching occurs naturally during inhabitant search in intersection type systems and is known to be NP-complete [28]. We strengthen this result by showing that the problem remains NP-hard even when restricted to the fixed-parameter case where only a single type variable and only a single constant is used in the input.

For $\tau \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$ let $\mathrm{var}(\tau) \subseteq \mathbb{V}$ denote the set of type variables occurring in τ.

Problem 14 (Intersection Type Matching)**.** *Given a set of constraints* $\{\sigma_1 \stackrel{\cdot}{\leq} \tau_1, \ldots, \sigma_n \stackrel{\cdot}{\leq} \tau_n\}$*, where for each* $i \in \{1, \ldots, n\}$ *we have* $\mathrm{var}(\sigma_i) = \emptyset$ *or* $\mathrm{var}(\tau_i) = \emptyset$*, is there a substitution* $S \colon \mathbb{V} \to \mathbb{T}^{\cap}_{\mathbb{C}_0}$ *such that* $S(\sigma_i) \leq S(\tau_i)$ *for* $i = 1 \ldots n$*?*

We say that a substitution S *satisfies* $\{\sigma_1 \stackrel{\cdot}{\leq} \tau_1, \ldots, \sigma_n \stackrel{\cdot}{\leq} \tau_n\}$ if $S(\sigma_i) \leq S(\tau_i)$ for $i = 1 \ldots n$.

In the Context of Synthesis

Commonly, library components are combinators with schematic type specification, e.g. $b : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$ in Example 41, whereas user input is specific, e.g. $0 \to 0$.
Matching (parts of) specification against the user input corresponds to solving an intersection type matching problem instance, e.g. $\alpha \to \gamma \stackrel{\cdot}{\leq} 0 \to 0$.

Any matching constraint set $\{\sigma_1 \stackrel{\cdot}{\leq} \tau_1, \ldots, \sigma_n \stackrel{\cdot}{\leq} \tau_n\}$ can be reduced to a single matching constraint $\sigma \stackrel{\cdot}{\leq} \tau$ with $\mathrm{var}(\sigma) = \emptyset$ by fixing a type constant $\bullet \in \mathbb{C}_0$, defining for $i = 1 \ldots n$

$$(\sigma'_i, \tau'_i) = \begin{cases} (\sigma_i \to \bullet, \tau_i \to \bullet) & \text{if } \mathrm{var}(\sigma_i) = \emptyset \\ (\tau_i, \sigma_i) & \text{if } \mathrm{var}(\tau_i) = \emptyset \end{cases}$$

and taking $\sigma \equiv \sigma'_1 \to \ldots \to \sigma'_n \to \bullet$, $\tau \equiv \tau'_1 \to \ldots \to \tau'_n \to \bullet$. We have that $\mathrm{var}(\sigma) = \emptyset$. By Lemma 43, for any substitution S we have $S(\sigma) \leq S(\tau)$ iff $S(\sigma_i) \leq S(\tau_i)$ for $i = 1 \ldots n$. Therefore, matching is NP-complete even when restricted to single constraints.

In [28] the lower bound for intersection type matching is shown by reduction from 3SAT and requires two type variables $\alpha_x, \alpha_{\neg x}$ for each propositional variable x. Since 3SAT, parameterized by the number of propositional variables, is fixed-parameter tractable, it is natural to ask whether the same holds for matching parameterized by the number of type variables. Unfortunately, this is not the case (Theorem 13).

Theorem 13 ([33, Lemma 5])**.** *Intersection type matching (Problem 14) is* NP*-hard even if only a single type variable and a single type constant is used.*

Proof. (Sketch) Fix a 3SAT instance F containing clauses $(L_1 \vee L_2 \vee L_3) \in F$ over propositional variables V where L_i is either x or $\neg x$ for some $x \in V$. We reduce satisfiability of F to matching with one type variable α. First, we fix a set of type constants $B = V \cup \{\neg x \mid x \in V\}$ and the type constant $\bullet$. Let $\sigma_x \equiv \bigcap(B \setminus \{\neg x\})$ and $\sigma_{\neg x} \equiv \bigcap(B \setminus \{x\})$ for $x \in V$. Let the set $\mathscr{C}_F$ contain the following constraints

for $x \in V$ (consistency) :

$$((\sigma_{\neg x} \to \bullet) \to (\neg x \to \bullet)) \cap ((\sigma_x \to \bullet) \to (x \to \bullet)) \dot{\leq} (\alpha \to \bullet) \to (\alpha \to \bullet)$$

for $(L_1 \vee L_2 \vee L_3) \in F$ (validity) :

$$(L_1 \to \bullet) \cap (L_2 \to \bullet) \cap (L_3 \to \bullet) \dot{\leq} \alpha \to \bullet$$

If F is satisfied by a valuation v, then the substitution $\alpha \mapsto \bigcap\limits_{v(x)=1} x \cap \bigcap\limits_{v(x)=0} \neg x$ satisfies $\mathscr{C}_F$.
If $\mathscr{C}_F$ is satisfied by a substitution S, by the consistency constraints we have either $\sigma_{\neg x} \leq S(\alpha) \leq \neg x$ or $\sigma_x \leq S(\alpha) \leq x$ for $x \in V$. A valuation v constructed according to these cases satisfies each clause in F by the validity constraints.

Instead of using constants $\{a_1, \ldots, a_k, \bullet\}$, encode $[a_i] = \underbrace{\bullet \to \ldots \to \bullet \to}_{i \text{ times}} \bullet$ for $i = 1 \ldots k$. Using this technique, a single type constant $\bullet$ is sufficient. □

Combining the above Theorem 13 with results in [28] we conclude that neither restricting substitution domain to $S : \{\alpha\} \to \mathbb{T}^{\cap}_{\mathbb{C}_0}$ nor restricting substitution codomain to $S : \mathbb{V} \to \mathbb{C}_0$ reduces the complexity of intersection type matching.

3.3.3 Intersection Type Unification ExpTime-hardness

Compared to intersection type matching (Problem 14), *intersection type satisfiability* (Problem 15), short for intersection type subtyping constraint satisfiability, does not restrict occurrences of type variables in constraints. Similarly, *intersection type unification* (Problem 16) is formulated wrt. type equality.

Problem 15 (Intersection Type Satisfiability)**.** *Given a set of constraints* $\{\sigma_1 \dot{\leq} \tau_1, \ldots, \sigma_n \dot{\leq} \tau_n\}$*, is there a substitution* $S\colon \mathbb{V} \to \mathbb{T}^{\cap}_{\mathbb{C}_0}$ *such that* $S(\sigma_i) \leq S(\tau_i)$ *for* $i = 1 \ldots n$*?*

Problem 16 (Intersection Type Unification)**.** *Given a set of constraints* $\{\sigma_1 \doteq \tau_1, \ldots, \sigma_n \doteq \tau_n\}$*, is there a substitution* $S\colon \mathbb{V} \to \mathbb{T}^{\cap}_{\mathbb{C}_0}$ *such that* $S(\sigma_i) = S(\tau_i)$ *for* $i = 1 \ldots n$*?*

In the Context of Synthesis

Intersection type satisfiability arises naturally whenever two schematic component specifications are examined wrt. composability. For example, some component $x : \alpha \to \alpha$ serving as an argument of some other component $y : (\beta \to b \to a) \to a$ may lead to the question of satisfiability of $\alpha \to \alpha \dot{\leq} \beta \to b \to a$. A negative answer would imply that the component y cannot be applied to x regardless of user queries. However, in this case a positive answer $\alpha, \beta \mapsto b \to a$ implies that

$$\{x : \alpha \to \alpha, y : (\beta \to b \to a) \to a\} \vdash_{\mathcal{C}(\cap)} y\, x : a$$

For any $\sigma, \tau \in \mathbb{T}^{\cap}_{\mathbb{C}_0}$ and any substitution S we have

$$S(\sigma) \leq S(\tau) \iff S(\sigma) \cap S(\tau) = S(\sigma)$$

Therefore, satisfiability and unification are equivalent. Similarly to matching, restricting satisfiability (resp. unification) to single constraints does not change its complexity.

Intersection type unification is non-structural in the sense that types with highly different syntax trees may be related (Example 43). Additionally, neither excluding ω from solutions nor restricting substitution codomain to paths overcomes this obstacle (Example 43). This impedes usual approaches to unification via an occurs-check.

Example 43. *The satisfiability constraint* $\alpha \dot{\leq} \alpha \to a$ *(resp. unification constraint* $\alpha \doteq \alpha \cap (\alpha \to a)$*) is solved by any of the following substitutions*

- $S_1(\alpha) = \omega \to a$
  showing that non-structural constraints may be solved
- $S_2(\alpha) = a \cap (a \to a)$
  showing that non-structural constraints may be solved without ω
- $S_3(\alpha) = ((a \cap (a \to a)) \to a) \to a$
  showing that non-structural constraints may be solved using a single path

Interestingly, in the above Example 43 any substitution S solving the constraint $\alpha \dot{\leq} \alpha \to a$ induces a typing of the λ-term $\lambda x^{S(\alpha)}.x^{S(\alpha \to a)}\, x^{S(\alpha)}$ by using intersection type subtyping. Again, this underlines the non-structural nature of the subtyping relation.

Similarly to matching, satisfiability (resp. unification) problems arise naturally in combinatory logic derivations (Example 44).

Example 44. *Let* $\Gamma = \{x : (\sigma \to \tau) \to a, y : \alpha \to \alpha\}$ *such that* $\alpha \notin \mathrm{var}(\sigma \to \tau)$. *In this scenario, type-checking* $\Gamma \vdash_{\mathcal{C}(\cap)} x\, y : a$ *is equivalent to solving the satisfiability problem instance* $\alpha \to \alpha \dot{\leq} \sigma \to \tau$, *or equivalently, the unification problem instance* $\sigma \cap \tau \doteq \sigma$, *because we need to find substitutions* $S, S_1, \ldots, S_n$ *for some* $n \in \mathbb{N}$ *such that*

$$\begin{aligned}
&\bigcap_{i=1}^{n} S_i(\alpha \to \alpha) \leq S(\sigma \to \tau) \\
\overset{\text{Lem. } 43}{\Longleftrightarrow}\ & S(\sigma) \leq \bigcap_{i \in I} S_i(\alpha) \text{ and } \bigcap_{i \in I} S_i(\alpha) \leq S(\tau) \text{ for some } I \subseteq \{1, \ldots, n\} \\
\Longleftrightarrow\ & S(\alpha \to \alpha) \leq S(\sigma \to \tau) \text{ setting } S(\alpha) = \bigcap_{i \in I} S_i(\alpha)
\end{aligned}$$

In the Context of Synthesis

Unification problem instances in inhabitant search arise in the presence of *cut types*. In this case schematic subformulae are not matched against specific user input but need to be unified with schematic specifications of other components.

In addition to the previously illustrated non-structurality of unification, the following Example 45 shows that unification is not finitary, i.e. unification problem instances may have infinitely many most general unifiers.

Example 45. *Consider the unification constraint* $a \to a \to (\beta \cap b) \doteq \beta \cap \alpha$ *and some solution* S. *We have* $S(\beta) = a \to a \to \bigcap_{i \in I} \pi_i$ *such that for all* $i \in I$ *either* $\pi_i = b$ *or* $\pi_i = a \to a \to \pi_j$ *for some* $j \in I$. *Therefore,* $S(\beta)$ *(and consequently* $S(\alpha)$*) contains paths that may be arbitrary long, end in the constant* b, *and have an even number of "a"s as arguments. As a result, solutions to the above constraint cannot be built by specialization from a finite set of unifiers.*

The only hitherto known non-trivial lower bound for intersection type unification is ExpTime [33, Theorem 1] by reduction from existence of winning strategies in two-player tiling games, which we will outline in more detail in the remainder of this section.

Tiling games

We briefly describe a special kind of domino tiling game [21], referred to as *two-player spiral tiling* game, for which the problem of existence of winning strategies is ExpTime-complete. This problem will be used to prove our ExpTime-lower bound for intersection type unification.

Definition 43 (Tiling System)**.** *A* tiling system *is a tuple* $(D, H, V, \bar{b}, \bar{t}, n)$*, where*

- D *is a finite set of tiles (also called dominoes)*
- $H, V \subseteq D \times D$ *are horizontal and vertical constraints*
- $\bar{b}, \bar{t} \in D^n$ *are* n*-tuples of tiles*
- n *is a unary encoded natural number*

Definition 44 (Spiral Tiling)**.** *Given a tiling system* $(D, H, V, \bar{b}, \bar{t}, n)$*, a* spiral tiling *is a sequence* $d_1 \dots d_m \in D^m$ *for some* $m \in \mathbb{N}$ *such that*

- $\bar{b} = d_1 \dots d_n$ *(correct prefix)*
- $\bar{t} = d_{m-n+1} \dots d_m$ *(correct suffix)*
- $(d_i, d_{i+1}) \in H$ *for* $1 \leq i \leq m-1$ *(horizontal constraints)*
- $(d_i, d_{i+n}) \in V$ *for* $1 \leq i \leq m-n$ *(vertical constraints)*

Given a tiling system $(D, H, V, \bar{b}, \bar{t}, n)$, a *two-player spiral tiling* game is played by *Constructor* and *Spoiler*. The game starts with the sequence $\bar{b}$. Each player adds a copy of a tile to the end of the current sequence taking turns starting with Constructor. While Constructor tries to construct a spiral tiling, Spoiler tries to prevent it. Constructor *wins* if Spoiler makes an illegal move (with respect to H or V), or when a correct spiral tiling is completed. Constructor has *winning strategy*, if he can win regardless of what Spoiler does. To decide whether a winning strategy exists is ExpTime-complete (Lemma 46).

Lemma 46 ([33, Lemma 8])**.** *The decision problem whether Constructor has a winning strategy in a given two-player spiral tiling game is* ExpTime*-complete.*

In [21] instead of spiral tilings so-called *corridor tilings* are considered. The difference between a corridor tiling and a spiral tiling is the lack of individual rows. While a tile at the beginning of a row of a corridor is not constrained by the previous tile, in a spiral tiling each new tile is constrained by the previous one. Additionally, any corridor tiling contains $l \cdot n$ tiles for some l; a spiral tiling does not obey this restriction. To clarify aspects of winning strategies in spiral tiling games, consider the following Example 46 and Example 47.

Example 46. *Consider the tiling system* $D = \{a, b\}$*,* $H = \{(a, b), (b, a), (b, b)\}$*,* $V = D^2$*,* $\bar{b} = aaa$*,* $\bar{t} = bbb$ *and* $n = 3$*. Constructor does not have a winning strategy in the corresponding two-player spiral tiling game. During the game on Spoiler's turn there are two possibilities. In case the current sequence ends in* a*, Spoiler is forced/allowed to append* b*, which does not result in the suffix* bbb*. Regardless of Constructor's next tile the suffix is not* bbb*. In case the current sequence ends in* b*, Spoiler is allowed to append* a *and, similarly, Constructor is not able to produce a spiral tiling.*

Example 47. *Consider the tiling system* $D = \{a, b\}$, $H = D^2$, $V = D^2 \setminus \{(b, a)\}$, $\bar{b} = aaaaa$, $\bar{t} = bbbbb$ *and* $n = 5$. *Constructor has a winning strategy in the corresponding two-player spiral tiling game by always appending* b. *Due to* V, *if the current sequence has the tile* b *at position* i, *then it will have the tile* b *at any later position* $i + 5j$ *for* $j \in \mathbb{N}$. *Therefore, after the first nine turns of the game all positions* $1 + 5j$, $3 + 5j$, $5 + 5j$, $7 + 5j$ *and* $9 + 5j$ *for* $j \in \mathbb{N}$ *will have the tile* b *regardless of Spoiler's moves. Since those positions will form a suffix bbbbb after* 9 *turns, Constructor is able to produce a spiral tiling.*

Unification ExpTime Lower Bound

Let us outline the reduction from spiral tiling games to the intersection type satisfiability problem.

Let $T = (D, H, V, \bar{b} = b_1 \dots b_n, \bar{t} = t_1 \dots t_n, n)$ be a tiling system. Wlog. $(b_i, b_{i+1}) \in H$ and $(t_i, t_{i+1}) \in H$ for $1 \leq 1 < n$, otherwise the given prefix $\bar{b}$ or the given suffix $\bar{t}$ would already violate constraints on consecutive tiles. We fix the set of type constants $D \dot\cup \{\bullet\} \subseteq \mathbb{C}_0$ and variables $\{\alpha\} \cup \{\beta_d \mid d \in D\} \subseteq \mathbb{V}$ and construct the following set of constraints $\mathscr{C}_T$:

$$(i)\ \ \sigma_\bot^H \cap \sigma_\bot^V \cap \sigma_t \cap \bigcap_{d \in D} \beta_d \dot\leq \sigma_b \cap \bigcap_{d' \in D} \bigcap_{d \in D} (d' \to d \to \beta_d) \qquad \text{(Game moves)}$$

$$(ii)\ \ \bigcap_{(d', d) \in H} (d \to d' \to \alpha) \dot\leq \bigcap_{d \in D} (d \to \beta_d) \qquad (d \text{ respects } H)$$

$$(iii)\ \ \bigcap_{(d', d) \in V} (d \to \underbrace{\omega \to \ldots \to \omega \to}_{n-1 \text{ times}} d' \to \alpha) \dot\leq \bigcap_{d \in D} (d \to \beta_d) \qquad (d \text{ respects } V)$$

where

$$\sigma_b \equiv b_n \to \ldots \to b_1 \to \bullet \qquad \text{(Initial state)}$$

$$\sigma_t \equiv (t_n \to \ldots \to t_1 \to \alpha) \cap (\omega \to t_n \to \ldots \to t_1 \to \alpha) \qquad \text{(Final states)}$$

$$\sigma_\bot^H \equiv \bigcap_{(d, d') \in D \times D \setminus H} (d' \to d \to \alpha) \qquad (d' \text{ violates } H)$$

$$\sigma_\bot^V \equiv \bigcap_{(d, d') \in D \times D \setminus V} (d' \to \underbrace{\omega \to \ldots \to \omega \to}_{n-1 \text{ times}} d \to \alpha) \qquad (d' \text{ violates } V)$$

Let us provide some intuition for the above construction. A sequence $d_1 \dots d_l$ of tiles is represented by the type $d_l \to \ldots \to d_1 \to \bullet$. By Lemma 43 we have that $\bigcap_{i \in I} (d^i_{l_i} \to \ldots \to d^i_1 \to \bullet) \leq d_l \to \ldots \to d_1 \to \bullet$ implies $d^i_1 \dots d^i_{l_i} = d_1 \dots d_l$ for some $i \in I$. Therefore, the above constraints for certain solutions correspond to set inclusion constraints on tile sequences.

The rhs of (i) is an intersection of representations of sequences which Constructor may face. For all such sequences he needs to find a suitable move by choosing a path on the lhs of (i). He can either state that the Spoiler's last move violates H (resp. V) choosing $\sigma_\bot^H$ (resp. $\sigma_\bot^V$), or that the game is finished choosing σ_t, or he can pick his next move $d \in D$ choosing β_d. Intuitively, β_d captures all sequences in which Constructor decides to place d next. Accordingly, on the rhs of (i) in the type $d' \to d \to \beta_d$ the tile d' is not constrained (Spoiler may add any tile d') while the tile d is constrained to the index of β_d,

i.e. Constructor's previous choice. Therefore, by picking a move d Constructor faces sequences that arise from the previous sequence extended by d and each possible d'. Constraints (ii) and (iii) ensure that whenever Constructor chooses to add $d \in D$ choosing β_d he has to respect H and V.

As a result, if Constructor has a winning strategy for the two-player spiral tiling game in T, then we can construct a solution for $\mathscr{C}_T$ substituting β_d by an intersection of representations of sequences in which Constructor decides to place d [33, Lemma 9]. Conversely, if the constraint system $\mathscr{C}_T$ is satisfiable, then we can use Corollary 8 to guide Constructor's choices resulting in a winning strategy [33, Lemma 10]. In sum, we obtain an ExpTime lower bound for satisfiability and unification (Theorem 14, Corollary 9).

Theorem 14 ([33, Theorem 1])**.** *The intersection type satisfiability problem (Problem 15) is* ExpTime*-hard.*

Corollary 9. *The intersection type unification problem (Problem 16) is* ExpTime*-hard.*

Let us conclude this section with observations regarding restrictions of intersection type unification. First, unification remains ExpTime-hard even in presence of only one type constant [33, Corollary 2]. Second, removing ω from the type language has no impact on the presented ExpTime lower bound [33, Theorem 2]. Third, restricting the codomain of substitutions to rank 1 types (intersections of simple types) does not change the ExpTime lower bound construction [33, Corollary 3]. In fact, the main proof in [33, Theorem 1] uses rank 1 types. Since rank 1 subtyping behaves similarly to set inclusion, we conjecture that the rank 1 intersection type unification problem is decidable (Conjecture 7).

Problem 17 (Rank 1 Intersection Type Unification)**.** *Given a set of constraints* $\{\sigma_1 \doteq \tau_1, \ldots, \sigma_n \doteq \tau_n\}$*, is there a substitution* $S\colon \mathbb{V} \to \mathbb{T}^{\cap}_{\mathbb{C}_0}$ *such that* $\mathrm{rank}(S(\alpha)) \leq 1$ *for each* $\alpha \in \mathbb{V}$ *and* $S(\sigma_i) = S(\tau_i)$ *for* $i = 1 \ldots n$*?*

Conjecture 7. *The rank* 1 *intersection type unification problem (Problem 17) is decidable.*

Our reasoning is that rank 1 unification can be reduced to set constraints with projections and cardinalities in finite sets [33, Section 6]. The closest known result is that satisfiability of set constraints with projections in infinite sets is in NExpTime [20]. Reasonably, techniques from set constraint solving may be applicable to rank 1 unification.

Concluding Remarks

Of the three considered decision problems (subtyping, matching, and unification) in this section, only for intersection type matching we have tight upper and lower bounds on complexity. Although it is reasonable to think that intersection type subtyping can encode sharing (in terms of Boolean circuits) and may therefore be PTime-complete any efforts done by the author to show a tight lower bound were unsuccessful. For intersection type unification, both a tree automaton approach to attempt a proof of decidability (similarly to set constraints [20]), or an approach that uses variance wrt. the arrow type constructor to represent two counters of a Minsky machine to attempt a proof of undecidability (similarly to second-order subtyping [64]) seem reasonable. It remains an open problem whether intersection type unification is decidable.

Chapter 4

(CL)S-F#

Chapter 2 outlines the methodology of synthesis from scratch driven by inhabitant search in typed λ-calculi. In particular, types correspond to desired functional specification and inhabitants correspond to synthesized programs. Complementarily, Chapter 3 outlines the methodology of synthesis from a given collection of existing components driven by relativized inhabitant search in typed combinatory logic. In this scenario, component behavior is specified by corresponding typed combinators and synthesized programs are restricted to applicative compositions of given components. We believe that the latter approach is applicable in realistic scenarios because it is aligned towards embedding domain-specific knowledge in a modular fashion.

The seminal paper [54] by Rehof conveys a vision that relativized inhabitation in combinatory logic with intersection types may be a practical approach for component-based synthesis. Specifically, intersection types are used to alleviate "specification complexity" providing a theoretical foundation to extend native type specification with domain-specific information (so-called *semantic types*). The idea of composition synthesis based on combinatory logic in [54] has been prototypically implemented under the name (CL)S (Combinatory Logic Synthesizer [9]) in the C# programming language.

Subsequently, two lines of work followed the prototypic C# implementation. First, the (CL)S-Scala framework [6] aims to provide a mature user interface including initial component specification by Scala[1] metaprograms and a web-based synthesis algorithm interface. The theoretical foundation behind (CL)S-Scala is a restricted form of bounded combinatory logic [30] where substitution codomain has to be listed explicitly by the user and intersection introduction may be performed only at the top level. The most prominent use of (CL)S-Scala is a broad evaluation of synthesis approaches to develop a product line of Solitaire games [43].

The second line of work inspired by [9] is the (CL)S-F# inhabitant search algorithm [32] (written in the F# programming language[2]) that is presented in this chapter in more detail.

The motivating idea behind (CL)S-F# is that inhabitant search in typed combinatory logic can be seen as execution of a logic program given by type

[1] `https://scala-lang.org/`
[2] `https://fsharp.org/`

assumptions. Therefore, instead of implementing the naive algorithm, (CL)S-F# applies methods known from logic program evaluation such as partial evaluation [46] and sideways information passing [5]. We argue that any scalable approach needs to rely on such techniques because, even at level 0, the search space in bounded combinatory logic [30] grows doubly-exponentially. Overall, (CL)S-F# is developed as a testing ground for inhabitant search techniques rather than a complete synthesis framework.

Chapter Outline In this chapter we present and evaluate the (CL)S-F# inhabitant search algorithm [32].

In Section 4.1 we outline combinatory logic with intersection types with constructors, which is the theoretical foundation of (CL)S-F#.

Section 4.2 provides an overview over the inhabitant search interface (Section 4.2.5) and implementation (Section 4.2.6) of (CL)S-F#. Additionally, we describe the realization of key techniques such as partial evaluation (Section 4.2.3) and sideways information passing (Section 4.2.4), known in the area of logic program evaluation, in (CL)S-F#.

In Section 4.3 we evaluate (CL)S-F# in context of functional program synthesis (Section 4.3.1), object-oriented program synthesis (Section 4.3.2), and process synthesis (Section 4.3.3). Additionally, we inspect scalability of (CL)S-F# in deterministic (Section 4.3.4) as well as non-deterministic (Section 4.3.5) scenarios.

4.1 (CL)S-F# Theoretical Foundation

This section provides an overview over the theoretical foundation of the (CL)S-F# implementation [32].

Combinatory terms (Definition 29) serve as the term language of (CL)S-F#. The type language of (CL)S-F# is *intersection types with constructors* $\mathbb{T}^{\cap}_{\mathbb{C}}$ (Definition 45) which extends intersection types with constants $\mathbb{T}^{\cap}_{\mathbb{C}_0}$ (Definition 34) by covariant, distributing constructors of arbitrary arity.

Definition 45 (Intersection Types with Constructors, $\mathbb{T}^{\cap}_{\mathbb{C}}$)**.**

$$\mathbb{T}^{\cap}_{\mathbb{C}} \ni \sigma, \tau ::= \alpha \mid \omega \mid \sigma \to \tau \mid \sigma \cap \tau \mid c(\tau_1, \ldots, \tau_{\text{arity}(c)})$$
$$\textit{where } \alpha \textit{ ranges over type variables } \mathbb{V} \textit{ and}$$
$$c \textit{ ranges over type constructors } \mathbb{C}$$
$$\textit{each associated with a particular } \text{arity}(c)$$

We write $\mathbb{C}_n$ for the set of type constructors of arity n. In case of nullary type constructors we omit the parentheses, i.e. we write c for $c()$, essentially treating nullary type constructors as type constants. Covariance and distribution properties of type constructors are captured by extended subtyping rules (Definition 46).

Definition 46 (Intersection Type with Constructors Subtyping, $\leq$)**.** *Given a partial order* $\leq_{\mathbb{C}} \subseteq \bigcup_{n=0}^{\infty}(\mathbb{C}_n \times \mathbb{C}_n)$ *that respects constructor arity, the relation* $\leq$ *is the least preorder over* $\mathbb{T}^{\cap}_{\mathbb{C}}$ *such that*

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau,$$
$$(\sigma \to \tau_1) \cap (\sigma \to \tau_2) \leq \sigma \to \tau_1 \cap \tau_2,$$
$$\textit{if } \sigma \leq \tau_1 \textit{ and } \sigma \leq \tau_2 \textit{ then } \sigma \leq \tau_1 \cap \tau_2,$$
$$\textit{if } \sigma_2 \leq \sigma_1 \textit{ and } \tau_1 \leq \tau_2 \textit{ then } \sigma_1 \to \tau_1 \leq \sigma_2 \to \tau_2$$
$$c(\sigma_1, \ldots, \sigma_n) \cap c(\tau_1, \ldots, \tau_n) \leq c(\sigma_1 \cap \tau_1, \ldots, \sigma_n \cap \tau_n)$$
$$\textit{if } c \leq_{\mathbb{C}} d \textit{ and } \sigma_i \leq \tau_i \textit{ for } i = 1 \ldots n \textit{ then } c(\sigma_1, \ldots, \sigma_n) \leq d(\tau_1, \ldots, \tau_n)$$

Similarly to notation in Section 3.2, we use $\equiv$ for syntactic identity and write $\sigma = \tau$ when $\sigma \leq \tau$ and $\tau \leq \sigma$, making $\leq$ a partial order.

The last two rules in above Definition 46 are motivated by practical use cases as shown in the following Example 48, Example 49, and Example 50. In particular, we can represent taxonomic domain knowledge by $\leq_{\mathbb{C}}$ and embed this knowledge into intersection type subtyping.

Example 48. *Let* $\texttt{Int}, \texttt{Real} \in \mathbb{C}_0$ *and* $\texttt{List}, \texttt{Seq} \in \mathbb{C}_1$ *such that* $\texttt{Int} \leq_{\mathbb{C}} \texttt{Real}$ *(integers can be treated as real numbers) and* $\texttt{List} \leq_{\mathbb{C}} \texttt{Seq}$ *(lists can be treated as sequences).*

We have that $\texttt{List}(\texttt{Int}) \leq \texttt{Seq}(\texttt{Real})$*, which describes that a list of integers can be treated as a sequence of real numbers.*

Example 49. *Let* $\texttt{Real}, \texttt{Celsius} \in \mathbb{C}_0$ *and* $\texttt{List} \in \mathbb{C}_1$*.*

We have that $\texttt{List}(\texttt{Real}) \cap \texttt{List}(\texttt{Celsius}) = \texttt{List}(\texttt{Real} \cap \texttt{Celsius})$*, i.e. if something is a list of real numbers and simultaneously a list of temperature measurements in degree Celsius, then each element of that list is a real number providing a temperature measurement in degree Celsius (and vice versa).*

Example 50. *Let* $\texttt{Real} \in \mathbb{C}_0$ *and* $\texttt{Pair} \in \mathbb{C}_2$.

We have that $\texttt{Pair}(\texttt{Real}, \texttt{Real}) \leq \texttt{Pair}(\texttt{Real}, \omega) \leq \texttt{Pair}(\omega, \omega)$*, i.e. a pair of real numbers can be treated as a pair, where we know that the first entry is a real number, or alternatively it can be treated as just a pair.*

Overall, intersection types with constructors have proved useful to represent a variety of features that may be included in type languages. Those features include product types [54], modal types used for staged composition synthesis [29], record types used for mixin composition synthesis [7], and generics used in object-oriented code [8].

The typing rules underlying the (CL)S-F# implementation (Definition 47) correspond to combinatory logic with intersection types (Definition 36), where we use the above Definition 46 of intersection type subtyping.

Definition 47 (Combinatory Logic with Intersection Types with Constructors, $\vdash_{\mathcal{C}(\mathbb{C})}$)**.**

$$\frac{S \text{ is a substitution}}{\Gamma, x : \sigma \vdash_{\mathcal{C}(\mathbb{C})} x : S(\sigma)}\ (\text{Ax}) \qquad \frac{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \sigma \to \tau \qquad \Gamma \vdash_{\mathcal{C}(\mathbb{C})} G : \sigma}{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F\ G : \tau}\ (\to\text{E})$$

$$\frac{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \sigma \qquad \Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \tau}{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \sigma \cap \tau}\ (\cap\text{I}) \qquad \frac{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \tau}\ (\leq)$$

(CL)S-F# by design does not introduce restrictions such as level or explicit codomains (so-called kinding) of substitutions. The guiding principle behind this decision is to shift the burden of inhabitant search to the implementation (relying on techniques from logic program evaluation) away from explicit user specification. Of course, inhabitation (Problem 18) in such an expressive calculus is undecidable (Theorem 15). Therefore, the actual implementation has to rely on sound but incomplete (i.e. potentially non-terminating) methods. Nevertheless, as we will see in Section 4.3, empiric evaluation shows that oftentimes the incomplete approach succeeds whereas the complete implementation requires an unreasonable amount of time.

Problem 18 (Inhabitation in $(\vdash_{\mathcal{C}(\mathbb{C})})$, $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$)**.** *Given a type environment* Γ *and a type* τ*, is there a combinatory term* F *such that* $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \tau$ *is derivable?*

Theorem 15. *Inhabitation in* $(\vdash_{\mathcal{C}(\mathbb{C})})$ *(Problem 18) is undecidable.*

Proof. The claim follows from undecidability of inhabitation in λ-calculus with intersection types [66] in its combinatory logic equivalen [27] using the fixed basis

$$\Gamma = \{\mathsf{S} : (\alpha_1 \to \beta \to \gamma) \to (\alpha_2 \to \beta) \to (\alpha_1 \cap \alpha_2) \to \gamma, \mathsf{K} : \alpha \to \beta \to \alpha\} \qquad \square$$

In practice, given a type environment Γ and a type τ we are not only interested in whether there exist combinatory terms F such that $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \tau$ holds, but want to know F. Additionally, there may be multiple such combinatory terms, in which case we are interested in the whole (possibly infinite) solution space of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$. Similarly to [55, Corollary 11], if the considered substitution space is finite, then the solution space of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$ constitutes a regular tree language [22] (Example 51).

Example 51. *Let* $\mathtt{0}, \mathtt{Nat} \in \mathbb{C}_0$, $\mathtt{S} \in \mathbb{C}_1$, *and*

$$\Gamma = \{z : \mathtt{0} \cap \mathtt{Nat}, s : (\mathtt{Nat} \to \mathtt{Nat}) \cap (\alpha \to \mathtt{S}(\alpha))\}$$

The solution space of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \mathtt{Nat}$ *is a regular tree language* $\{z, s(z), s(s(z)), ...\}$ *given by the following tree grammar with the starting symbol* $\mathtt{Nat}$

$$\mathtt{Nat} \longrightarrow z \mid s(\mathtt{Nat})$$

The solution space of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \mathtt{S(S(0))}$ *is a finite tree language* $\{s(s(z))\}$ *given by the following tree grammar with the starting symbol* $\mathtt{0}$

$$\begin{aligned} \mathtt{0} &\longrightarrow z \\ \mathtt{S(0)} &\longrightarrow s(\mathtt{0}) \\ \mathtt{S(S(0))} &\longrightarrow s(\mathtt{S(0)}) \end{aligned}$$

In the above Example 51 codomains of substitutions for the type variable α are not a priori bounded. However, the space of substitutions that are of relevance to determine the solutions space (even though it might be infinite) of the given queries is finite. Additionally, the above Example 51 provides an intuition that types are well-suited as non-terminals in tree grammars describing sets of inhabitants (generated by the corresponding type as starting symbol).

Since beta-soundness (Lemma 43) is essential to decide intersection type subtyping in quadratic time (Section 3.3.1), it is extended to type constructors (Lemma 47).

Lemma 47 (Extended Beta-Soundness)**.**
Given $\sigma = \bigcap_{i \in I} (\sigma_i \to \tau_i) \cap \bigcap_{j \in J} c_j(\sigma_1^j, \ldots, \sigma_{\mathrm{arity}(c_j)}^j) \cap \bigcap_{k \in K} \alpha_k$, *we have*

(*i*) *If* $\sigma \leq c(\sigma_1, \ldots, \sigma_n)$ *for some* $c \in \mathbb{C}_n$, *then* $J' = \{j \in J \mid c_j \leq_{\mathbb{C}} c\} \neq \emptyset$ *and* $\bigcap_{j \in J'} \sigma_i^j \leq \sigma_i$ *for* $i = 1 \ldots n$.

(*ii*) *If* $\sigma \leq \alpha$ *for some* $\alpha \in \mathbb{V}$, *then* $\alpha \equiv \alpha_k$ *for some* $k \in K$.

(*iii*) *If* $\sigma \leq \sigma' \to \tau' \neq \omega$ *for some* $\sigma', \tau' \in \mathbb{T}_{\mathbb{C}}^{\cap}$, *then* $I' = \{i \in I \mid \sigma' \leq \sigma_i\} \neq \emptyset$ *and* $\bigcap_{i \in I'} \tau_i \leq \tau'$.

Proof. (Sketch) Analogous to the proof of Lemma 43 where (i) is proven similarly to (iii) because of corresponding distribution properties. □

Additionally, the notion of paths and type organization is extended to type constructors as follows. Let us for $c \in \mathbb{C}_n$ and $i \in \{1, \ldots, n\}$ write $c(i := \tau)$ for $c(\sigma_1, \ldots, \sigma_n)$ where $\sigma_i \equiv \tau$, and $\sigma_j \equiv \omega$ for $j = 1 \ldots (i-1)$ and $j = (i+1) \ldots n$. In case $c \in \mathbb{C}_0$, we write $c(0 := \omega)$ for $c()$.

Definition 48 (Paths, $\mathbb{P}_{\mathbb{C}}^{\cap}$)**.** $\mathbb{P}_{\mathbb{C}}^{\cap} \ni \pi ::= \alpha \mid \sigma \to \pi \mid c(i := \omega) \mid c(i := \pi)$
where $c \in \mathbb{C}_n$ *and* $i \in \{1, \ldots, n-1\} \cup \{n\}$.

Lemma 48 (Type Organization)**.** *For any type* $\sigma \in \mathbb{T}_{\mathbb{C}}^{\cap}$, *an organized type* $\overline{\sigma} \in \mathbb{T}_{\mathbb{C}}^{\cap}$ *such that* $\sigma = \overline{\sigma}$ *can be computed in polynomial time by*

$$\overline{\alpha} \equiv \alpha \qquad \overline{\omega} \equiv \omega \qquad \overline{\sigma \cap \tau} \equiv \overline{\sigma} \cap \overline{\tau} \qquad \overline{\sigma \to \tau} \equiv \bigcap_{i \in I} (\sigma \to \pi_i) \text{ where } \overline{\tau} \equiv \bigcap_{i \in I} \pi_i$$

$$\overline{c(\sigma_1, \ldots, \sigma_n)} = c(n := \omega) \cap \bigcap_{j=1}^{n} \bigcap_{i \in I_j} \pi_i^j \text{ where } \overline{\sigma_j} \equiv \bigcap_{i \in I_j} \pi_i^j \text{ for } j = 1 \ldots n$$

4.2 Implementation and Techniques

This section gives an overview over algorithms realized as parts of (CL)S-F# [32]. It references implementations of key algorithms for intersection types, deciding intersection type subtyping, matching, and semi-deciding satisfiability. Since inhabitant search can be seen as evaluation of a logic programming language, techniques such as partial evaluation and sideways information passing constitute a fundamental part of (CL)S-F# and are outlined in this section.

(CL)S-F# is considered a testing ground for inhabitant search techniques rather than a complete synthesis framework. Therefore, this section aims to convey successful key ideas, and does not provide a full user documentation. For a developer documentation see [32]. For a comprehensive, user-oriented synthesis framework, the reader is referred to (CL)S-Scala [6].

Section Outline Section 4.2.1 describes the implementation of intersection types together with references to (semi) decision procedures for intersection type subtyping, matching and satisfiability. Partial evaluation in the context of inhabitant search is described in Section 4.2.3, and sideways information passing is described in Section 4.2.4. Finally, the inhabitant search interface is illustrated in Section 4.2.5 and its implementation is illustrated in Section 4.2.6.

4.2.1 Simplified Types

Intersection types with constructors $\mathbb{T}_{\mathbb{C}}^{\cap}$ (Definition 45), which constitute the type language of (CL)S-F#, are implemented in `Type.fs` as

```
type IntersectionType =
    | Var of ID
    | Arrow of IntersectionType * IntersectionType
    | Constructor of ID * IntersectionType list
    | Intersect of Set<IntersectionType>
```

where `type ID = string` is implemented in `ID.fs`.

The above implementation borrows two key aspects from modern intersection type presentations (cf. Definition 12). First, intersection has arbitrary arity and is definitionally treated as a set (modulo associativity, commutativity and idempotence) instead of relying on subtyping. Second, the universal type ω is presented as the empty intersection.

Although a stratification into strict types and intersection types (Definition 12) did not prove to be practical due to a potentially exponential blowup, several aspects of the stratified presentation are enforced algorithmically. In particular, `Intersect` constructors are never nested and the target of the `Arrow` constructor is never an empty intersection. We call types respecting those restrictions *simplified* types. Two simplified types that are equal modulo associativity, commutativity and idempotence of intersection are treated as syntactically equal. Additionally, the only simplified type equal to the universal type ω is the empty intersection. Although not changing the theoretical complexity, simplified types have proven effective in practice.

Example 52. *The type $c(\omega) \to (\alpha \cap \beta)$ is implemented in (CL)S-F# as*

```
Arrow(Constructor("c", [Intersect(Set.empty)]),
  Intersect(Set.ofList [Var "alpha"; Var "beta"]))
```

or alternatively as

```
Arrow(Constructor("c", [Intersect(Set.empty)]),
  Intersect(Set.ofList [Var "beta"; Var "alpha"]))
```

while those two implementations are simplified types and are treated as syntactically equal.

4.2.2 Simplified Type Subtyping

The three main problems related to intersection type subtyping are deciding the subtyping predicate, intersection type matching and intersection type satisfiability (cf. Section 3.3).

Subtyping A quadratic time algorithm to decide subtyping for intersection types with constructors is implemented in `Subtyping.fs` as

```
isSimplifiedSubType (isAtomicSubtype : ID → ID → bool) :
  IntersectionType → IntersectionType → bool
```

As discussed in Section 4.2.1 this implementation is correct for simplified types. The argument `isAtomicSubtype` captures the atomic subtyping predicate $\leq_{\mathbb{C}}$. The implementation corresponds to Algorithm SUB with the addition of type constructors and is based on the extended beta soundness property (Lemma 47).

Matching An exponential time algorithm to decide matching for intersection types with constructors is implemented in `Matching.fs` as

```
isMatchable (isAtomicSubtype : ID → ID → bool)
  (constraints : seq<IntersectionType * IntersectionType>) : bool
```

Again, the argument `isAtomicSubtype` is the atomic subtyping predicate $\leq_{\mathbb{C}}$, and the argument `constraints` is a sequence of pairs of simplified types such that at most one type per pair contains type variables.

The implementation relies on the following auxiliary method

```
enumerateMaximalBasicConstraintSystems (isAtomicSubtype : ID → ID → bool) :
 seq<IntersectionType * IntersectionType> → seq<BasicConstraintSystem>
```

which extends [28, `Match`] in two aspects. In practice, we are not just interested in a yes/no answer, but in so-called basic constraints (Definition 49) that concisely bound solutions wrt. individual variables.

Definition 49 (Basic Constraint [28, Definition 2])**.** *We call $\sigma \dot{\leq} \tau$ a* basic constraint*, if σ is a type variable and* $\mathrm{var}(\tau) = \emptyset$*, or τ is a type variable and* $\mathrm{var}(\sigma) = \emptyset$*.*

The method `enumerateMaximalBasicConstraintSystems` is used to enumerate consistent (in the sense of [28]) basis constraint sets that are equivalent to the input (non-basic) constraints. This allows a further inspection on lower and upper bounds of solutions (Example 53).

Example 53. *Consider the constraint $a \to \omega \to (a \cap b) \dot{\leq} \alpha \to \beta \to \alpha$, which by co-/contravariance of the arrow type constructor is equivalent to the set of basic constraints $\{\alpha \dot{\leq} a, \beta \dot{\leq} \omega, a \cap b \dot{\leq} \alpha\}$. The basic constraint $\beta \dot{\leq} \omega$ is trivially satisfied. The only solutions for $a \cap b \dot{\leq} \alpha \dot{\leq} a$ are S_1 such that $S_1(\alpha) = a$ and S_2 such that $S_2(\alpha) = a \cap b$.*

As outlined in Section 3.3.2, matching problem instances arise when specific user input such as $a \to \omega \to (a \cap b)$ is matched against schematic specification such as $\alpha \to \beta \to \alpha$ in the above Example 53. Using basic constraint sets generated by `enumerateMaximalBasicConstraintSystems` we may narrow down substitution codomains useful for the inhabitant search algorithm without additional user specification (for example as variable kinding annotations).

Satisfiability A semi-algorithm to decide satisfiability for intersection types with constructors is implemented in `Satisfiability.fs` as

```
enumerateSatisfyingSubstitutions (isAtomicSubtype : ID → ID → bool)
  (substitutableVariables : Map<ID, Variance>)
  (constraints : seq<IntersectionType * IntersectionType>)
    : seq<Substitution>
```

In addition to the atomic subtype predicate $\leq_{\mathbb{C}}$ a sequence of simplified type constraints, the above implementation takes type variable variance as the argument `substitutableVariables`. This additional information is useful to guide solution enumeration with respect to intended use of occurring variables (Example 54).

Example 54. *Let* $\Gamma \supseteq \{\mathsf{B} : (\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \mathsf{I} : \delta \to \delta\}$. *Let us inspect inhabitants that use* B *for function composition with identity, i.e. inhabitants that have* $(\mathsf{B}\,\mathsf{I}\,F)$ *as a subterm for some combinatory term* F. *For that reason, we may consider the satisfiability constraint* $\delta \to \delta \leq \beta \to \gamma$ *which is equivalent to* $\beta \doteq \delta \doteq \gamma$. *Additionally, we need to derive* $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : S(\alpha \to \beta)$ *for substitutions* S *that solve* $\beta \doteq \delta \doteq \gamma$. *Observe that* β *appears positively in* $\alpha \to \beta$ *and is otherwise only constrained by* $\beta \doteq \delta \doteq \gamma$. *Therefore, we do not lose solutions choosing substitutions for* β *(and* δ*) as large as possible wrt.* $\leq$.

Decidability of intersection type satisfiability is unknown, and we yet have little understanding of algorithms for this decision problem. Therefore, the implementation of the method `enumerateSatisfyingSubstitutions` is rudimentary and relies on similar methodology as `enumerateMaximalBasicConstraintSystems`.

4.2.3 Partial Evaluation

Viewing inhabitant search as execution of a logic program given by a collection of typed combinators, it is natural to consider evaluation techniques known in the area of logic and functional programming. In this section we consider the technique of *partial evaluation* [46] to improve inhabitant search.

A *partial evaluator* [46] (or, *program specializer*) for a programming language, given a program p and a partial input v, constructs a specialized program p_v such that for all inputs w we have $[\![p]\!](v, w) = [\![p_v]\!](w)$, where $[\![\_]\!]$ denotes the functional interpretation of programs. A practical example for partial evaluation is regular expression recognition. As soon as a particular regular expression is fixed, a faster, specialized recognizer can be generated.

In the setting of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$, the type environment Γ corresponds to the program p, the type τ corresponds to v and w, and inhabitant search corresponds to the interpretation $[\![\_]\!]$.

There are two key aspects of successful partial evaluation in our setting. First, identifying where partial information becomes available. Second, generating a better program based on the partial information. At first sight, it may not appear to be a common scenario where parts of the goal type τ are known a priori. However, inhabitant search algorithms (cf. [54, Figure 5]) commonly generate recursive subgoals based on combinator argument types which may contain partially known information regardless of user input. For such subgoals inhabitant search may be specialized (Example 55).

Example 55. *Let* $\Gamma = \{x : (\alpha \to a) \to b, y : (b \cap \beta) \to a\}$. *During inhabitant search we may consider applying the combinator* y *to one argument leading to a recursive subgoal* $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : b \cap \tau$, *where* τ *is an instance of* β. *In this case we know the partial input* b *(while* τ *may vary) and can restrict inhabitant search to consider only* $\Gamma_b = \{x : (\alpha \to a) \to b\}$, *excluding the combinator* y, *for such subgoals. Of course,* y *may need to be included again at a later point.*

The above Example 55 provides two insights. First, if the (sub)goal type is partially known, then we may determine a subset of the type environment Γ relevant for that subgoal. Second, inspecting the type environment Γ we may consider combinator argument types (which may be partially known) as potential subgoals and specialize Γ excluding any combinators not relevant in presence of the partially known information.

This technique is successful for type environments that encode automata-like behavior with low fan-out (Section 4.3.4). Since the encoded transition relation is "local", and at each step most of the type environment is not relevant.

The implementation of the outlined partial evaluation approach is found in `InhabitationUtil.fs` in the method `preprocessEnvironment`

```
preprocessEnvironment (atomicSubtypes : ID → ID → bool)
  (environment : Environment) : EnvironmentEntry list
```

where the record `EnvironmentEntry` contains in the field `RelevantEnvironments` for each potential argument of the corresponding combinator a subset of the current type environment that is relevant for that argument based on partial information. Whether a combinator is relevant for some argument is decided by the method `isPossiblySatisfiable` implemented in `Orthogonality.fs`.

4.2.4 Sideways Information Passing

By similar motivation as in Section 4.2.3, in this section we discuss *sideways information passing* [5] known from logic programming language evaluation.

In logic programming, there are two kinds of information passing [5]. First, unification with the current goal propagates variable instantiation from the rule head to the rule body. Second, by evaluating a partially defined predicate inside the rule body, new variable instantiations may arise. Those new instantiations are passed sideways to an other predicate inside the rule body.

In the remainder of this section we illustrate both kinds of information passing in context of inhabitant search.

For the remainder of this section, let $\mathtt{0}, \bullet \in \mathbb{C}_0$, $\mathtt{num}, \mathtt{s} \in \mathbb{C}_1$, $\mathtt{isSucc} \in \mathbb{C}_2$, and

$$\begin{aligned}\Gamma = \{&z : \mathtt{num}(\mathtt{0}),\\ &s : \mathtt{num}(\alpha) \to \mathtt{isSucc}(\alpha \to \bullet, \mathtt{s}(\alpha)),\\ &n : \mathtt{isSucc}(\beta \to \bullet, \gamma) \to \mathtt{num}(\beta) \to \mathtt{num}(\gamma)\}\end{aligned}$$

Intuitively, the above type environment Γ describes natural numbers with the predicate $\mathtt{num}$ such that $\mathtt{num}(\mathtt{0})$, $\mathtt{num}(\mathtt{s}(\mathtt{0}))$, ... are inhabited, and the predicate $\mathtt{isSucc}$ such that $\mathtt{isSucc}(\mathtt{0} \to \bullet, \mathtt{s}(\mathtt{0}))$, $\mathtt{isSucc}(\mathtt{s}(\mathtt{0}) \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0})))$, ... are inhabited. The combinator z states that $\mathtt{0}$ is a natural number; the combinator s states that the successor of a natural number α is $\mathtt{s}(\alpha)$; the combinator n states that if γ is a successor of a natural number β, then γ is a natural number.

The following Example 56 illustrates the first kind of information passing.

Example 56. *Consider the instance* $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ *of the inhabitation problem. The only combinator that (applied to some arguments) is typable by* $\mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ *is* n *applied to two arguments. Therefore, we are interested in substitutions* $S_1, \ldots, S_m$ *such that*

$$\bigcap_{i=1}^{m} S_i(\mathtt{isSucc}(\beta \to \bullet, \gamma) \to \mathtt{num}(\beta) \to \mathtt{num}(\gamma)) \leq \sigma_1 \to \sigma_2 \to \mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$$

where the types σ_1, σ_2 *are inhabited in* Γ. *By Lemma 47, one substitution is sufficient, leading to the following set of constraints*

$$\{\delta_1 \stackrel{\cdot}{\leq} \mathtt{isSucc}(\beta \to \bullet, \gamma), \delta_2 \stackrel{\cdot}{\leq} \mathtt{num}(\beta), \mathtt{num}(\gamma) \stackrel{\cdot}{\leq} \mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))\}$$

where instances of δ_1 *(resp.* δ_2*) correspond to types* σ_1 *(resp.* σ_2*). Due to the typing rule* $(\leq)$ *we are interested in the largest possible instances of* δ_1 *(and therefore* γ*) and* δ_2 *wrt. subtyping (cf. Example 54). Since we are free to choose the maximal solution* $\gamma \mapsto \mathtt{s}(\mathtt{s}(\mathtt{0}))$*, we do not lose inhabitants by specializing the type of* n *to*

$$\mathtt{isSucc}(\beta \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0}))) \to \mathtt{num}(\beta) \to \mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$$

Overall, information is passed from the right-hand side of the arrow type constructor to a recursive subgoal on the left-hand side.

In the above Example 56 the type variable β constitutes a *cut-type* in $\mathtt{isSucc}(\beta \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0}))) \to \mathtt{num}(\beta) \to \mathtt{num}(\mathtt{s}(\mathtt{s}(\mathtt{0})))$ because it appears only in

argument positions. In the worst case scenario, the correct instantiation of β is arbitrary, leading to an intractable search space. However, in the scenario at hand we can use the second kind of information passing to instantiate β as illustrated by the following Example 57

Example 57. *We are interested in types σ such that both the type $\mathtt{num}(\sigma)$ and the type $\mathtt{isSucc}(\sigma \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0})))$ are inhabited in Γ. Necessarily, inhabitants of the type $\mathtt{isSucc}(\sigma \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0})))$ in Γ are of shape $(s\,F)$ for some combinatory term F such that*

$$\bigcap_{i=1}^{m} S_i(\mathtt{isSucc}(\alpha \to \bullet, \mathtt{s}(\alpha))) \leq \mathtt{isSucc}(\sigma \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0})))$$

for some substitutions $S_1, \ldots, S_m$ and $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} F : \bigcap_{i=1}^{m} S_i(\mathtt{num}(\alpha))$.

By Lemma 47, one substitution S is sufficient. Therefore, after simplification, we are interested in solutions of the satisfiability problem instance $\{\beta \dot{\leq} \alpha, \alpha \dot{\leq} \mathtt{s}(\mathtt{0})\}$ where β represents σ. By a variance argument (cf. Example 56) it suffices to consider the largest solution wrt. intersection type subtyping, i.e. $S(\alpha) = S(\beta) = \mathtt{s}(\mathtt{0})$.

As a result, by inspection of inhabitants of the type $\mathtt{isSucc}(\sigma \to \bullet, \mathtt{s}(\mathtt{s}(\mathtt{0})))$ we obtain restrictions on the shape of σ, and information is passed sideways to the argument $\mathtt{num}(\sigma)$, reducing search space.

Complementary to Example 57 (passing information from the type environment sideways via an argument), the following Example 58 demonstrates a different sideways information passing technique that utilizes intersection type matching within a type to restrict search space.

Example 58. *Let $a, b, c, \bullet \in \mathbb{C}_0$, and*

$$\Gamma = \{x : a \to \bullet, y : b \to \bullet, z : (a \to \bullet) \cap (\alpha \to \bullet) \to (b \to \bullet) \cap (\alpha \to \bullet) \to c\}$$

Inhabitants of the type c in Γ are necessarily of shape $(z\,F\,G)$ for some combinatory terms F and G. Since the type variable α appears only in argument positions in the type of z, an inhabitant search procedure has to guess a correct instantiation. Inspecting the arguments individually (cf. Example 57), the substitutions $S_1(\alpha) = a$ and $S_2(\alpha) = b$ appear relevant. However, for both substitutions only one of the two argument types of z is inhabited.

Observe that for any type σ, due to intersection type subtyping, any inhabitant of $(a \to \bullet) \cap (\sigma \to \bullet)$ (resp. $(a \to \bullet) \cap (\sigma \to \bullet)$) is also an inhabitant of $a \to \bullet$ (resp. $b \to \bullet$). Therefore, if the instance $\{\alpha \to \bullet \dot{\leq} a \to \bullet, \alpha \to \bullet \dot{\leq} b \to \bullet\}$ of the intersection type matching problem has any solution, then there exists a substitution for α such that the subformulae $\alpha \to \bullet$ are immaterial for inhabitant search. In fact, the above set of constraints is solved by $\alpha \mapsto a \cap b$.

Overall, by intersection type matching we know that there exists a single substitution S that subsumes the solution space and eliminates the type variable α. This is effective in combination with other information passing techniques that provide partial information.

Overall Examples 56–58 illustrate effective use of intersection type matching and unification to realize information passing techniques known from logic program evaluation. Unfortunately, decidability of intersection type unification is still an open problem.

4.2.5 Inhabitant Search Interface

In this section we give an overview over the inhabitant search interface of (CL)S-F#. Overall, (CL)S-F# inhabitant search is divided in three phases: type environment initialization, tree grammar construction and inhabitant construction. Key for the first two phases of inhabitant search is the following method

```
getAllInhabitants (maxTreeDepth : int) (logger : ILogger)
  (atomicSubtypes : ID → ID → bool) (environment : Environment)
    : (IntersectionType → TreeGrammar)
```

implemented in `Inhabitation.fs` where

```
type ILogger = int → Lazy<string> → unit
type Environment = list<ID * IntersectionType>
type TreeGrammar = Map<IntersectionType, Set<CombinatorExpression>>
type CombinatorExpression = (ID * IntersectionType list)
```

The arguments `maxTreeDepth` and `logger` are of bookkeeping nature (to restrict stack memory use and provide a side-effect for logging evens in explored subgoals). The argument `atomicSubtypes` is used to pass the atomic subtyping predicate $\leq_{\mathbb{C}}$. Finally, given the argument `environment`, which corresponds to the type environment Γ (implemented as a list of pairs containing combinator name and assigned type), we obtain a function of type `IntersectionType` $\rightarrow$ `TreeGrammar` mapping a given type τ to the solution space of $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$ (implemented as a possibly empty normalized regular tree grammar [22, Proposition 2.1.4]). As exemplified in Section 4.1, types constitute non-terminals of the computed tree grammar. The tree language generated from each non-terminal σ corresponds to combinatory terms typable by σ in the type environment Γ.

Actual inhabitants are computed in the third phase from a tree grammar by the method `listMinimalCombinatorTerms` implemented in `CombinatorTerm.fs`

```
listMinimalCombinatorTerms (numberOfTerms : int) (grammar : TreeGrammar)
  (nonTerminal : IntersectionType) : list<CombinatorTerm>
```

The above method `listMinimalCombinatorTerms` constructs a list of length at most `numberOfTerms` containing minimal (wrt. number of nodes in the syntax tree) combinatory terms derivable from the symbol `nonTerminal` in the grammar `grammar`. The constructed list of combinatory terms is considered the result for the query $\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \tau$ in (CL)S-F#.

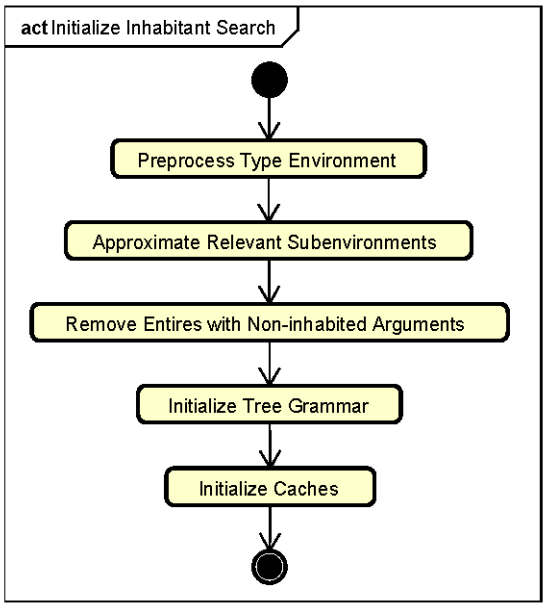
4.2.6 Inhabitant Search Implementation

While Section 4.2.5 describes the developer interface to inhabitant search exposed by (CL)S-F#, this section outlines the basic structure of the implementation of that interface.

As described in Section 4.2.5, (CL)S-F# inhabitant search is divided in three phases.

In the first phase, the method `getAllInhabitants`, which is implemented in `Inhabitation.fs`, given some bookkeeping parameters and a type environment Γ results in a function that maps a goal type τ to a tree grammar presentation of inhabitants of τ in Γ. Figure 4.1 outlines the structure of this initialization procedure, that includes an inspection of the given type environment according to partial evaluation techniques described in Section 4.2.3. In particular, arguments of type assumptions are associated with relevant entries of the given type environment. The implementation can be found in `InhabitationUtil.fs` as the method `preprocessEnvironment`. Most importantly, initialization is done once, and is independent from the goal type. This allows to reuse the initialized method for multiple queries sharing the same type environment. For example, this is useful to enumerate multiple products of a product line that share a common code base (type environment) but differ in specification (goal type).

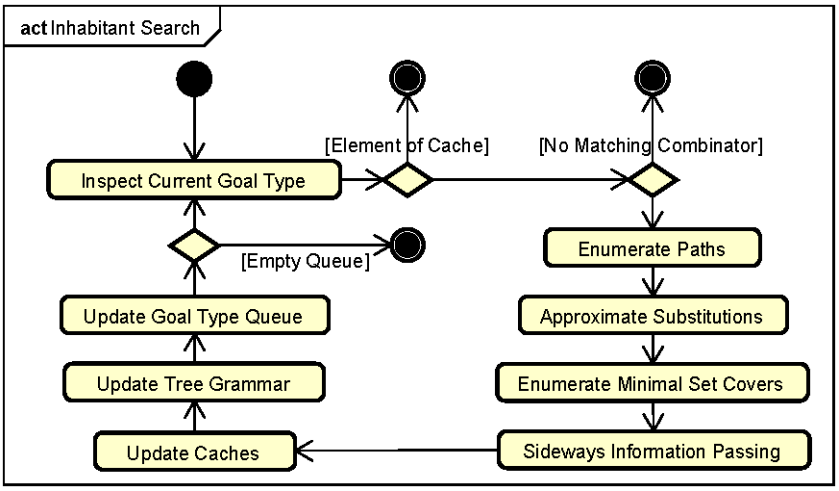
Figure 4.1: Initialize Inhabitant Search



The second phase is based on the method returned by `getAllInhabitants` that maps a given goal type τ to a tree grammar presentation of inhabitants of τ in the previously fixed type environment Γ. Figure 4.2 outlines the structure of this procedure (presented iteratively). The key component to inhabitant search is, similarly to the alternating decision procedure in [54, Figure 5], the coverage of paths (Definition 48) in the organized (Lemma 48) presentation of the current goal type. Instead of a priori fixing a substitution space, (CL)S-F#

narrows it step by step using methods described in Section 4.2.4. The non-deterministic choice of relevant paths is abstracted via the minimal (wrt. set inclusion) set cover problem. Specifically, a path π of a goal type is covered by some combinator type ρ, if $S(\rho) \leq \pi$ for some substitution S. Therefore, covering all paths of a given goal type τ corresponds to $\bigcap_{i=1}^{m} S_i(\rho) \leq \tau$. Naturally, considering only minimal covers is sufficient. Although Figure 4.2 outlines the algorithm iteratively, the actual implementation is (mutually) recursive and manages the queue implicitly in the invocation stack.

Figure 4.2: Inhabitant Search



The final phase of inhabitant search is to enumerate particular inhabitants presented via a tree grammar. The used approach is based on dynamic programming, computing for every non-terminal (intersection type) the k minimal (wrt. nodes in the syntax tree) trees (combinatory terms). It is a single loop that computes larger (minimal) trees from smaller (minimal) ones, and terminates when the currently computed k minimal trees for all non-terminals do not change. The implementation is located in `CombinatorTerm.fs` as the method `listMinimalCombinatorTerms`.

4.3 Evaluation

This section provides an overview over several key examples studied during the development of (CL)S-F#, each accompanied by a performance evaluation[3]. All described examples (among others) are implemented and documented in the `cls-fsharp-experiments` project [32]. The choice of examples is motivated by two factors. First, the presented examples give a broad overview over modeling features that (CL)S-F# supports. Second, the presented examples in their generality are either infeasible or do not scale well using previous combinatory logic synthesis implementations.

Section Outline First, in Section 4.3.1 a service composition example [54, Section 4.3] is examined. This example uses the universal type ω as a "don't care" placeholder, and relies on generally formulated component specification. Due to the general specification, type variable instantiation cannot be bounded a priori. Therefore, this example is among the most difficult ones for combinatory logic synthesis implementations.

Second, in Section 4.3.2 we shift our focus to an object-oriented scenario of [7]. The examples presented in Section 4.3.2 illustrate the use of covariant distributing constructors to represent and manipulate object-oriented programs.

Third, in Section 4.3.3 a product line scenario is described encompassing robot control programs that are automatically synthesized, deployed, and executed [10]. While the example in Section 4.3.3 is larger than the previous ones, it also includes taxonomically structured domain knowledge (specified by the atomic subtype predicate) and relies on metaprogramming (specified by modal types) to compose process fragments.

Fourth, in Section 4.3.4 we examine in a deterministic automation simulation scenario scaling properties of (CL)S-F#. The examples in Section 4.3.4 illustrate inhabitant search scaling, reaching inhabitants that contain thousands of combinators.

Fifth, complementary to the deterministic scenario in Section 4.3.4, in Section 4.3.5 we examine scaling properties of (CL)S-F# in a non-deterministic path finding scenario.

[3] Intel Core i7–4790 CPU, 8.00 GB RAM

4.3.1 Service Composition

In his visionary proposal [54], Rehof illustrates (among other examples) combinatory logic synthesis in the setting of service composition [54, Section 4.3]. This example stands out due to its use of mostly generally specified components, such as functional composition of type $(\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma$. Traditionally, such general specification poses a difficulty for inhabitant search. Additionally, the example uses the universal type ω as a "don't care" placeholder. Since any previous (CL)S implementation failed to master this particular example in its general form, solving it has been the main motivation behind (CL)S-F#.

In the following, we provide an overview over the service composition example from [54, Section 4.3] and its evaluation in (CL)S-F#.
Let $\mathtt{Int}, \mathtt{Bool}, \mathtt{Filter}, \mathtt{Sorted}, \mathtt{TopSorted}, \mathtt{TotalOrder}, \mathtt{PartialOrder}, \mathtt{Task}, \mathtt{SessionID}, \mathtt{UserID}, \mathtt{TID}, \mathtt{Result} \in \mathbb{C}_0$, $\mathtt{Graph}, [\cdot] \in \mathbb{C}_1$, and

$$
\begin{aligned}
\Gamma = \{ F : \quad & ([\alpha] \to (\alpha \to \mathtt{Bool}) \to [\alpha]) \cap \mathtt{Filter}, \\
S : \quad & (([\alpha] \to (\alpha \to \mathtt{Bool}) \to [\alpha]) \cap \mathtt{Filter}) \\
& \quad \to \big(((\alpha \to \alpha \to \mathtt{Bool}) \to [\alpha] \to [\alpha]) \\
& \qquad \cap (\mathtt{TotalOrder} \to \omega \to \mathtt{Sorted}) \\
& \qquad \cap (\mathtt{PartialOrder} \to \omega \to \mathtt{TopSorted})\big), \\
G : \quad & \mathtt{Graph}(\alpha) \to ((\alpha \to \alpha \to \mathtt{Bool}) \cap \mathtt{PartialOrder}), \\
N : \quad & \mathtt{Graph}(\alpha) \to [\alpha], \\
\circ : \quad & (\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma, \\
\diamond : \quad & (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma, \\
\mathit{Connect} : \quad & \mathtt{Int} \cap \mathtt{SessionID}, \\
\mathit{ReqTransaction} : \quad & (\mathtt{Int} \cap \mathtt{UserID}) \to (\mathtt{Int} \cap \mathtt{SessionID}) \\
& \quad \to ([\mathtt{Task}] \cap \mathtt{TopSorted}) \to (\mathtt{Int} \cap \mathtt{TID}), \\
\mathit{EndTransaction} : \quad & (\mathtt{Int} \cap \mathtt{TID}) \to (\mathtt{Int} \cap [\mathtt{Result}]), \\
\mathit{MyID} : \quad & \mathtt{Int} \cap \mathtt{UserID}, \\
\mathit{GetTasks} : \quad & \mathtt{Graph}(\mathtt{Task}) \}
\end{aligned}
$$

Intuitively, the combinator F implements a list filter function parameterized over a predicate; the combinator S implements a list sorting function parameterized over a total order (or, alternatively, a partial order); the combinator G constructs a partial order from a graph; and the combinator N linearizes a graph. The combinators $\circ$ and $\diamond$ are generic function composition and application in context. The combinators *Connect*, *ReqTransaction*, *EndTransaction*, *MyID*, and *GetTasks* are protocol specific combinators to manage users, sessions and tasks.

In [54, Section 4.3] the query

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : [\mathtt{Int} \cap \mathtt{Result}]$$

is answered positively by the combinatory term

$$((\mathit{ReqTransaction}\ \mathit{MyID}\ \mathit{Connect}) \circ ((G \circ (S\ F)) \diamond N) \circ \mathit{EndTransaction})\ \mathit{GetTasks}$$

Interestingly, the above combinatory term is not found by (CL)S-F# because in the original example in [54, Section 4.3] there appears to be a mixup in use of

the combinator $\circ$. An inhabitant found by (CL)S-F# in 2.1 seconds is

$$((((G \circ (S\,F)) \diamond N) \circ (\mathit{ReqTransaction}\,\mathit{MyID}\,\mathit{Connect})) \circ \mathit{EndTransaction})\ \mathit{GetTasks}$$

Alternatively, (CL)S-F# suggests the following solution that does not contain any generic combinators, and is easier to verify by hand

$$\mathit{EndTransaction}\,(\mathit{ReqTransaction}\,\mathit{MyID}\,\mathit{Connect}\,M)$$
$$\text{where } M = S\,F\,(G\,\mathit{GetTasks})\,(N\,\mathit{GetTasks})$$

The combinator *GetTasks* appears twice in the above inhabitant (in contrast to the previous one). This could be a problem in a real world scenario, where resources are linear or bound to side effects.

After removing the combinator *GetTasks* from Γ and adjusting the goal type to be `Graph(Task)` $\rightarrow$ [`Int` $\cap$ `Result`], (CL)S-F# discovers in 2.2 seconds the inhabitant

$$(((G \circ (S\,F)) \diamond N) \circ (\mathit{ReqTransaction}\,\mathit{MyID}\,\mathit{Connect})) \circ \mathit{EndTransaction}$$

which is, in spirit, the favored solution.

The above queries are implemented in `ServiceComposition.fs`.

Overall, combining inhabitant search strategies described in Section 4.2 (CL)S-F# is able to positively answer queries that contain generic higher-order functional specification that (even for humans) require a non-trivial amount of work. It is important to point out that (CL)S-F# does not require the user to restrict substitutions of individual type variables (e.g. specifying that some variable can only be substituted by either `Graph(Task)` or [`Task`]). Therefore, correct type variable instantiation in the above examples is discovered autonomously by the inhabitant search procedure.

4.3.2 Mixin Composition

Combinatory logic synthesis is successfully used in [7] in an object-oriented setting. In particular, *mixins* (functions from classes to classes) are exposed as typed combinators and composed functionally. In this section we examine two key examples from [7]. Whereas the first example [7, Section 5.2] uses types derived in the λ-calculus with records directly, the second example [7, Section 5.3] enriches the specification with semantic information. Most importantly, record types coincide with distributing covariant type constructors, and are therefore easily represented in (CL)S-F#.

In the first part of this section, let us consider the type constructors $\texttt{Int}$, $\texttt{Bool} \in \mathbb{C}_0$, $\langle\cdot\rangle, \texttt{get}, \texttt{set}, \texttt{succ}, \texttt{succ2}, \texttt{compare} \in \mathbb{C}_1$, and the type environment

$$\begin{aligned}
\Gamma = \{ \mathit{Num}: \quad & \texttt{Int} \to \langle \texttt{get}(\texttt{Int}) \cap \texttt{set}(\texttt{Int} \to \texttt{Int}) \cap \texttt{succ}(\texttt{Int}) \rangle, \\
\mathit{Comparable}: \quad & \big((\texttt{Int} \to \langle \texttt{get}(\texttt{Int}) \rangle) \\
& \quad \to (\texttt{Int} \to \langle \texttt{compare}(\langle \texttt{get}(\texttt{Int}) \rangle \to \texttt{Bool}) \rangle) \big) \\
& \cap \sigma_{\texttt{get}} \cap \sigma_{\texttt{set}} \cap \sigma_{\texttt{succ}} \cap \sigma_{\texttt{succ2}}, \\
\mathit{Succ2}: \quad & \big((\texttt{Int} \to \langle \texttt{succ}(\texttt{Int}) \rangle) \to (\texttt{Int} \to \langle \texttt{succ2}(\texttt{Int}) \rangle) \big) \\
& \cap \sigma_{\texttt{get}} \cap \sigma_{\texttt{set}} \cap \sigma_{\texttt{succ}} \cap \sigma_{\texttt{compare}} \}
\end{aligned}$$

where $\sigma_{\texttt{v}} = (\texttt{Int} \to \langle \texttt{v}(\alpha_{\texttt{v}}) \rangle) \to (\texttt{Int} \to \langle \texttt{v}(\alpha_{\texttt{v}}) \rangle)$.

Intuitively, *Num* is a class encapsulating a number and has methods to get, set and increment that number. *Comparable* (resp. *Succ2*) is a mixins that may add the method `compare` (resp. `succ2`) to a given class that has a `get` (resp. `succ`) method.

In [7, Section 5.2] the query

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \texttt{Int} \to \langle \texttt{succ}(\texttt{Int}) \cap \texttt{compare}(\langle \texttt{get}(\texttt{Int}) \rangle \to \texttt{Bool}) \cap \texttt{succ2}(\texttt{Int}) \rangle$$

is answered positively by the combinatory term $\mathit{Succ2}\,(\mathit{Comparable}\,\mathit{Num})$.

The above query (together with other examples used in [7]) is implemented in `RecordCalculus.fs` and is answered correctly in 0.1 seconds by (CL)S-F#. Although the above query appears simplistic, it relies on (alongside records) a difficult to bound search space. In particular, applying *Succ2* requires to instantiate the type variable $\alpha_{\texttt{compare}}$ in $\sigma_{\texttt{compare}}$ by the type $\langle \texttt{get}(\texttt{Int}) \rangle \to \texttt{Bool}$, which emerges only after having applied the combinator *Comparable*. Therefore, it is not advisable to bound variable instantiation a priori. Information passing techniques (Section 4.2.4) are useful in this scenario.

In the second part of this section, let us consider the type constructors $\texttt{String}, \texttt{plain}, \texttt{time} \in \mathbb{C}_0$, $\langle\cdot\rangle, \texttt{get}, \texttt{enc}, \texttt{sign} \in \mathbb{C}_1$, and the type environment

$$\begin{aligned}
\Gamma = \{ \mathit{Reader}: \quad & \texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \texttt{plain}) \rangle, \\
\mathit{Enc}: \quad & (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \alpha) \rangle) \\
& \quad \to (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \texttt{enc}(\alpha)) \rangle), \\
\mathit{Sign}: \quad & (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \alpha) \rangle) \\
& \quad \to (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \alpha \cap \texttt{sign}(\alpha)) \rangle), \\
\mathit{Time}: \quad & (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \alpha) \rangle) \\
& \quad \to (\texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \alpha \cap \texttt{time}) \rangle) \}
\end{aligned}$$

Intuitively, *Reader* is a class encapsulating plain text. *Enc*, *Sign*, and *Time* are mixins that encrypt, sign and add a time stamp to the contents of a given class. Type constructors `plain`, `time`, `enc`, `sign` are added to the specification to expose the intuitive meaning of individual mixins, which otherwise would be typed identically, to inhabitant search.

In [7, Section 5.3] the query

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \texttt{String} \to \langle \texttt{get}(\texttt{String} \cap \texttt{enc}(\texttt{plain} \cap \texttt{time} \cap \texttt{sign}(\texttt{plain} \cap \texttt{time})))\rangle$$

is answered positively by the combinatory term $Enc(Sign(Time(Reader)))$.

The above query (together with other examples used in [7]) is implemented in `RecordCalculus.fs` and is answered correctly in 0.2 seconds by (CL)S-F#.

Similarly to the previous example, the type variable α is used to carry emergent semantic information as arbitrary nested intersections of types, and instances of α cannot be bounded a priori. Again, information passing techniques (Section 4.2.4) are essential in this scenario.

4.3.3 Process Synthesis

Combinatory logic synthesis has been successfully applied to compose BPMN[4] processes [10] that can be automatically deployed and executed on LEGO Mindstorms NXT robots. In [10] a product line of robot control programs is developed that considers distinguished physical features of a robot (i.e. varying sensor equipment). As a result, synthesized robot control programs are tailored for specific robot configurations.

From the theoretical perspective, examples in [10] differ from previous ones in the following two aspects.

First, examples in [10] include taxonomically structured domain knowledge (e.g. a car-robot is a wheeled robot), that has to be taken into account during inhabitant search. This aspect is covered by the user defined atomic subtype predicate $\leq_{\mathbb{C}}$ in (CL)S-F# (cf. Section 4.2.2).

Second, examples in [10] use modal types $\Box\tau$ that carry the meaning of "code of type τ". This allows to include metaprograms that compose program text. Since properties of the modal type constructor $\Box$, that are relevant for the particular example, coincide with covariant distributing unary constructors, (CL)S-F# is capable of representing such types.

The main contribution of [10] is an empiric evaluation of combinatory logic synthesis in a realistic scenario. Therefore, presenting the examples in full is out of proportion in this work. The type environment Γ in the main example in [10, Section 3] contains around 40 combinators exposing process fragments and process composition methods for a product line encompassing around 50 different robot control programs. For example, Γ contains the combinator

$$\mathit{taskToSubProc} : \Box(\mathtt{task} \cap \alpha) \to \Box(\mathtt{subproc} \cap \alpha)$$

that represents a metaprogram which transforms program text of a BPMN process task (with some additional properties α) into program text of a BPMN subprocess (preserving the properties α). BPMN itself has no means to perform such a transformation, therefore metaprogramming is required.

The main example in [10, Section 3] contains the following query

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \Box(\mathtt{proc} \cap \mathtt{car} \cap \mathtt{followsLine} \cap \mathtt{twoLightSensors} \cap \mathtt{stopsOnTouch}) \cap \mathtt{robotProgram}$$

Answering the above query and executing the resulting metaprogram produces a deployable BPMN process (around 200 lines of code) for a car-robot which follows a line using two light sensors and stops when touched.

The full query is implemented in `ProcessSynthesis.fs` in (CL)S-F# and is answered correctly in 0.8 seconds. Compared to Section 4.3.1 and Section 4.3.2, while the given type environment is larger, component specification is more restricted and variable instantiation is atomic by design. Therefore, search strategies described in Section 4.2 (with the exception of partial evaluation) are unnecessary in this scenario.

[4] `http://www.bpmn.org/`

4.3.4 Two Counter Automaton Simulation

In order to illustrate expressiveness of combinatory logic synthesis, Rehof in [54, Section 2.3] provides a reduction from the (undecidable) two counter automaton halting problem to inhabitation in the simply typed combinatory logic (cf. Section 3.1). Two counter automata extend finite state automata by two counters containing natural numbers, where state transition is performed while incrementing or decrementing a counter, or testing whether it is zero.

At first glance, simulation of two counter automata by inhabitant search is mostly of theoretical interest. However, two counter automata concisely encompass features (such as state, locality, and natural number arithmetic) that are frequently encountered in real world scenarios. Additionally, even for simple problems, such as checking parity of a binary representation of a natural number (Figure 4.3), two counter automata tend be large and computationally inefficient. Therefore, two counter automata are suitable to explore scaling of (CL)S-F# with larger queries.

For the following example we use the two counter automaton given in Figure 4.3, where the counters are named c and d. The automaton accepts, if the number of ones in the binary representation of the input number initially stored in the counter c is even.

In the type environment Γ, that represents the automaton, each state transition is represented by a combinator. For example, the transition from state p to state $p1$ decrementing the counter c in Figure 4.3 is represented by the combinator

$$\mathit{DEC\_c\_p\_p1} : (\mathtt{c}(\alpha) \to \mathtt{d}(\beta) \to \mathtt{p1}) \to (\mathtt{c}(\mathtt{s}(\alpha)) \to \mathtt{d}(\beta) \to \mathtt{p})$$

where $\mathtt{p}, \mathtt{p1} \in \mathbb{C}_0$ and $\mathtt{c}, \mathtt{d}, \mathtt{s} \in \mathbb{C}_1$.

Intuitively, inhabitant search simulates a transition

$$(c = \alpha + 1, d = \beta, \text{state} = p) \mapsto (c = \alpha, d = \beta, \text{state} = p1)$$

of the automaton by searching an inhabitant for $(\mathtt{c}(\alpha) \to \mathtt{d}(\beta) \to \mathtt{p1})$ in order to find an inhabitant for $(\mathtt{c}(\mathtt{s}(\alpha)) \to \mathtt{d}(\beta) \to \mathtt{p})$. Accordingly, the representation of final state of the automaton is as follows

$$\mathit{FIN\_twoCounter\_accept} : (\mathtt{c}(\alpha) \to \mathtt{d}(\beta) \to \mathtt{twoCounter\_accept})$$

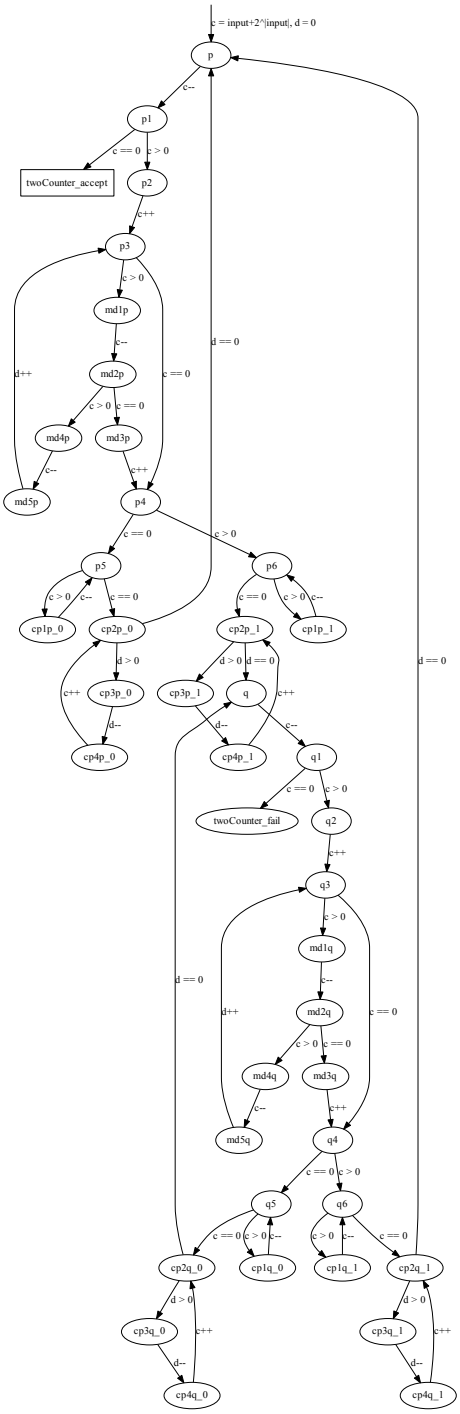
The full encoding of the automaton in Figure 4.3 contains 57 combinators and is implemented in `TwoCounterAutomaton.fs`. One example query for the input number 3 (or $(11)_2$ in binary, having an even number of ones) is

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \mathtt{c}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{s}(\mathtt{zero}))))))) \to \mathtt{d}(\mathtt{zero}) \to \mathtt{p}$$

In the above query the counter c is initialized[5] to $3 + 2^{|\text{`11`}|} = 7$ and d to 0. It is positively answered by (CL)S-F# in 0.1 seconds resulting in a combinatory term of size (number of nodes in the syntax tree) 59 corresponding to the particular accepting run of the automaton.

[5] The encoding includes an additional leading one in the binary representation.

Figure 4.3: Two Counter Automaton Checking Binary Parity



The following Table 4.1 contains detailed information on queries that simulate runs of the described automaton with increasing input size in (CL)S-F#. The column "input" contains binary representations of the input number, the column "goal type size" contains the number of nodes in the syntax tree of the corresponding goal type, the column "inhabitant size" contains the number of nodes in the syntax tree of the corresponding inhabitant (length of the corresponding run of the automaton), and the column "time in seconds" contains the time required to find an inhabitant. The last row of Table 4.1 contains an input which the automaton does not accept. Accordingly, (CL)S-F# signals that there is no inhabitant for the given goal type.

Table 4.1: Two Counter Automaton Simulation

input	goal type size	inhabitant size	time in seconds
$(11)_2$	14	59	0.1
$(1001)_2$	32	217	0.2
$(10111)_2$	68	506	0.4
$(101011)_2$	124	961	0.9
$(1010011)_2$	236	1864	2.8
$(10100011)_2$	460	3663	11.7
$(100100011)_2$	912	7286	62.5
$(100000011)_2$	904	–	53.8

Table 4.1 shows that (CL)S-F# scales well (polynomially in the size of the unary encoded input), successfully finding inhabitants that are beyond human ability to compose by hand. This example (similarly to Section 4.3.1) requires all evaluation techniques described in Section 4.2. Since natural numbers are unary encoded, (CL)S-F# needs to find type variable substitutions with codomains containing types with hundreds of nodes in their syntax trees. Additionally, due to the locality of the transition function, partial evaluation (Section 4.2.3) is useful.

The described scenario exhibits similar properties to *stateful factory planning* [51], in which labeled transition systems describe processes associated with factory planning. A comprehensive evaluation of combinatory logic synthesis in this scenario will be part of a future PhD thesis by Jan Winkels. During implementation of real-world scenarios the ability of (CL)S-F# to inspect representations of automata-like structures helped to discover problems that were previously overlooked[6] in [51].

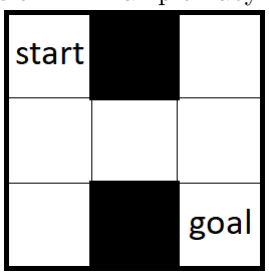
[6] Communicated by Jan Winkels to the author in 2019.

4.3.5 Labyrinth Exploration

Complementary to deterministic automaton simulation in Section 4.3.4, this section contains a scalability evaluation of non-deterministic state exploration simulated by inhabitant search in (CL)S-F#. Specifically, in this section we evaluate queries corresponding to path finding problems in a labyrinth.

At any position in a grid-like bird-view labyrinth a choice is made whether to go up, down, left, or right (if the corresponding path is available). Starting by example, consider the following 3×3 labyrinth (Figure 4.4) with a top-left starting position and bottom-right goal position.

Figure 4.4: Example Labyrinth



The shortest path from start to goal is "down, right, right, down". This scenario is represented as an instance of the inhabitation problem as follows.

Let $\texttt{zero} \in \mathbb{C}_0$, $\texttt{s} \in \mathbb{C}_1$, $\texttt{pos}, \texttt{free} \in \mathbb{C}_2$, and

$$
\begin{aligned}
\Gamma = \{ & \mathit{Left} : && \texttt{pos}(\alpha, \texttt{s}(\beta)) \to \texttt{free}(\alpha, \beta) \to \texttt{pos}(\alpha, \beta), \\
& \mathit{Right} : && \texttt{pos}(\alpha, \beta) \to \texttt{free}(\alpha, \texttt{s}(\beta)) \to \texttt{pos}(\alpha, \texttt{s}(\beta)), \\
& \mathit{Up} : && \texttt{pos}(\texttt{s}(\alpha), \beta) \to \texttt{free}(\alpha, \beta) \to \texttt{pos}(\alpha, \beta), \\
& \mathit{Down} : && \texttt{pos}(\alpha, \beta) \to \texttt{free}(\texttt{s}(\alpha), \beta) \to \texttt{pos}(\texttt{s}(\alpha), \beta), \\
& \mathit{Start} : && \texttt{pos}(\texttt{zero}, \texttt{zero}), \\
& \mathit{isFree}_{(0,0)} : && \texttt{free}(\texttt{zero}, \texttt{zero}), \\
& \mathit{isFree}_{(0,2)} : && \texttt{free}(\texttt{zero}, \texttt{s}(\texttt{s}(\texttt{zero}))), \\
& \mathit{isFree}_{(1,0)} : && \texttt{free}(\texttt{s}(\texttt{zero}), \texttt{zero}), \\
& \mathit{isFree}_{(1,1)} : && \texttt{free}(\texttt{s}(\texttt{zero}), \texttt{s}(\texttt{zero})), \\
& \mathit{isFree}_{(1,2)} : && \texttt{free}(\texttt{s}(\texttt{zero}), \texttt{s}(\texttt{s}(\texttt{zero}))), \\
& \mathit{isFree}_{(2,0)} : && \texttt{free}(\texttt{s}(\texttt{s}(\texttt{zero})), \texttt{zero}), \\
& \mathit{isFree}_{(2,2)} : && \texttt{free}(\texttt{s}(\texttt{s}(\texttt{zero})), \texttt{s}(\texttt{s}(\texttt{zero})))\}
\end{aligned}
$$

Intuitively, $\texttt{pos}(x, y)$ represents the current position at coordinates (x, y), and $\texttt{free}(x, y)$ signals that position at coordinates (x, y) is accessible. Therefore, combinators *Left*, *Right*, *Up*, *Down* provide means to transition to an accessible neighboring position. The combinator *Start* provides a starting position, and combinators $\mathit{isFree}_{(x,y)}$ signal that position (x, y) is accessible.

The query

$$\Gamma \vdash_{\mathcal{C}(\mathbb{C})} ? : \texttt{pos}(\texttt{s}(\texttt{s}(\texttt{zero})), \texttt{s}(\texttt{s}(\texttt{zero})))$$

is answered positively by (CL)S-F# in 0.2 seconds resulting in the inhabitant

$$\mathit{Down}\ (\mathit{Right}\ (\mathit{Right}\ (\mathit{Down}\ \mathit{Start}\ \mathit{isFree}_{(1,0)})\ \mathit{isFree}_{(1,1)})\ \mathit{isFree}_{(1,2)})\ \mathit{isFree}_{(2,2)}$$

where both the directions "down, right, right, down" taken as well as visited positions "$(1,0), (1,1), (1,2), (2,2)$" are exposed.

In the remainder of this section we systematically increase problem size, generating labyrinths in `LabyrinthExploration.fs`, and solving the corresponding path finding problem (top left to bottom right) using inhabitant search. The evaluation results are collected in the following Table 4.2, where the column "environment size" (resp. "inhabitant size") contains the size of the corresponding type environment (resp. constructed inhabitant), and the column "time in seconds" contains the time required to find an inhabitant. The row 24×24 describes a labyrinth that has no solution.

Table 4.2: Labyrinth Exploration

labyrinth size	environment size	inhabitant size	time in seconds
3×3	12	9	0.2
8×8	54	29	0.4
12×12	115	45	0.9
16×16	199	61	2.2
20×20	305	77	5.4
24×24	442	–	0.2
28×28	608	109	27.9
32×32	784	129	51.7

The above Table 4.2 exposes three facts. First, (CL)S-F# is able to handle non-deterministic problems represented by type environments containing hundreds of combinators. Second, (CL)S-F# does not scale perfectly, showing exponential increase in running time, whereas polynomial increase could have been expected. Third, if a problem instance has no solution (row 24×24), then (CL)S-F# signals this fact using very little time. The last two points are due to the design choice of (CL)S-F# to first compute the entire solution space as a tree grammar. In particular, the space of all (even looping) solutions is computed, before the shortest solution is presented. In unsolvable cases (row 24×24) this is very fast, in other cases the full solution space exploration exponentially increases the amount of computation.

Concluding Remarks

Overall, Section 4.3 provides an overview over (CL)S-F# performance in varying scenarios, most of which were infeasible using previous combinatory logic synthesis implementations. The presented examples cover many modeling features, including general component specification, record types, modal types, taxonomically structured domain knowledge, and natural number arithmetic. Additionally, (CL)S-F# is shown to scale adequately in automata-like scenarios, reaching inhabitants containing thousands of combinators.

Chapter 5

Conclusion

This work puts selected type-theoretic results in context of type-based program synthesis. We considered typed λ-calculi for program synthesis from scratch and typed combinatory logic for domain-specific program synthesis. Types (specifically, simple types and intersection types) assume the role of program specifications while constructive inhabitant search corresponds to program synthesis. Additionally, we inspected two classes of restrictions for inhabitant search. Restrictions (such as principality and relevance) for which constructed inhabitants are more closely tied to given specifications, and restrictions (such as dimension, rank, order, and arity) that make inhabitant search more tractable.

Considering λ-terms as functional programs, we have seen that principal inhabitation in simple types is of equal complexity (PSPACE) as its non-principal counterpart. When using intersection types as specification language, we observed that the rank restriction does not provide a suitable bounding mechanism, making an "infinite" jump in complexity from rank 2 to rank 3. Orthogonally, the dimensional restriction appears to be of practical relevance, capturing vector spaces of program features. Unfortunately, the set-dimensional restriction does not lead to a decidable inhabitation question, and complexity of inhabitation in fixed dimensions remains unknown.

Considering combinatory terms as programs (resp. metaprograms), relativized inhabitation in combinatory logic with intersection types is suited for domain-specific program synthesis. In particular, bases can be tailored to specific domains of interest, containing domain-specific components from which programs are synthesized. Although relativized inhabitation in combinatory logic is undecidable even in simple types (as we have seen, even in subintuitionistic scenarios), order and arity restrictions provide a way to gradually scale the complexity of inhabitant search.

By empiric evaluation of the (CL)S-F# inhabitant search algorithm, we have seen that program synthesis based on combinatory logic with intersection types with constructors can tractably capture functional program synthesis, object-oriented program synthesis and process synthesis scenarios. For a better grasp of underlying information passing methods (in the sense of logic program evaluation), further inspection of the intersection type unification problem, for which decidability is unknown, is of interest.

Bibliography

[1] H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, 2nd Edition. Elsevier Science Publishers, 1984.

[2] H. P. Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

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

[4] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[5] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3&4):255–299, 1991.

[6] J. Bessai. The Combinatory Logic Synthesizer (CL)S Framework in Scala. `https://github.com/combinators/cls-scala`. Accessed: 2019-02-25.

[7] 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.

[8] J. Bessai, B. Düdder, A. Dudenhefer, and M. Martens. Delegation-based Mixin Composition Synthesis. In *ITRS 2014, Intersection Types and Related Systems, Proceedings*, LNCS. Springer, 2014.

[9] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Logic Synthesizer. In *ISoLA 2014, Leveraging Applications of Formal Methods, Verification and Validation, Proceedings*, pages 26–40, 2014.

[10] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory Process Synthesis. In *ISoLA 2016, Leveraging Applications of Formal Methods, Verification and Validation, Proceedings*, pages 266–281, 2016.

[11] J. Bessai, J. Rehof, and B. Düdder. Fast Verified BCD Subtyping. 2018.

[12] G. V. Bokov. Undecidable Problems for Propositional Calculi with Implication. *Logic Journal of the IGPL*, 24(5):792–806, 2016.

[13] S. Broda and L. Damas. Counting a Type's Principal Inhabitants. In *TLCA 1999, Typed Lambda Calculi and Applications, Proceedings*, pages 69–82, 1999.

[14] S. Broda and L. Damas. Counting a Type's (Principal) Inhabitants. *Fundamenta Informaticae*, 45(1-2):33–51, 2001.

[15] S. Broda and L. Damas. On Long Normal Inhabitants of a Type. *Journal of Logic and Computation*, 15(3):353–390, 2005.

[16] A. Bucciarelli, D. Kesner, and S. Ronchi Della Rocca. The Inhabitation Problem for Non-idempotent Intersection Types. In *TCS 2014, Theoretical Computer Science, Proceedings*, pages 341–354, 2014.

[17] A. Bucciarelli, D. Kesner, and S. Ronchi Della Rocca. Inhabitation for Non-idempotent Intersection Types. *Logical Methods in Computer Science*, 14(3), 2018.

[18] A. Bucciarelli, D. Kesner, and D. Ventura. Non-idempotent Intersection Types for the lambda-Calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.

[19] S. Carlier and J. B. Wells. Expansion: the Crucial Mechanism for Type Inference with Intersection Types: A Survey and Explanation. *Electronic Notes in Theoretical Computer Science*, 136:173–202, 2005.

[20] W. Charatonik and L. Pacholski. Set Constraints with Projections are in NExpTime. In *35th Annual Symposium on Foundations of Computer Science*, pages 642–653. IEEE, 1994.

[21] B. S. Chlebus. Domino-tiling Games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[22] H. Comon. Tree Automata Techniques and Applications. `http://www.grappa.univ-lille3.fr/tata/`, 1997.

[23] M. Coppo and M. Dezani-Ciancaglini. An Extension of the Basic Functionality Theory for the λ-Calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[24] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. *Principal Type Schemes and lambda-Calculus Semantics*, pages 480–490. Accademic Press, London, 1980.

[25] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981.

[26] D. de Carvalho. Execution Time of λ-Terms via Denotational Semantics and Intersection Types. *Mathematical Structures in Computer Science*, 28(7):1169–1203, 2018.

[27] M. Dezani-Ciancaglini and J. R. Hindley. Intersection Types for Combinatory Logic. *Theoretical Computer Science*, 100(2):303–324, 1992.

[28] B. Düdder, M. Martens, and J. Rehof. Intersection Type Matching with Subtyping. In *TLCA 2013, Typed Lambda Calculi and Applications, Proceedings*, pages 125–139, 2013.

[29] B. Düdder, M. Martens, and J. Rehof. Staged Composition Synthesis. In *ESOP 2014, European Symposium on Programming, Proceedings*, pages 67–86, 2014.

[30] B. Düdder, M. Martens, J. Rehof, and P. Urzyczyn. Bounded Combinatory Logic. In *CSL 2012, Computer Science Logic, Proceedings*, pages 243–258, 2012.

[31] A. Dudenhefner. Reduction from Simple semi-Thue System Rewriting to Inhabitation in the Coppo-Dezani Type Assignment System. `https://github.com/mrhaandi/lambda-cap/tree/cdv`. Accessed: 2019-02-20.

[32] A. Dudenhefner. The Combinatory Logic Synthesizer (CL)S Framework in F#. `https://github.com/mrhaandi/cls-fsharp`. Accessed: 2019-03-20.

[33] A. Dudenhefner, M. Martens, and J. Rehof. The Algebraic Intersection Type Unification Problem. *Logical Methods in Computer Science*, 13(3), 2017.

[34] A. Dudenhefner and J. Rehof. Intersection Type Calculi of Bounded Dimension. In *POPL 2017, Principles of Programming Languages, Proceedings*, pages 653–665, 2017.

[35] A. Dudenhefner and J. Rehof. Lower End of the Linial-Post Spectrum. In *TYPES 2017, Types for Proofs and Programs, Proceedings*, pages 2:1–2:15, 2017.

[36] A. Dudenhefner and J. Rehof. Rank 3 Inhabitation of Intersection Types Revisited (Extended Version). *CoRR*, abs/1705.06070, 2017.

[37] A. Dudenhefner and J. Rehof. The Complexity of Principal Inhabitation. In *FSCD 2017, Formal Structures for Computation and Deduction, Proceedings*, pages 15:1–15:14, 2017.

[38] A. Dudenhefner and J. Rehof. Typability in Bounded Dimension. In *LICS 2017, Logic in Computer Science, Proceedings*, pages 1–12, 2017.

[39] A. Dudenhefner and J. Rehof. Principality and Approximation under Dimensional Bound. *PACMPL*, 3(POPL):8:1–8:29, 2019.

[40] A. Dudenhefner and J. Rehof. Undecidability of Intersection Type Inhabitation at Rank 3 and its Formalization. *Fundamenta Informaticae, accepted, to appear*, 2019.

[41] Y. Forster, E. Heiter, and G. Smolka. Verification of PCP-Related Computational Reductions in Coq. In *ITP 2018, Interactive Theorem Proving, Proceedings*, pages 253–269, 2018.

[42] S. Ghilezan. Strong Normalization and Typability with Intersection Types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

[43] G. T. Heineman, J. Bessai, B. Düdder, and J. Rehof. A Long and Winding Road Towards Modular Synthesis. In *ISoLA 2016, Leveraging Applications of Formal Methods, Verification and Validation, Proceedings*, pages 303–317, 2016.

[44] J. R. Hindley. The Simple Semantics for Coppo-Dezani-Sallé Types. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 212–226. Springer, 1982.

[45] J. R. Hindley. *Basic Simple Type Theory.* Vol. 42, Cambridge University Press, 2008.

[46] N. D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[47] D. Kesner and D. Ventura. Quantitative Types for the Linear Substitution Calculus. In *TCS 2014, Theoretical Computer Science, Proceedings*, pages 296–310, 2014.

[48] T. Kurata and M. Takahashi. Decidable Properties of Intersection Type Systems. In *TLCA 1995, Typed Lambda Calculi and Applications, Proceedings*, volume 902 of *LNCS*, pages 297–311. Springer, 1995.

[49] D. Leivant. Polymorphic Type Inference. In *POPL 1983, Principles of Programming Languages, Proceedings*, pages 88–98, 1983.

[50] S. Linial and E. L. Post. Recursive Unsolvability of the Deducibility, Tarski's Completeness and Independence of Axioms Problems of Propositional Calculus. *Bulletin of the American Mathematical Society*, 55:50, 1949.

[51] J. C. Nocker. *Zustandsbasierte Fabrikplanung.* PhD thesis, RWTH Aachen, Apprimus Verlag, Steinbachstr. 25, 52074 Aachen, 2 2012.

[52] F. Pfenning. Refinement Types for Logical Frameworks. In *Types for Proofs and Programs, Proceedings*, pages 285–299, 1993.

[53] E. L. Post. A Variant of a Recursively Unsolvable Problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[54] J. Rehof. Towards Combinatory Logic Synthesis. In *BEAT 2013, Behavioural Types, Proceedings.* ACM, 2013.

[55] J. Rehof and P. Urzyczyn. Finite Combinatory Logic with Intersection Types. In *TLCA 2011, Typed Lambda Calculi and Applications, Proceedings*, volume 6690 of *LNCS*, pages 169–183. Springer, 2011.

[56] J. Rehof and P. Urzyczyn. The Complexity of Inhabitation with Explicit Intersection. In *Logic and Program Semantics - Essays Dedicated to Dexter Kozen on the Occasion of His 60th Birthday*, pages 256–270, 2012.

[57] S. Schmitz. Implicational Relevance Logic is 2-ExpTime-Complete. *Journal of Symbolic Logic*, 81(2):641–661, 2016.

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

[59] H. Schwichtenberg. Definierbare Funktionen im lambda-Kalkül mit Typen. *Archiv für mathematische Logik und Grundlagenforschung*, pages 113–114, 1976.

[60] W. E. Singletary. Many-one Degrees Associated with Partial Propositional Calculi. *Notre Dame Journal of Formal Logic*, XV(2):335–343, 1974.

[61] M. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

[62] R. Statman. Intuitionistic Propositional Logic is Polynomial-space Complete. *Theoretical Computer Science*, 9:67–72, 1979.

[63] R. Statman. A Finite Model Property for Intersection Types. In *ITRS 2014, Intersection Types and Related Systems, Proceedings*, LNCS, pages 1–9, 2015.

[64] J. Tiuryn and P. Urzyczyn. The Subtyping Problem for Second-Order Types is Undecidable. *Information and Computation*, 179(1):1–18, 2002.

[65] P. Urzyczyn. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *TLCA 1997, Typed Lambda Calculi and Applications, Proceedings*, volume 1210 of *LNCS*, pages 373–389. Springer, 1997.

[66] P. Urzyczyn. Inhabitation of Low-Rank Intersection Types. In *TLCA 2009, Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 356–370. Springer, 2009.

[67] S. van Bakel. Complete Restrictions of the Intersection Type Discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

[68] S. van Bakel. Strict Intersection Types for the Lambda Calculus. *ACM Computing Surveys*, 43(3), 2011.

[69] B. Venneri. Intersection Types as Logical Formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.

[70] E. Zolin. Undecidability of the Problem of Recognizing Axiomatizations of Superintuitionistic Propositional Calculi. *Studia Logica*, 102(5):1021–1039, 2014.