technische universität
dortmund

department of
computer science

TECHNICAL REPORTS IN COMPUTER SCIENCE

Technical University of Dortmund

Intersection Type Matching and Bounded Combinatory Logic (Extended Version)

Boris Düdder
Technical University of Dortmund
Department of Computer Science
boris.duedder@cs.tu-dortmund.de

Moritz Martens
Technical University of Dortmund
Department of Computer Science
moritz.martens@cs.tu-dortmund.de

Jakob Rehof
Technical University of Dortmund
Department of Computer Science
jakob.rehof@cs.tu-dortmund.de

Number: 841
October 2012

Boris Düdder, Moritz Martens, and Jakob Rehof: *Intersection Type Matching and Bounded Combinatory Logic (Extended Version),* Technical Report, Department of Computer Science, Technical University of Dortmund. © October 2012

ABSTRACT

Bounded combinatory logic with intersection types has recently been proposed as a foundation for composition synthesis from software repositories. In such a framework, the algorithmic core in synthesis consists of a type inhabitation algorithm. Since the inhabitation problem is exponential, engineering the theoretical inhabitation algorithm with optimizations is essential. In this paper we derive several such optimizations from first principles and show how the theoretical algorithm is stepwise transformed accordingly.

Our optimizations require solving the intersection type matching problem, which is of independent interest in the algorithmic theory of intersection types: given types τ and σ, where σ is a constant type, does there exist a type substitution S such that $S(\tau)$ is a subtype of σ? We show that the matching problem is NP-complete. Membership in NP turns out to be challenging, and we provide an optimized algorithm.

This technical report is an extended version of a paper of the same title. It contains more detailed discussions and in particular the technical proofs of the various results.

CONTENTS

1 INTRODUCTION

Bounded combinatory logic with intersection types (BCL_k for short) has been proposed in [15, 6, 5, 8] as a foundation for type-based composition synthesis from repositories of software components. The idea is that a type environment Γ containing typed term variables (combinators) represents a repository of named components whose standard types (interfaces) are enriched with intersection types [3] to express semantic information. Composition synthesis can be achieved by solving the *relativized inhabitation* (provability) problem:

Given Γ *and a type* τ*, is there an applicative term* e *such that* $\Gamma \vdash_k e : \tau$*?*

Here, τ expresses the synthesis goal, and the set of inhabitants e constitute the solution space, consisting of applicative compositions of combinators in Γ.

In contrast to standard combinatory logic [11, 4], in BCL_k we bound the depth k of types used to instantiate types of combinators (k-bounded polymorphism, [6]). But rather than considering a *fixed* base of combinators (for example, the base $\mathbf{S}, \mathbf{K}$), as is usual in combinatory logic, we consider the *relativized* inhabitation problem where the set Γ of typed combinators is not held fixed but given in the input. This is the natural problem to consider, when Γ models a changing repository.

The relativized (unbounded) inhabitation problem is harder than the fixed-base problem, being undecidable even in simple types (where the fixed-base problem is PSPACE-complete [16], see [6] for more details). But, due to the expressive power of intersection types, also the standard fixed-base problem with intersection types is undecidable [18]. All problems become decidable when we impose a bound k, but the bounded systems retain enormous expressive power. The relativized inhabitation problem with intersection types, where types are restricted to be monomorphic, is EXPTIME-complete [15], and the generalization to BCL_k (k-bounded polymorphism) was shown to be $(k+2)$-EXPTIME-complete [6] by encoding space bounded alternating Turing machines (ATMs) [2].

In this paper we are concerned with principles for engineering and optimizing the theoretical inhabitation algorithm for BCL_k of [6]. This algorithm is "theoretical" in so far as it matches the $(k+2)$-EXPTIME lower bounds but does so (as is indeed desirable in theoretical work) in the simplest possible way, disregarding pragmatic concerns of efficiency. Since then we have applied inhabitation in BCL_k to software synthesis problems [5, 8] and have begun to engineer the algorithm. Here, we derive optimizations from first principles in the theory of bounded combinatory logic and apply them by a stepwise transformation of the inhabitation algorithm. It turns out that a major principle for optimization can be derived from solving the *intersection type matching* problem:

Given types τ *and* σ *where* σ *does not contain any type variables, is there a type substitution* S *with* $S(\tau) \leq_{\mathbf{A}} \sigma$*?*

The relation $\leq_{\mathbf{A}}$ denotes the standard intersection type theory of subtyping [3]. Perhaps surprisingly, the algorithmic properties of this relation, clearly of independent importance in type theory, do not appear to have been very much investigated. The (more general) problem of subtype satisfiability has been studied in various other theories, including simple types (see, e.g., [17, 14, 9, 12]), but the presence of intersection changes the problem fundamentally, and, to the best of our knowledge, there are no tight results regarding the matching or satisfiability problem for the relation $\leq_{\mathbf{A}}$. In [15] it was shown that $\leq_{\mathbf{A}}$ itself is decidable in PTIME (decidability follows from the results of [10], but with an exponential time algorithm, see [15]). Here we show that the matching problem is NP-complete and provide an algorithm that is engineered for efficiency (interestingly, the NP-upper bound appears to be somewhat challenging).

This technical report accompanies the paper of the same title. It contains the technical details and proofs.

2 PRELIMINARIES

2.1 INTERSECTION TYPES

Type expressions, ranged over by τ, σ, etc., are defined by $\tau ::= a \mid \tau \to \tau \mid \tau \cap \tau$ where $a, b, c, \ldots$ range over *atoms* comprising of *type constants*, drawn from a finite set including the constant ω, and *type variables*, drawn from a disjoint denumerable set $\mathbb{V}$ ranged over by α, β etc. We let $\mathbb{T}$ denote the set of all types.

As usual, types are taken modulo commutativity ($\tau \cap \sigma = \sigma \cap \tau$), associativity ($(\tau \cap \sigma) \cap \rho = \tau \cap (\sigma \cap \rho)$), and idempotency ($\tau \cap \tau = \tau$). As a matter of notational convention, function types associate to the right, and $\cap$ binds stronger than $\to$. A type *environment* Γ is a finite function from term variables to types, written as a finite set of type assumptions of the form $(x : \tau)$. We let $Var(\tau)$ and $Var(\Gamma)$ denote the sets of type variables occurring in a type τ respectively in Γ.

A type $\tau \cap \sigma$ is said to have τ and σ as *components*. For an intersection of several components we sometimes write $\bigcap_{i=1}^{n} \tau_i$ or $\bigcap_{i \in I} \tau_i$ or $\bigcap\{\tau_i \mid i \in I\}$, where the empty intersection is identified with ω.

The standard [3] intersection type *subtyping* relation $\leq_{\mathbf{A}}$ is the least preorder (reflexive and transitive relation) on $\mathbb{T}$ generated by the following set $\mathbf{A}$ of axioms:

$$\sigma \leq_{\mathbf{A}} \omega, \quad \omega \leq_{\mathbf{A}} \omega \to \omega, \quad \sigma \cap \tau \leq_{\mathbf{A}} \sigma, \quad \sigma \cap \tau \leq_{\mathbf{A}} \tau, \quad \sigma \leq_{\mathbf{A}} \sigma \cap \sigma;$$
$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq_{\mathbf{A}} \sigma \to \tau \cap \rho;$$
$$\text{If } \sigma \leq_{\mathbf{A}} \sigma' \text{ and } \tau \leq_{\mathbf{A}} \tau' \text{ then } \sigma \cap \tau \leq_{\mathbf{A}} \sigma' \cap \tau' \text{ and } \sigma' \to \tau \leq_{\mathbf{A}} \sigma \to \tau'.$$

We identify σ and τ when $\sigma \leq_{\mathbf{A}} \tau$ and $\tau \leq_{\mathbf{A}} \sigma$. The distributivity properties $(\sigma \to \tau) \cap (\sigma \to \rho) = \sigma \to (\tau \cap \rho)$ and $(\sigma \to \tau) \cap (\sigma' \to \tau') \leq_{\mathbf{A}} (\sigma \cap \sigma') \to (\tau \cap \tau')$ follow from the axioms of subtyping. Note also that $\tau_1 \to \cdots \to \tau_m \to \omega = \omega$. We say that a type τ is *reduced with respect to* ω if it has no subterm of the form $\rho \cap \omega$ or $\tau_1 \to \cdots \to \tau_m \to \omega$ with $m \geq 1$. It is easy to reduce a type with respect to ω, by applying the equations $\rho \cap \omega = \rho$ and $\tau_1 \to \cdots \to \tau_m \to \omega = \omega$ left to right.

If $\tau = \tau_1 \to \cdots \to \tau_m \to \sigma$, then we write $\sigma = tgt_m(\tau)$ and $\tau_i = arg_i(\tau)$, for $i \leq m$. A type of the form $\tau_1 \to \cdots \to \tau_m \to a$, where $a \neq \omega$ is an atom, is called a *path of length* m.

A type τ is *organized* if it is a (possibly empty) intersection of paths (those are called *paths in* τ). Every type τ is equal to an organized type $\overline{\tau}$, computable in polynomial time, with $\overline{a} = a$, if a is an atom, and with $\overline{\tau \cap \sigma} = \overline{\tau} \cap \overline{\sigma}$. Finally, if $\overline{\sigma} = \bigcap_{i \in I} \sigma_i$, then take $\overline{\tau \to \sigma} = \bigcap_{i \in I} (\tau \to \sigma_i)$. Note that premises in an organized type do not have to be organized, i.e., organized types are not necessarily *normalized* as defined in [10] (in contrast to organized types, the normalized form of a type may be exponentially large in the size of the type).

For an organized type σ, we let $\mathbb{P}_m(\sigma)$ denote the set of all paths in σ of length m or more. We extend the definition to arbitrary τ by implicitly organizing τ, i.e., we write $\mathbb{P}_m(\tau)$ as a shorthand for $\mathbb{P}_m(\overline{\tau})$. The *path length* of a type τ is denoted $\|\tau\|$ and is defined to be the maximal length of a path in $\overline{\tau}$.

A *substitution* is a function $S : \mathbb{V} \to \mathbb{T}$, such that S is the identity everywhere but on a finite subset of $\mathbb{V}$. For a substitution S, we define the *support* of S, written $Supp(S)$, as $Supp(S) = \{\alpha \in \mathbb{V} \mid \alpha \neq S(\alpha)\}$. We may write $S : V \to \mathbb{T}$ when V is a finite subset of $\mathbb{V}$ with $Supp(S) \subseteq V$. A substitution S is tacitly lifted to a function on types, $S : \mathbb{T} \to \mathbb{T}$, by homomorphic extension.

The following property, probably first stated in [1], is often called *beta-soundness*. Note that the converse is trivially true.

Lemma 1 *Let a and a_j, for $j \in J$, be atoms.*

1. *If $\bigcap_{i\in I}(\sigma_i \to \tau_i) \cap \bigcap_{j\in J} a_j \leq_{\mathbf{A}} a$ then $a = a_j$, for some $j \in J$.*
2. *If $\bigcap_{i\in I}(\sigma_i \to \tau_i) \cap \bigcap_{j\in J} a_j \leq_{\mathbf{A}} \sigma \to \tau$, where $\sigma \to \tau \neq \omega$, then the set $\{i \in I \mid \sigma \leq_{\mathbf{A}} \sigma_i\}$ is nonempty and $\bigcap\{\tau_i \mid \sigma \leq_{\mathbf{A}} \sigma_i\} \leq_{\mathbf{A}} \tau$.*

2.2 ALTERNATING TURING MACHINES

An *alternating Turing machine* (ATM) [2] is a tuple $\mathcal{M} = (\Sigma, Q, q_0, q_a, q_r, \Delta)$. The set of states $Q = Q_\exists \uplus Q_\forall$ is partitioned into a set $Q_\exists$ of existential states and a set $Q_\forall$ of universal states. There is an initial state $q_0 \in Q$, an accepting state $q_a \in Q_\forall$, and a rejecting state $q_r \in Q_\exists$. We take $\Sigma = \{0, 1, \sqcup\}$, where $\sqcup$ is the blank symbol (used to initialize the tape but not written by the machine). The transition relation Δ satisfies

$$\Delta \subseteq \Sigma \times Q \times \Sigma \times Q \times \{\text{L}, \text{R}\},$$

where $h \in \{\text{L}, \text{R}\}$ are the moves of the machine head (left and right). For $b \in \Sigma$ and $q \in Q$, we write $\Delta(b, q) = \{(c, p, h) \mid (b, q, c, p, h) \in \Delta\}$. We assume $\Delta(b, q_a) = \Delta(b, q_r) = \emptyset$, for all $b \in \Sigma$, and $\Delta(b, q) \neq \emptyset$ for $q \in Q \setminus \{q_a, q_r\}$. A *configuration* of $\mathcal{M}$ is a word wqw' with $q \in Q$ and $w, w' \in \Sigma^*$. The *successor* relation $\mathcal{C} \Rightarrow \mathcal{C}'$ on configurations is defined as usual [13], according to Δ. We classify a configuration wqw' as *existential*, *universal*, *accepting* etc., according to q. A configuration wqw' is

- *halting*, if and only if $q \in \{q_a, q_r\}$.
- *accepting* if and only if $q = q_a$.
- *rejecting* if and only if $q = q_r$.

The notion of *eventually accepting* configuration is defined by induction:[1]

1 Formally we define the set of all eventually accepting configurations as the smallest set satisfying the appropriate closure conditions.

- An accepting configuration is eventually accepting.
- If $\mathcal{C}$ is existential and some successor of $\mathcal{C}$ is eventually accepting, then so is $\mathcal{C}$.
- If $\mathcal{C}$ is universal and all successors of $\mathcal{C}$ are eventually accepting, then so is $\mathcal{C}$.

We introduce the following notation for existential and universal states. A command of the form CHOOSE $r \in R$ branches from an existential state to successor states in which r gets assigned distinct elements of R (it implicitly rejects if $R = \emptyset$). A command of the form FORALL$(i = 1 \ldots k)\, R_i$ branches from a universal state to successor states from which each instruction sequence R_i is executed.

3 BOUNDED COMBINATORY LOGIC

We briefly present (Sect. 3.1) the systems of k-bounded combinatory logic with intersection types, denoted $\text{BCL}_k(\to,\cap)$, referring the reader to [6] for a full introduction. We then describe (Sect. 3.2) our first optimization to the theoretical algorithm of [6]. The optimized algorithm is close enough to the theoretical algorithm of [6] that we can use it to explain the latter also.

3.1 TYPE SYSTEM

For each $k \geq 0$ the system $\text{BCL}_k(\to,\cap)$ (or, BCL_k for short) is defined by the type assignment rules shown in Fig. 3.1, assigning types to applicative (combinatory) terms $e ::= x \mid (e\ e)$, where x ranges over term variables (combinators). We assume that application associates to the left and we omit outermost parentheses. Notice that any applicative term can be written uniquely as $x\ e_1 \dots e_n$. We freely use the following notation to denote the previous term, which represents x as an n-ary function: $x(e_1,\dots,e_n)$. In rule (var), the condition $\ell(S) \leq k$ is understood as a side condition to the axiom $\Gamma, x : \tau \vdash_k x : S(\tau)$. Here, the *level* of a substitution S, denoted $\ell(S)$, is defined as follows. First, for a type τ, define its level $\ell(\tau)$ by $\ell(a) = 0$ for $a \in \cup\mathbb{V}$, $\ell(\tau \to \sigma) = 1 + \max\{\ell(\tau), \ell(\sigma)\}$, and $\ell(\bigcap_{i=1}^{n} \tau_i) = \max\{\ell(\tau_i) \mid i = 1,\dots,n\}$. Now define $\ell(S) = \max\{\ell(S(\alpha)) \mid \alpha \in \mathbb{V}\}$. Notice that the level of a type is independent of the number of components in an intersection.

$$\frac{[\ell(S) \leq k]}{\Gamma, x : \tau \vdash_k x : S(\tau)}(\text{var}) \qquad \frac{\Gamma \vdash_k e : \tau \to \tau' \quad \Gamma \vdash_k e' : \tau}{\Gamma \vdash_k (e\ e') : \tau'}(\to\text{E})$$

$$\frac{\Gamma \vdash_k e : \tau_1 \quad \Gamma \vdash_k e : \tau_2}{\Gamma \vdash_k e : \tau_1 \cap \tau_2}(\cap\text{I}) \qquad \frac{\Gamma \vdash_k e : \tau \quad \tau \leq_{\mathbf{A}} \tau'}{\Gamma \vdash_k e : \tau'}(\leq)$$

Figure 3.1: Bounded combinatory logic $\text{BCL}_k(\to,\cap)$

A *level-k type* is a type τ with $\ell(\tau) \leq k$, and a *level-k substitution* is a substitution S with $\ell(S) \leq k$. For $k \geq 0$, we let $\mathbb{T}_k$ denote the set of all level-k types. For a subset A of atomic types, we let $\mathbb{T}_k(A)$ denote the set of level-k types with atoms (leaves) in the set A.

In bounded combinatory logic [6] and its use in synthesis [5, 8] we are addressing the following *relativized inhabitation* problem:

Given Γ *and* τ*, is there an applicative term* e *such that* $\Gamma \vdash_k e : \tau$*?*

The cause of the exponentially growing complexity of inhabitation in BCL_k (compared to the monomorphic restriction [15]) lies in the need to search for suitable instantiating substitutions S in rule (var). In [6] it is shown that one needs only to consider rule (var) restricted to substitutions of the form $S : Var(\Gamma) \to \mathbb{T}_k(At_\omega(\Gamma, \tau))$, where $At_\omega(\Gamma, \tau)$ denotes the set of atoms occurring in Γ or τ, together with ω. This finitizes the inhabitation problem and immediately leads to decidability. Now, given a number k, an environment Γ and a type τ, define for each x occurring in Γ the set of substitutions $\mathcal{S}_x^{(\Gamma,\tau,k)} = Var(\Gamma(x)) \to \mathbb{T}_k(At_\omega(\Gamma, \tau))$. This set, as well as the type size, grows exponentially with k, and at the root of the $(k+2)$-EXPTIME-hardness result of [6] for inhabitation in BCL_k is the fact that one cannot bypass, in the worst case, exploring such vast spaces of types and substitutions. However, in applications [5, 8] it is to be expected that a complete, brute-force exploration of the sets $\mathcal{S}_x^{(\Gamma,\tau,k)}$ is unnecessary. This is the point of departure for our optimizations of the theoretical algorithm of [6], which, for convenience, is stated in the following. It is an (k+1)-EXPSPACE ATM, yielding a (k+2)-EXPTIME decision procedure.

The idea behind our first optimization is to show that for a type $\tau = \bigcap_{i\in I} \tau_i$ to satisfy $\tau \leq \sigma$, where $\sigma = \bigcap_{j\in J} \sigma_j$, the size of the index set I can be bounded by the size of the index set J of σ. One might at first conjecture that it always suffices to consider an index set I where $|I| \leq |J|$. This is not true as can easily seen by considering $(a \to b) \cap (a \to c) \leq_{\mathbf{A}} a \to (b \cap c)$, for example. But we show that the property holds for organized types.

Based on Lem. 1 we characterize the subtypes of a path (generalizing Lem. 3 in [6]):

Lemma 2 *Let* $\tau = \bigcap_{i\in I} \tau_i$ *where the* τ_i *are paths and let* $\sigma = \beta_1 \to \ldots \to \beta_n \to p$ *where* $p \neq \omega$ *is an atom.*

We have $\tau \leq_{\mathbf{A}} \sigma$ *if and only if there is an* $i \in I$ *with* $\tau_i = \alpha_1 \to \ldots \to \alpha_n \to p$ *and* $\beta_j \leq_{\mathbf{A}} \alpha_j$ *for all* $j \leq n$*.*

Proof: Once and for all we write $\bigcap_{i\in I} \tau_i = \bigcap_{j\in J} a_j \cap \bigcap_{k\in K} \sigma_k \to \sigma'_k$ (in particular $I = J \cup K$).

$\Rightarrow$: We use induction over n.

- $n = 0$: We have $\bigcap_{i\in I} \tau_i \leq_{\mathbf{A}} p$ where p is a type constant. Lem. 1 states that there must be a $j \in J$ with $a_j = p$.
- $n \Rightarrow n+1$: Assume $\bigcap_{i\in I} \tau_i \leq_{\mathbf{A}} \beta_1 \to \ldots \to \beta_{n+1} \to p$. Lem. 1 further states that the set $H = \{k \in K | \beta_1 \leq_{\mathbf{A}} \sigma_k\}$ is non-empty and $\bigcap_{h\in H} \sigma'_k \leq_{\mathbf{A}} \beta_2 \to \ldots \to \beta_{n+1} \to p$. Note that each of the σ'_k is a again a path. Therefore, we may apply the induction hypothesis to the last inequality and we see that there is some $h_0 \in H$ with $\sigma'_{h_0} = \alpha_2 \to \ldots \to \alpha_{n+1} \to p$ where $\beta_l \leq_{\mathbf{A}} \alpha_l$ for all $2 \leq l \leq n+1$. Because $h_0 \in H$ we

Algorithm 0 Alternating Turing machine deciding inhabitation in BCL_k

1: *Input:* Γ, τ, k
2: *Output:* INH1 accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3:
4: *loop:*
5: CHOOSE $(x : \sigma) \in \Gamma$;
6: $\sigma' := \bigcap\{S(\sigma) \mid S \in \mathcal{S}_x^{(\Gamma,\tau,k)}\}$;
7: CHOOSE $n \in \{0, \ldots \|\sigma'\|\}$;
8: CHOOSE $P \subseteq \mathbb{P}_n(\sigma')$;
9:
10: **if** $\bigcap_{\pi \in P} tgt_n(\pi) \leq_{\mathbf{A}} \tau$ **then**
11: **if** $n = 0$ **then**
12: ACCEPT;
13: **else**
14: FORALL $(j = 1 \ldots n)$ $\tau := \bigcap_{\pi \in P} arg_j(\pi)$;
15: GOTO *loop*;
16: **end if**
17: **else**
18: FAIL;
19: **end if**

know that $\beta_1 \leq_{\mathbf{A}} \sigma_{h_0}$. Setting $\alpha_1 = \sigma_{h_0}$, the type $\sigma_{h_0} \to \sigma'_{h_0} = \alpha_1 \to \ldots \to \alpha_{n+1} \to \tau$ has the desired properties.

$\Leftarrow$: For this direction we first show by induction over n that a type $\alpha_1 \to \ldots \to \alpha_n \to p$ with $\beta_j \leq_{\mathbf{A}} \alpha_j$ for all $j \leq n$ is a subtype of $\beta_1 \to \ldots \to \beta_n \to p$.

$n = 0$: There is nothing to prove because both types are equal to p.

$n \Rightarrow n + 1$: We want to show $\alpha_1 \to \ldots \to \alpha_{n+1} \to p \leq_{\mathbf{A}} \beta_1 \to \ldots \to \beta_{n+1} \to p$. Because of Lem. 1 this inequality holds if and only if $\beta_1 \leq_{\mathbf{A}} \alpha_1$ and $\alpha_2 \to \ldots \to \alpha_{n+1} \to p \leq_{\mathbf{A}} \beta_2 \to \ldots \to \beta_{n+1} \to p$. The first inequality holds by assumption the second one holds because of the induction hypothesis.

By assumption there is an $i \in I$ with $\tau_i = \alpha_1 \to \ldots \to \alpha_n \to p$ and $\beta_j \leq_{\mathbf{A}} \alpha_j$ for all $j \leq n$. From the above we know $\tau_i \leq_{\mathbf{A}} \beta_1 \to \ldots \to \beta_n \to p$ and therefore also $\bigcap_{i \in I} \tau_i \leq_{\mathbf{A}} \beta_1 \to \ldots \to \beta_n \to p$.

□

For $\sigma = \bigcap_{j \in J} \sigma_j$ (not necessarily organized), it is easy to see that one has $\tau \leq_{\mathbf{A}} \sigma$ iff for all j we have $\tau \leq_{\mathbf{A}} \sigma_j$. Using this observation together with Lem. 2 we obtain:

Lemma 3 *Let $\tau = \bigcap_{i\in I} \tau_i$ and $\sigma = \bigcap_{j\in J} \sigma_j$ be organized types that are reduced with respect to ω.*
We have $\tau \leq_{\mathbf{A}} \sigma$ iff there exists $I' \subseteq I$ with $|I'| \leq |J|$ and $\bigcap_{i\in I'} \tau_i \leq_{\mathbf{A}} \sigma$.

Proof: The right-to-left implication is obvious.

Assume $\tau \leq_{\mathbf{A}} \sigma$. This implies $\tau \leq_{\mathbf{A}} \sigma_j$ for all $j \in J$. Fix $j \in J$. σ is organized and we write $\sigma_j = \beta^j_1 \to \ldots \to \beta^j_{n_j} \to p^j$. By the "if"-part of Lem. 2 there is an index $i_j \in I$ such that $\tau_{i_j} = \alpha^{i_j}_1 \to \ldots \to \alpha^{i_j}_{n_j} \to p^j$ with $\beta^j_k \leq_{\mathbf{A}} \alpha^{i_j}_k$ for all $1 \leq k \leq n_j$. Set $I' := \{i \in I | \exists j \in J \text{ with } i = i_j\}$. Clearly $|I'| \leq |J|$ holds. For every $j \in J$ the "only if"-part of Lem. 2 shows $\bigcap_{i\in I'} \tau_i \leq_{\mathbf{A}} \sigma_j$ because τ_{i_j} satisfies the condition stated. This implies $\bigcap_{i\in I'} \tau_i \leq_{\mathbf{A}} \sigma$. □

As shown in [6], the key to an algorithm matching the lower bound for BCL_k is a *path lemma* ([6, Lemma 11]) which characterizes inhabitation by the existence of certain sets of paths in instances of types in Γ. The following lemma is a consequence of Lem. 3 and [6, Lemma 11].

Lemma 4 *Let $\tau = \bigcap_{i\in I} \tau_i$ be organized and let $x : \sigma \in \Gamma$.*
The following are equivalent conditions:

1. $\Gamma \vdash_k x\ e_1 \ldots e_m : \tau$
2. *There exists a set P of paths in $\mathbb{P}_m(\bigcap\{\overline{S(\sigma)} \mid S \in \mathcal{S}^{(\Gamma,\tau,k)}_x\})$ such that*
   a) $\bigcap_{\pi\in P} tgt_m(\pi) \leq_{\mathbf{A}} \tau$;
   b) $\Gamma \vdash_k e_i : \bigcap_{\pi\in P} arg_i(\pi)$, *for all $i \leq m$.*
3. *There exists a set $\mathcal{S} \subseteq \mathcal{S}^{(\Gamma,\tau,k)}_x$ of substitutions with $|\mathcal{S}| \leq |I|$ and a set $P' \subseteq \mathbb{P}_m(\bigcap_{S\in\mathcal{S}} \overline{S(\sigma)})$ of paths with $|P'| \leq |I|$ such that*
   a) $\bigcap_{\pi\in P'} tgt_m(\pi) \leq_{\mathbf{A}} \tau$;
   b) $\Gamma \vdash_k e_i : \bigcap_{\pi\in P'} arg_i(\pi)$, *for all $i \leq m$.*

Proof: The implication 1. $\Rightarrow$ 2. follows from Lemma 10 in [7].

We prove 2. $\Rightarrow$ 3 : Let P be as in the condition, i.e., $\bigcap_{\pi\in P} tgt_m(\pi) \leq_{\mathbf{A}} \tau$. Lemma 3 states that there is $P' \subseteq P$ with $|P'| \leq |I|$ and $\bigcap_{\pi\in P'} tgt_m(\pi) \leq_{\mathbf{A}} \tau$. For each $\pi \in P'$ there exists $S_\pi \in \mathcal{S}^{(\Gamma,\tau,k)}_x$ such that $\pi \in \mathbb{P}_m(\overline{S_\pi(\sigma)})$. Define $\mathcal{S} = \{S_\pi | \pi \in P'\}$. It is clear that $|\mathcal{S}| \leq |I|$ and that $P' \subseteq \mathbb{P}_m(\overline{\bigcap_{S\in\mathcal{S}} S(\sigma)})$. Thus, 3.(a) holds. Fix $i \leq m$. Because $P' \subseteq P$ we have $\bigcap_{\pi\in P} arg_i(\pi) \leq_{\mathbf{A}} \bigcap_{\pi\in P'} arg_i(\pi)$. Since we have $\Gamma \vdash_k e_i : \bigcap_{\pi\in P} arg_i(\pi)$, rule ($\leq_{\mathbf{A}}$) yields $\Gamma \vdash_k e_i : \bigcap_{\pi\in P'} arg_i(\pi)$. Therefore, 3.(b) also holds.

The implication 3. $\Rightarrow$ 1. follows from a suitable application of the type rules. □

We immediately get the following corollary.

Corollary 5 (Path Lemma) *Let $\tau = \bigcap_{i\in I} \tau_i$ be organized and let $(x : \sigma) \in \Gamma$.*
The following are equivalent conditions:

1. $\Gamma \vdash_k x\ e_1 \dots e_m : \tau$
2. *There exists a set* $\mathcal{S} \subseteq \mathcal{S}_x^{(\Gamma,\tau,k)}$ *of substitutions with* $|\mathcal{S}| \leq |I|$ *and a set* $P \subseteq \mathbb{P}_m(\overline{\bigcap_{S\in\mathcal{S}} S(\sigma)})$ *of paths with* $|P| \leq |I|$ *such that*
   a) $\bigcap_{\pi\in P} tgt_m(\pi) \leq_{\mathbf{A}} \tau$;
   b) $\Gamma \vdash_k e_j : \bigcap_{\pi\in P} arg_j(\pi)$, *for all* $j \leq m$.

Algorithm 1 below is a direct implementation of the path lemma (Cor. 5) and therefore decides inhabitation in BCL_k.

Algorithm 1 INH1(Γ, τ, k)

```
1:  Input: Γ, τ, k — wlog: All types in Γ and τ = ⋂_{i∈I} τ_i are organized
2:  Output: INH1 accepts iff ∃e such that Γ ⊢_k e : τ
3:
4:  loop:
5:  CHOOSE (x : σ) ∈ Γ;
6:  CHOOSE S ⊆ S_x^{(Γ,τ,k)} with |S| ≤ |I|;
7:  σ' := ⋂{S(σ) | S ∈ S};
8:  CHOOSE n ∈ {0, ... ‖σ'‖};
9:  CHOOSE P ⊆ P_n(σ') with |P| ≤ |I|;
10:
11: if ⋂_{π∈P} tgt_n(π) ≤_A τ then
12:    if n = 0 then
13:       ACCEPT;
14:    else
15:       FORALL (j = 1 ... n) τ := ⋂_{π∈P} arg_j(π);
16:       GOTO loop;
17:    end if
18: else
19:    FAIL;
20: end if
```

Algorithm 0 is identical to Alg. 1 but for the fact that it ignores the restrictions $|\mathcal{S}| \leq |I|$ (line 6 in Alg. 1) and $|P| \leq |I|$ (line 9). It can be seen, from purely combinatorial considerations, that the optimization resulting from taking these bounds into account can lead to arbitrarily large speed-ups (when $|I|$ is relatively small, as can be expected in practice).

4 INTERSECTION TYPE MATCHING

The work done in lines 5 through 9 of Alg. 1 aims at constructing paths π descending from σ (using instantiating substitutions) such that the condition $\bigcap_{\pi \in P} tgt_n(\pi) \leq_{\mathbf{A}} \tau$ in line 11 is satisfied. Clearly, it would be an important optimizing heuristic if we could rule out uninteresting paths π that do not contribute to the satisfaction of the condition. The earlier we can do this, the better. So the optimal situation would be if we could somehow do it by inspecting paths in σ very early on, i.e., right after choosing σ in line 5.

As we will show, it turns out that this can indeed be done based on a solution to the *intersection type matching* problem:

Given types τ and σ where σ does not contain any type variables, is there a substitution $S : \mathbb{V} \to \mathbb{T}$ with $S(\tau) \leq_{\mathbf{A}} \sigma$?

We shall proceed to show that this problem is NP-complete (lower and upper bound in Sect(s). 4.1 respectively 4.2). Interestingly, the upper bound is quite challenging. It is also worth emphasizing that the lower bound turns out to hold even when restricting the matching problem to atomic (level-0) substitutions.

Definition 6 *Let $C = \{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ be a set of type constraints such that for every i either σ_i or τ_i does not contain any type variables. We say that C is matchable if there is a substitution $S : \mathbb{V} \to \mathbb{T}$ such that for all i we have $S(\tau_i) \leq_{\mathbf{A}} S(\sigma_i)$. We say that S matches C.*

CMATCH *denotes the decision problem of whether a given set of constraints C is matchable.* cMATCH *denotes the decision problem of whether a given constraint $\tau \leq \sigma$ where σ does not contain any type variables is matchable.*

We sometimes denote CMATCH and cMATCH as matching problems. Furthermore, we write $S(\sigma) \leq_{\mathbf{A}} S(\tau)$ if it is not known which of the two types contains variables, and we omit parentheses if C is a singleton set. Note that we use $\leq$ to denote a formal constraint whose matchability is supposed to be checked whereas $\tau \leq_{\mathbf{A}} \sigma$ states that τ is a subtype of σ.

4.1 LOWER BOUND

We show that intersection type matching is NP-hard by defining a reduction $\mathcal{R}$ from 3SAT to CMATCH such that any formula F in 3-CNF is satisfiable iff $\mathcal{R}(F)$ is matchable. Let $F = c_1 \wedge \ldots \wedge c_m$ where for each i we have $c_i = L_i^1 \vee L_i^2 \vee L_i^3$ and each L_i^j is either a propositional variable x or a negation $\neg x$ of such a variable. For all propositional variables x, occurring in F we define two fresh type variables called α_x and $\alpha_{\neg x}$. Furthermore, we assume the two type constants 1 and 0. For a given formula F, let $\mathcal{R}(F)$ denote the set containing the following constraints:

1. For all x in F: $\big((1 \to 1) \to 1\big) \cap \big((0 \to 0) \to 0\big) \leq (\alpha_x \to \alpha_x) \to \alpha_x$
2. For all x in F: $(0 \to 1) \cap (1 \to 1) \leq \alpha_x \to 1$
3. For all x in F: $\big((1 \to 1) \to 1\big) \cap \big((0 \to 0) \to 0\big) \leq (\alpha_{\neg x} \to \alpha_{\neg x}) \to \alpha_{\neg x}$
4. For all x in F: $(0 \to 1) \cap (1 \to 1) \leq \alpha_{\neg x} \to 1$
5. For all x in F: $(1 \to 0) \cap (0 \to 1) \leq \alpha_x \to \alpha_{\neg x}$
6. For all c_i: $\alpha_{L_i^1} \cap \alpha_{L_i^2} \cap \alpha_{L_i^3} \leq 1$

It is clear that $\mathcal{R}(F)$ can be constructed in polynomial time.

Theorem 7 *F satisfiable $\Leftrightarrow$ $\mathcal{R}(F)$ matchable*

Proof: For the "only if"-direction let v be a valuation that satisfies F. We define a substitution S_v as follows:

- $S_v(\alpha_x) = v(x)$
- $S_v(\alpha_{\neg x}) = \neg v(x)$

By way of slight notational abuse the right hand sides of these defining equations represent the truth values $v(x)$ and $\neg v(x)$ as types. We claim that S_v matches $\mathcal{R}(F)$. For the first five constraints this is obvious. Consider a clause c_i in F and the corresponding constraint in the sixth group of constraints: $v(F) = 1$ implies that there is a literal L_i^j with $v(L_i^j) = 1$. Thus, $S_v(\alpha_{L_i^j}) = 1$ and the constraint corresponding to c_i is matched.

For the "if"-direction, from a substitution S matching $\mathcal{R}(F)$ we construct a satisfying valuation v_S for F. We define $v_S(x) = S(\alpha_x)$, and show that v_S is well-defined and satisfies F. Consider a type variable α_x. Using Lem. 1 it is not difficult to show that S can only match the first constraint if $S(\alpha_x) \in \{0, 1, \omega\}$. The second constraint, however, will not be matched if $S(\alpha_x) = \omega$. It is matched by the instantiations $S(\alpha_x) = 0$ and $S(\alpha_x) = 1$, though. Thus, the first two constraints make sure that $S(\alpha_x) \in \{0, 1\}$. The same argument, using the third and fourth constraint, shows $S(\alpha_{\neg x}) \in \{0, 1\}$. These two observations can be used together with the fact that the fifth constraint is matched for x to show that $S(\alpha_x) = 1$ if and only if $S(\alpha_{\neg x}) = 0$ and vice versa. We conclude that v_S is well-defined. In order to show that it satisfies F we need to show that for every clause c_i there is a literal L_i^j with $v_S(L_i^j) = 1$. Because S matches $\mathcal{R}(F)$ we have $S(\alpha_{L_i^1}) \cap S(\alpha_{L_i^2}) \cap S(\alpha_{L_i^3}) \leq_{\mathbf{A}} 1$. Lemma 1 states that the type on the left-hand side must have a component which is equal to 1. We already know that each of the three variables is instantiated either to 0 or 1. Thus, at least one of them must be instantiated to 1. Therefore, $v_S(c_i) = 1$ and v_S satisfies F. □

We immediately get the following corollary:

Corollary 8 CMATCH *is* NP*-hard.*

Exploiting co- and contravariance, a set C of constraints can be transformed into a single constraint c such that C is matchable if and only if c is matchable. This yields a reduction from CMATCH to cMATCH, hence the following corollary:

Corollary 9 cMATCH *is* NP*-hard.*

4.2 UPPER BOUND

We show that CMATCH, and thus also cMATCH, is in NP. We derive an algorithm engineered for efficiency by a case analysis that attempts to minimize nondeterminism. We first need some definitions:

Definition 10 *We call $\tau \leq \sigma$ a basic constraint if either τ is a type variable and σ does not contain any type variables or σ is a type variable and τ does not contain any type variables.*

Definition 11 *Let C be a set of basic constraints.*

Let α be a variable occurring in C. Let $\tau_i \leq \alpha$ for $1 \leq i \leq n$ be the constraints in C where α occurs on the right hand side of $\leq$ and let $\alpha \leq \sigma_j$ for $1 \leq j \leq m$ be the constraints in C where α occurs on the left hand side of $\leq$. We say that C is consistent with respect to α if for all i and j we have $\tau_i \leq_{\mathbf{A}} \sigma_j$.

C is consistent if it is consistent with respect to all variables occurring in C.

In the following we will need a lemma which formalizes the observation that a set of basic constraints is matchable if and only if it is consistent.

Lemma 12 *Let C be a set of basic constraints. C can be matched if and only if C is consistent.*

Proof: In the following let α be a variable occurring in C and let $\tau_i \leq \alpha$ for $1 \leq i \leq n$ be the constraints in C where α occurs on the right hand side of $\leq$ and let $\alpha \leq \sigma_j$ for $1 \leq j \leq m$ be the constraints in C where α occurs on the left hand side of $\leq$. We now show both directions:

$\Rightarrow$: Assume that C is matchable. We want to show that it is consistent. We have to show that for all i and j we have $\tau_i \leq_{\mathbf{A}} \sigma_j$. Because C is matchable there exists a substitution S such that $\tau_i \leq_{\mathbf{A}} S(\alpha)$ for $1 \leq i \leq n$ and $S(\alpha) \leq_{\mathbf{A}} \sigma_j$ for $1 \leq j \leq m$. By transitivity of $\leq_{\mathbf{A}}$ we get $\tau_i \leq_{\mathbf{A}} S(\alpha) \leq_{\mathbf{A}} \sigma_j$ for all i and j.

$\Leftarrow$: For the direction from right to left, let C be a set of basic constraints and let $\alpha \leq \sigma_j$ for $1 \leq j \leq m$ be the constraints in C where α occurs on the left hand side of $\leq$. Define the substitution S_C by setting $S_C(\alpha) = \alpha$, if α does not occur in C, and $S_C(\alpha) = \bigcap_{j=1}^{m} \sigma_j$ otherwise[1]. Notice that the basic constraints in C do not contain any variables on one side of $\leq$. If C is consistent, then we have $\tau_i \leq_{\mathbf{A}} \sigma_j$ for all i and j which is equivalent to $\tau_i \leq_{\mathbf{A}} \bigcap_{j=1}^{m} \sigma_j = S_C(\alpha)$ for all i. Thus, S_C matches C.

1 Recall that empty intersections equal ω (thus, S_C is well-defined even if $m = 0$).

□

Note that it is important that we can treat variables independently in the proof, because the basic constraints in C do not contain any variables on one side of $\leq$ (hence the types $\bigcap_{j=1}^{m} \sigma_j$ contain no variables). The proof technique would not work for the satisfiability problem. Algorithm 2 below is a nondeterministic procedure that decomposes the constraints in a set C to be matched until we arrive at a set of basic constraints. Using the lemma above, we know that, if and only if this set is consistent, we may conclude that C is matchable. We make the following more detailed remarks about the algorithm:

Remark 13

1. *Memoization is used to make sure that no constraint is added to C more than once.*

2. *Failing choices always return **false**.*

3. *The reduction with respect to ω in line 6 means, in particular, that, unless they are syntactically identical to ω, neither τ nor σ contain any types of the form $\rho_1 \to \ldots \to \rho_m \to \omega$ as top-level components.*

4. *Line 14 subsumes the case $\sigma = \omega$ if $I = \emptyset$. Then c is simply removed from C, and no new constraints are added.*

5. *We assume that the cases in the **switch**-block are mutually exclusive and checked in the given order. Thus, we know, for example, that for the two cases in lines 19 and 30 σ is not an intersection and therefore a path. Thus, we may fix the notation in line 17. Note, though, that the index set I for τ may be empty.*

6. *In line 21 the choice between the two options is made nondeterministically. The first option covers the case where $I_2 \cup I_3 = \emptyset$, i.e., all paths in τ are shorter than σ.[2] The only possibility to match such a constraint is to make sure that $S(\sigma) = \omega$, which is only possible if $S(a) = \omega$. Thus, a must be a variable. Clearly, a cannot be a constant different from ω, and it cannot be ω either, because then σ would have been reduced to ω in line 6 and the case in line 8 would have been applicable.*

7. *The following example shows that even if $I_2 \cup I_3 \neq \emptyset$ it must be possible to choose the first option in line 21: $\{a' \leq \beta, b \to a \leq \beta \to \alpha\}$ is matchable according to the substitution $\{\alpha \mapsto \omega, \beta \mapsto a'\}$. If the algorithm were not allowed to choose the option in line 22 it would have to choose the option in the following line which would result in the constraint set $\{a' \leq \beta, a \leq \alpha, \beta \leq b\}$. This set is clearly not matchable and the algorithm would incorrectly return **false**.*

8. *We assume that the nondeterministic choice in line 21 is made determinstically, choosing the first option, whenever $I_2 \cup I_3 = \emptyset$. In this case choosing the second option would always result in **false**.*

Algorithm 2 Match(C)

Input: $C = \{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ such that for all i at most one of σ_i and τ_i contains variables. Furthermore, all types have to be organized.
Output: **true** if C can be matched otherwise **false**

while $\exists$ nonbasic constraint in C **do**
  choose a nonbasic constraint $c = (\tau \leq \sigma) \in C$
  reduce τ and σ with respect to ω
  switch
    case: c does not contain any variables
      if $\tau \leq_{\mathbf{A}} \sigma$ **then**
        $C := C \backslash \{c\}$
      else
        return false
      end if
    case: $\sigma = \bigcap_{i \in I} \sigma_i$
      $C := C \backslash \{c\} \cup \{\tau \leq \sigma_i | i \in I\}$

    write $\tau = \bigcap_{i \in I} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$, $\sigma = \sigma_1 \to \ldots \to \sigma_m \to a$
    write $I_1 = \{i \in I | m_i < m\}$, $I_2 = \{i \in I | m_i = m\}$, $I_3 = \{i \in I | m_i > m\}$
    case: σ contains variables, τ does not contain any variables
      if $a \in \mathbb{V}$ **then**
        CHOICE:
          1.) $C := C \backslash \{c\} \cup \{\omega \leq a\}$
          2.) choose $\emptyset \neq I' \subseteq I_2 \cup I_3$
            $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i,j}} | i \in I', 1 \leq j \leq m\} \cup$
            $\{\bigcap_{i \in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq a\}$
      else
        choose $i_0 \in I_2$
        $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i_0,j}} | 1 \leq j \leq m\} \cup \{p_{i_0} \leq a\}$
      endif
    case: τ contains variables, σ does not contain any variables
      choose $i_0 \in I_1 \cup I_2$
      $C := C \backslash \{c\} \cup \{\overline{\sigma_j} \leq \overline{\tau_{i_0,j}} | 1 \leq j \leq m_{i_0}\} \cup \{p_{i_0} \leq \sigma_{m_{i_0}+1} \to \ldots \to \sigma_m \to a\}$
  end switch
end while
if C is consistent **then**
  return true
else
  return false
end if

Lemma 14 *Algorithm 2 terminates.*

Proof: The reduction-step in line 6 does not increase the height of the types involved. Furthermore, in every iteration of the **while**-loop the constraint c is either removed from C or it is replaced by a finite number of constraints, where for each newly added constraint at least one of the occurring types has a syntax-tree whose height is strictly smaller than the height of the syntax-tree of one of the types in c. Thus, if the algorithm does not return **false** it has to leave the **while**-loop after a finite number of iterations because all remaining constraints are basic. The consistency check terminates because there can only be a finite number of basic constraints. Thus, it only requires to check a finite number of constraints $\tau_i \leq \sigma_j$ not containing any type variables which can be done using the PTIME-procedure by [15]. □

Next, we prove that Alg. 2 operates in nondeterministic polynomial time. We first need a series of technical definitions and lemmas.

Definition 15 *Let τ be a type. The set of arguments $arg(\tau)$ of τ is inductively defined as follows:*

$$\begin{aligned} arg(a) &= \emptyset \\ arg(\bigcap_{i\in I} \tau_i) &= \bigcup_{i\in I} arg(\tau_i) \\ arg(\tau' \to \tau'') &= \{\tau'\} \cup arg(\tau') \cup arg(\tau'') \end{aligned}$$

Lemma 16 *Let ρ be a type and let ρ' be a subterm of ρ. Then $arg(\rho') \subseteq arg(\rho)$.*

Proof: The statement directly follows from Definition 15, using a structural induction argument:

If ρ is a type constant it is clear that the statement holds. For $\rho = \bigcap_{i\in I} \rho_i$ and $\rho' = \bigcap_{i\in I'} \rho_i$ for a subset $I' \subseteq I$ we have $arg(\rho') = \bigcup_{i\in I'} arg(\rho_i) \subseteq \bigcup_{i\in I} arg(\rho_i) = arg(\rho)$. If ρ' is a subterm of one of the ρ_i, then we know by induction that $arg(\rho') \subseteq arg(\rho_i) \subseteq \bigcup_{i\in I} arg(\rho_i) = arg(\rho)$. Finally, let $\rho = \rho_1 \to \rho_2$. If $\rho' = \rho_i$ for $i \in \{1,2\}$, then it is clear that $arg(\rho') \subseteq \{\rho_1\} \cup arg(\rho_1) \cup arg(\rho_2) = arg(\rho)$. If ρ' is a subterm of ρ_i for $i \in \{1,2\}$, then an analogous induction argument as in the previous case shows that $arg(\rho') \subseteq arg(\rho)$ holds. □

Lemma 17 *Let σ be a type and let $\rho \in arg(\sigma)$ be an argument of σ. Then $arg(\overline{\rho}) \subseteq arg(\sigma)$.*

Proof: We first show by induction that for every type τ we have $arg(\tau) = arg(\overline{\tau})$. If $\tau = a$ we have $a = \overline{a}$ and nothing has to be proved. If $\tau = \bigcap_{i\in I} \tau_i$ we have $arg(\tau) = arg(\bigcap_{i\in I} \tau_i) = \bigcup_{i\in I} arg(\tau_i) = \bigcup_{i\in I} arg(\overline{\tau_i}) = arg(\bigcap_{i\in I} \overline{\tau_i}) = arg(\overline{\tau})$. If $\tau = \tau' \to \tau''$ where $\overline{\tau''} = \bigcap_{i\in I} \tau_i''$ we have $arg(\tau) = \{\tau'\} \cup arg(\tau') \cup arg(\tau'') = \{\tau'\} \cup arg(\tau') \cup arg(\overline{\tau''}) = \{\tau'\} \cup arg(\tau') \cup arg(\bigcap_{i\in I} \tau_i'') = \{\tau'\} \cup arg(\tau') \cup \bigcup_{i\in I} arg(\tau_i'') = \bigcup_{i\in I} \left(\{\tau'\} \cup arg(\tau') \cup arg(\tau_i'')\right) = \bigcup_{i\in I} arg(\tau' \to$

2 In particular, $\tau = \omega$ is allowed if furthermore $I_1 = \emptyset$, as well.

$\tau_i'') = arg(\bigcap_{i\in I} \tau' \to \tau_i'') = arg(\overline{\tau})$. From $arg(\tau' \to \tau'') = \{\tau'\} \cup arg(\tau') \cup arg(\tau'')$ it immediately follows that for every $\rho \in arg(\sigma)$ we have $arg(\rho) \subseteq arg(\sigma)$. Together, these two observations yield the statement of the lemma. □

Lemma 18 *Let ρ be a type occurring in a non-basic constraint considered by Alg. 2 during the execution of the **while**-loop. Let $\{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ be the set of initial constraints given to the algorithm.*

There exists $1 \leq i \leq n$ such that ρ is a subterm of τ_i or σ_i or of an organized argument of τ_i or σ_i.

Proof: We prove by induction over the execution of the **while**-loop that every type occurring in a nonbasic constraint in C has the desired property. The statement of the lemma then directly follows because the algorithm only considers nonbasic constraints in C.

Before the first execution of the **while**-loop it is clear that the property holds for every type occurring in a nonbasic constraint in C because C only contains initial constraints. We now consider one execution of the **while**-loop. By induction we know that the property holds for every type occurring in a nonbasic constraint in C *before* the execution, and we have to show that the property also holds *after* the execution. If the execution of the **while**-block does not return **false** it is always the case that one nonbasic constraint c is removed from C and possibly some new constraints (basic and nonbasic) are added to C. Thus, it suffices to show that for every type in the new *nonbasic* constraints that are added the property holds. Fix $c = (\tau \leq \sigma)$ the nonbasic constraint that is considered by the algorithm in the current execution of the **while**-loop. τ and σ have the desired property. We now consider all possible cases. If it is not clear whether a new constraint is basic or nonbasic[3] we implicitly do the following arguments only for the new nonbasic constraints that are added.

Line 8: In the block following this case no new constraint is added to C and therefore there is nothing to prove.

Line 14: We have $\sigma = \bigcap_{i\in I} \sigma_i$ and $\tau \leq \sigma_i$ were added for all $i \in I$. The property holds for τ. Because the property held for σ and because every σ_i is a subterm of σ the property also holds for every σ_i.[4]

Line 19: The constraints added to C in lines 22 and 25 are basic constraints (because a is a variable). We explain why the constraints added in lines 24 and 28 have the desired property.

We start with the constraint $p_{i_0} \leq a$ that is added in line 28. p_{i_0} is a subterm of τ_{i_0} which itself is a subterm of τ. Thus, p_{i_0} is a subterm of τ. Because the property holds for τ, i.e., τ is a subterm of an initial type or of an organized argument of an initial type, the property clearly also holds for p_{i_0}. An analogous argument can be made for a and σ.

3 In line 15 this is the case, for example.

4 Note that $I = \emptyset$ is possible, subsuming the case $\sigma = \omega$.

Consider $\overline{\sigma_j}$ in one of the constraints added to C in line 24. σ_j is an argument of σ and we know that σ is a subterm of an initial type or of an organized argument of an initial type. In the first case Lem. 16 shows that σ_j is also an argument of this initial type. Thus, $\overline{\sigma_j}$ is an organized argument of an initial type and the property holds. In the second case denote by $\overline{\rho}$ the organized argument of an initial type which σ is a subterm of. Because σ_j is an argument of σ, Lem. 16 shows that it is also an argument of $\overline{\rho}$. We may now use Lem. 17 to conclude that σ_j is also an argument of the initial type ρ was an argument of. Therefore, $\overline{\sigma_j}$ is an organized argument of an initial type and we are done. In order to show that the property holds for a type $\overline{\tau_{i,j}}$ in one of the constraints added to C in line 24 we follow an analogous argument. Furthermore, the property also holds for the types $\overline{\sigma_j}$ and $\overline{\tau_{i_0,j}}$ in one of the constraints added to C in line 28 with the same argument.

Line 30: The argument that the constraints added in line 32 have the desired property is completely analogous to the previous argument.

□

Corollary 19 *The **while**-loop of Alg. 2 is executed polynomially often in the size of the set* $\{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ *of constraints initially given to the algorithm.*

Proof: From Lem. 18 we know that every type occurring in a nonbasic constraint that the algorithm may have to consider during one execution of the **while**-loop is a subterm of a type occurring in an initial constraint or a subterm of an organized argument of a type occurring in an initial constraint (we call such types initial types). The number of subterms of an initial type is linear. The number of arguments of an initial type is also linear. Organizing each of these linearly many arguments causes only a polynomial blowup. Therefore, it is clear that the number of subterms of an organized argument of an initial type is polynomial. Let k denote the number of subterms of the initial types plus the number of subterms of organized arguments of the initial types. The total number of nonbasic constraints that the algorithm considers is bounded by k^2.

Because we use memoization to make sure that no constraint is considered more than once by the algorithm and because during each iteration of the **while**-loop exactly one nonbasic constraint is considered, it is clear that the loop is iterated at most k^2 times. □

Corollary 20 *The size of a new constraint added to C during the execution of the **while**-loop of Alg. 2 is of polynomial size in the size of the set* $\{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ *of constraints initially given to the algorithm.*

Proof: Lemma 18 shows that each type occurring in a newly added constraint is either a subterm of an initial type or of an organized argument of an initial type. It is clear that each of these subterms is of polynomial size. □

Aggregating the results we obtain a non-deterministic polynomial upper bound:

Lemma 21 *Algorithm 2 operates in nondeterministic polynomial time.*

Proof: Corollary 19 shows that the **while**-loop is iterated a polynomial number of times. It remains to show that every such iteration only causes polynomial cost in the input: The reduction-step with respect to ω in line 6 can be implemented in linear time if it is done bottom-up, removing every occurrence of ω as component in an intersection and replacing empty intersections and arrows of the form $\rho \to \omega$ by ω. The case in line 9 requires a check whether c already holds. This can be done, using the PTIME-procedure for deciding subtyping proposed by [15]. The other cases only require the construction of the new constraints which clearly can be done in polynomial nondeterministic time. Consistency of C can also be checked in polynomial time because it boils down to checking a polynomial number of subtyping relations (without variables). Again, this can be done, using the PTIME-procedure mentioned above. Corollary 20 shows that each added constraint is of polynomial size, which means that each of the operations above can indeed be done in polynomial time. The memoization does not exceed polynomial time because we have already seen that there is at most a polynomial number of constraints, that are of polynomial size, that can possibly be considered.

Together with the termination-argument from Lem. 14 this shows that the algorithm operates in nondeterministic polynomial time. □

We make some remarks about this proof:

Remark 22

1. *The statement of the previous lemma might come as a surprise since the execution of the **while**-loop requires a repeated organization of the arguments of the occurring types. It can be asked why this repeated organization does not result in a normalization [10] of the types involved. As mentioned before, this could cause an exponential blowup in the size of the type. The reason why this problem does not occur is the fact that this organization is interleaved with decomposition steps. We illustrate this by the following small example. We inductively define two families of types:*

$$\begin{aligned} \tau_0 &= a_0 \cap b_0 & \sigma_0 &= \alpha_0 \\ \tau_l &= \tau_{l-1} \to (a_l \cap b_l) & \sigma_l &= \sigma_{l-1} \to \alpha_l \end{aligned}$$

*The size of τ_n in normalized form is exponential in n. However, if the algorithm processes the constraint $\tau_n \leq \sigma_n$ only a polynomial number of new constraints (of polynomial size) are constructed: First, the types have to be organized. We obtain $(\tau_{n-1} \to a_n) \cap (\tau_{n-1} \to b_n) \leq \sigma_n$. In the first iteration of the **while**-loop the case in line 20 applies and the nondeterministic choice in line 21 may be resolved in such a way that a subset of components of the toplevel intersection of $(\tau_{n-1} \to a_n) \cap (\tau_{n-1} \to b_n)$ has to be chosen. In order to maximize the size of C we choose both components which forces the construction of the following constraints: $a_n \cap b_n \leq \alpha_n$, $\overline{\sigma_{n-1}} \leq \overline{\tau_{n-1}}$, and $\overline{\sigma_{n-1}} \leq \overline{\tau_{n-1}}$. The last two constraints are the same, however, and therefore the memoization of the algorithm makes sure that this constraint is only treated once. In the next*

step the case in line 14 applies (note that $\overline{\tau_{n-1}}$ is a top-level intersection) and the constraints $\overline{\sigma_{n-1}} \leq \tau_{n-2} \to a_{n-1}$ and $\overline{\sigma_{n-1}} \leq \tau_{n-2} \to b_{n-1}$ are created. For both constraints the same rule applies and causes a change of C according to line 32. This leads to the construction of the basic constraints $\alpha_{n-1} \leq a_{n-1}$ and $\alpha_{n-1} \leq b_{n-1}$ as well as to the construction of $\overline{\sigma_{n-2}} \leq \overline{\tau_{n-2}}$ and $\overline{\sigma_{n-2}} \leq \overline{\tau_{n-2}}$. Then, the same argument as above can be repeated.

We conclude that the doubling of the arguments of the τ_l that occurs in the normalization (and which eventually causes the exponential blowup if repeated) does not occur in the algorithm, because the types involved are decomposed such that the arguments and targets are treated separately. This implies that the arguments cannot be distinguished any more such that the new constraints coincide and are only added once.

2. *The proof of Lem. 21 relies on the fact that every new nonbasic constraint added to C only contains types that are essentially subterms of an initial type. A new intersection which possibly does not occur as a subterm in any of the initial types has to be constructed in line 25, though. Since, in principle, this new intersection represents a subset of an index set, it is not clear that there cannot be an exponential number of such basic constraints. However, this construction of new intersections only happens as a consequence to the nonbasic constraint that is treated there. As noted above there can be at most a polynomial number of nonbasic constraints and therefore new intersections can also only be introduced a polynomial number of times.*

We now show correctness (soundness and completeness) of the algorithm, i.e., it can return **true** if and only if the original set of constraints can be matched. We need some auxiliary lemmas, first:

Lemma 23 *Let τ be a type and let S be a substitution. Then $S(\tau) = S(\overline{\tau})$*[5]*.*

Proof: We prove the statement by structural induction on the organization rules:

If τ is an atom, then $\tau = \overline{\tau}$ and nothing has to be proved. If $\tau = \tau' \cap \tau''$ we have $S(\tau) = S(\tau') \cap S(\tau'') = S(\overline{\tau'}) \cap S(\overline{\tau''}) = S(\overline{\tau'} \cap \overline{\tau''}) = S(\overline{\tau' \cap \tau''}) = S(\overline{\tau})$. If $\tau = \tau' \to \tau''$ with $\overline{\tau''} = \bigcap_{i \in I} \tau_i''$ we have $S(\tau) = S(\tau') \to S(\tau'') = S(\tau') \to S(\overline{\tau''}) = S(\tau') \to S(\bigcap_{i \in I} \tau_i'') = S(\tau') \to \bigcap_{i \in I} S(\tau_i'') = \bigcap_{i \in I} (S(\tau') \to S(\tau_i'')) = \bigcap_{i \in I} S(\tau' \to \tau_i'') = S(\bigcap_{i \in I} (\tau' \to \tau_i'')) = S(\overline{\tau})$. □

The following two lemmas are derived using Lem. 1.

Lemma 24 *Let $\tau = \bigcap_{i \in I} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$ be an organized type and let $\sigma = \sigma_1 \to \ldots \to \sigma_m \to \rho$ be a type. $\tau \leq_{\mathbf{A}} \sigma$ if and only if there is a nonempty subset $I' \subseteq I$ such that for all $i \in I'$ and all $1 \leq j \leq m$ we have $\overline{\sigma_j} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ and such that $\bigcap_{i \in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \rho$.*

Proof: We prove both directions by induction over m.

$\Rightarrow$: $m = 0$: We have $\tau \leq_{\mathbf{A}} \sigma = \rho$. Choosing $I' = I$ the statement holds because we have $\tau = \bigcap_{i \in I} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$.

5 Equality refers to the identification of types σ and σ' if both $\sigma' \leq_{\mathbf{A}} \sigma$ and $\sigma \leq_{\mathbf{A}} \sigma'$ hold.

$m \Rightarrow m+1$: We have $\tau \leq_{\mathbf{A}} \sigma$. We write $\sigma = \sigma_1 \to \sigma'$, i.e., $\sigma' = \sigma_2 \to \ldots \to \sigma_{m+1} \to \rho$, and $\tau = \bigcap_{j\in J} p_j \cap \bigcap_{i\in I''} \tau_{i,1} \to \tau'_i$, i.e., $J \subseteq I$ is the subset of I where τ_i is an atom and $\tau'_i = \tau_{i,2} \to \ldots \to \tau_{i,m_i} \to p_i$. Lemma 1 states that there is a nonempty subset H of I'' such that for all $i \in H$ we have $\sigma_1 \leq_{\mathbf{A}} \tau_{i,1}$ and $\bigcap_{i\in H} \tau'_i \leq_{\mathbf{A}} \sigma'$. The second inequality can be rewritten as $\bigcap_{i\in H} \tau_{i,2} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \sigma_2 \to \ldots \to \sigma_{m+1} \to \rho$. Applying the induction hypothesis to this inequality we see that there is a subset I' of H such that for all $i \in I'$ and all $2 \leq j \leq m+1$ we have $\overline{\sigma_j} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ and such that $\bigcap_{i\in I'} \tau_{i,m+2} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \rho$. Because $I' \subseteq H$ for all $i \in I'$ we have $\sigma_1 \leq_{\mathbf{A}} \tau_{i,1}$ which is equivalent to $\overline{\sigma_1} \leq_{\mathbf{A}} \overline{\tau_{i,1}}$. Therefore the choice $I' \subseteq I$ satisfies the requirements in the lemma.

$\Leftarrow$: $m = 0$: We get $\tau = \bigcap_{i\in I} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \bigcap_{i\in I'} \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \rho = \sigma$, where the first inequality follows from $I' \subseteq I$ and the second inequality follows from the assumption.

$m \Rightarrow m+1$: We write $\sigma = \sigma_1 \to \sigma'$, i.e., $\sigma' = \sigma_2 \to \ldots \to \sigma_{m+1} \to \rho$, and $\tau = \bigcap_{j\in J} p_j \cap \bigcap_{i\in I''} \tau_{i,1} \to \tau'_i$, i.e., $J \subseteq I$ is the subset of I where τ_i is an atom and $\tau'_i = \tau_{i,2} \to \ldots \to \tau_{i,m_i} \to p_i$, and we assume that there is a nonempty subset $I' \subseteq I$ such that for all $i \in I'$ and all $1 \leq j \leq m+1$ we have $\overline{\sigma_j} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ and such that $\bigcap_{i\in I'} \tau_{i,m+2} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \rho$. We want to show $\tau \leq_{\mathbf{A}} \sigma$, and according to Lem. 1 we have to show that the subset H of I'' with $H = \{i \in I'' | \sigma_1 \leq_{\mathbf{A}} \tau_{i,1}\}$ is nonempty and that $\bigcap_{i\in H} \tau'_i \leq_{\mathbf{A}} \sigma'$. It is clear that $I' \subseteq H$ because for all $i \in I'$ we have $\sigma_1 = \overline{\sigma_1} \leq_{\mathbf{A}} \overline{\tau_{i,1}} = \tau_{i,1}$, and therefore H is nonempty. It remains to show $\bigcap_{i\in H} \tau'_i \leq_{\mathbf{A}} \sigma'$. We have $\bigcap_{i\in H} \tau'_i \leq_{\mathbf{A}} \bigcap_{i\in I'} \tau'_i = \bigcap_{i\in I'} \tau_{i,2} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} \sigma_2 \to \ldots \to \sigma_{m+1} \to \rho = \sigma'$, where the first inequality uses $I' \subseteq H$ and the second inequality follows from the induction hypothesis (note that the induction hypothesis may indeed be applied because I' itself has the desired properties).

□

Lemma 25 *Let $\tau = \tau_1 \to \ldots \to \tau_m \to \rho$ be a type and let $\sigma = \sigma_1 \to \ldots \to \sigma_n \to p$ be a path with $m \leq n$. $\tau \leq_{\mathbf{A}} \sigma$ if and only if for all $1 \leq j \leq m$ we have $\sigma_j \leq_{\mathbf{A}} \tau_j$ and $\rho \leq_{\mathbf{A}} \sigma_{m+1} \to \ldots \to \sigma_n \to p$.*

Proof: We prove both directions by induction over m.

$\Rightarrow$: $m = 0$: We have $\tau \leq_{\mathbf{A}} \sigma$ and $\tau = \rho$. This implies $\rho \leq_{\mathbf{A}} \sigma_1 \to \ldots \to \sigma_n \to p$.

$m \Rightarrow m+1$: We write $\sigma = \sigma_1 \to \sigma'$, i.e., $\sigma' = \sigma_2 \to \ldots \to \sigma_n \to p$, and $\tau = \tau_1 \to \tau'$, i.e., $\tau' = \tau_2 \to \ldots \to \tau_{m+1} \to \rho$. We assume $\tau \leq_{\mathbf{A}} \sigma$. Lemma 1 shows that this is only possible if $\sigma_1 \leq_{\mathbf{A}} \tau_1$ and $\tau' \leq_{\mathbf{A}} \sigma'$. The second inequality is equivalent to $\tau_2 \to \ldots \to \tau_{m+1} \to \rho \leq_{\mathbf{A}} \sigma_2 \to \ldots \to \sigma_n \to p$. Applying the induction hypothesis we get $\sigma_j \leq_{\mathbf{A}} \tau_j$ for all $2 \leq j \leq m+1$ and $\rho \leq_{\mathbf{A}} \sigma_{m+2} \to \ldots \to \sigma_n \to p$.

$\Leftarrow$: $m = 0$: We have $\tau = \rho \leq_{\mathbf{A}} \sigma_1 \to \ldots \to \sigma_n \to p = \sigma$.

$m \Rightarrow m+1$: We write $\sigma = \sigma_1 \to \sigma'$, i.e., $\sigma' = \sigma_2 \to \ldots \to \sigma_n \to p$, and $\tau = \tau_1 \to \tau'$, i.e., $\tau' = \tau_2 \to \ldots \to \tau_{m+1} \to \rho$, and we assume that for all $1 \leq j \leq m+1$ we have $\sigma_j \leq_{\mathbf{A}} \tau_j$ and $\rho \leq_{\mathbf{A}} \sigma_{m+2} \to \ldots \to \sigma_n \to p$. In order to show that $\tau \leq_{\mathbf{A}} \sigma$ holds, according to Lem. 1 we have to show that $\tau_1 \leq_{\mathbf{A}} \sigma_1$ and $\sigma' \leq_{\mathbf{A}} \tau'$ hold. The first inequality holds by assumption and the second inequality follows from the induction hypothesis.

□

We start by proving soundness of Alg. 2. The following auxiliary lemma consists of a detailed case analysis and states that every substitution that matches a set of constraints C' resulting from a set C by one execution of the **while**-loop also matches C.

Lemma 26 *Let C be a set of constraints and let C' be a set of constraints that results from C by application of one of the cases of Alg. 2.*

Every substitution that matches C' also matches C.

Proof: For all cases C' results from C by removing the constraint c and possibly by further adding some new constraints. Assuming we have a substitution S matching C', it suffices to show that S satisfies c in order to show that it also satisfies C. We do this for all cases separately:

Line 10: c does not contain any variables and $\tau \leq_{\mathbf{A}} \sigma$. Thus, S matches c.

Line 15: We have $\sigma = \bigcap_{i \in I} \sigma_i$ and the constraints $\tau \leq \sigma_i$ were added. Because S matches C' we have $S(\tau) \leq_{\mathbf{A}} S(\sigma_i)$ for all $i \in I$. By idempotence and monotonicity of $\cap$ we get $S(\tau) \leq_{\mathbf{A}} \bigcap_{i \in I} S(\sigma_i) = S\big(\bigcap_{i \in I} \sigma_i\big) = S(\sigma)$.[6]

For the remaining cases we have $\sigma = \sigma_1 \to \ldots \to \sigma_m \to a$ and $\tau = \bigcap_{i \in I} \tau_i$ with $\tau_i = \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$.

Line 22: $a = \alpha$ is a variable and the constraint $\omega \leq \alpha$ was added. Because S matches C' we must have $S(\alpha) = \omega$. Then it is clear that S matches $\tau \leq \sigma$ because $S(\sigma) = S(\sigma_1) \to \ldots \to S(\sigma_m) \to \omega = \omega$.

Line 24: $a = \alpha$ is a variable and $\overline{\sigma_j} \leq \overline{\tau_{i,j}}$ for all $i \in I'$ and all $1 \leq j \leq m$ and $\bigcap_{i \in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq \alpha$ were added. Since S matches C' we know $S(\overline{\sigma_j}) \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ and $\bigcap_{i \in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} S(\alpha)$. We want to show $\tau \leq_{\mathbf{A}} S(\sigma)$. We write $S(\sigma) = \sigma'_1 \to \ldots \to \sigma'_m \to \rho$ where $S(\sigma_j) = \sigma'_j$ and $S(\alpha) = \rho$. We have $\overline{\sigma'_j} = \sigma'_j = S(\sigma_j) = S(\overline{\sigma_j}) \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ where the third equality follows from Lem. 23. We may apply the "if"-part of Lem. 24 to conclude $\tau \leq_{\mathbf{A}} S(\sigma)$.

6 No argument is necessary for the special case $\sigma = \omega$, i.e., $I = \emptyset$, because no new constraints are added.

Line 28: a is a constant and the constraints $\overline{\sigma_j} \leq \overline{\tau_{i_0,j}}$ for all $1 \leq j \leq m$ and $p_{i_0} \leq a$ were added. Since S matches C' we know $S(\overline{\sigma_j}) \leq_{\mathbf{A}} \overline{\tau_{i_0,j}}$ for all $1 \leq j \leq m$ and $p_{i_0} \leq_{\mathbf{A}} a$. The second inequality implies $p_{i_0} = a$. On the other hand, using Lem. 23 we get $S(\sigma_j) = S(\overline{\sigma_j}) \leq_{\mathbf{A}} \overline{\tau_{i_0,j}} = \tau_{i_0,j}$ for all $1 \leq j \leq m$. Lemma 2 implies $\tau \leq_{\mathbf{A}} S(\sigma)$ and S matches c.

Line 32: The constraints $\overline{\sigma_j} \leq \overline{\tau_{i_0,j}}$ for all $1 \leq j \leq m_{i_0}$ and $p_{i_0} \leq \sigma_{m_{i_0}+1} \to \ldots \to \sigma_m \to a$ were added. Because S matches C' we know $\overline{\sigma_j} \leq_{\mathbf{A}} S(\overline{\tau_{i_0,j}})$ and $S(p_{i_0}) \leq_{\mathbf{A}} \sigma_{m_{i_0}+1} \to \ldots \to \sigma_m \to a$. This implies $\sigma_j = \overline{\sigma_j} \leq_{\mathbf{A}} S(\overline{\tau_{i_0,j}}) = S(\tau_{i_0,j})$, the last equality again following from Lem. 23. We may apply the "if"-part of Lem. 25 to conclude $S(\tau_{i_0}) \leq_{\mathbf{A}} \sigma$. Because $S(\tau_{i_0})$ is a component of $S(\tau)$ it is clear that $S(\tau) \leq_{\mathbf{A}} S(\tau_{i_0}) \leq_{\mathbf{A}} \sigma$. Therefore, S matches $\tau \leq \sigma$.

□

The following corollary uses the previous lemma as well as the result of Lem. 12 which states that a set of basic constraints is consistent if and only if it is matchable:

Corollary 27 *Algorithm 2 is sound.*

Proof: Assume that the algorithm returns **true**. This is only possible in line 36 if the algorithm leaves the **while**-loop with a consistent set C of basic constraints. By the "if"-direction of Lem. 12, C is matchable. Using Lem. 26, an inductive argument shows that *all* sets of constraints considered in the algorithm during execution of the **while**-loop are matchable. This is in particular true for the initial set of constraints. □

For proving completeness we need an auxiliary lemma which states that for a matchable set of constraints a choice can be made in the **while**-loop such that the resulting set of constraints is also matchable. Again, the proof comes down to a detailed case analysis.

Lemma 28 *Let C be matchable and $c \in C$.*
There exists a set of constraints C' such that C' results from C by application of one of the cases of Alg. 2 to c and C' is matchable.

Proof:
We show that no matter which case applies to c, a choice can be made that results in a matchable set C'. In particular, it must be argued that there is a choice that does not result in **false**.

Furthermore, note that for all cases C' results from C by removing c and by possibly adding some new constraints. Let S be a substitution that matches C. In order to show that it also matches C' it suffices to show that it matches the newly introduced constraints. If there are no new constraints we do not have to show anything.

Line 8: c does not contain any variables. Because C is matchable c holds and the case results in the set $C' = C \backslash \{c\}$ (line 10) and not in **false**.

Line 14: $\sigma = \bigcap_{i\in I} \sigma_i$ and $C' = C\backslash\{c\} \cup \{\tau \leq \sigma_i | i \in I\}$. We have to show $S(\tau) \leq_{\mathbf{A}} S(\sigma_i)$ for all $i \in I$. This holds if and only if $S(\tau) \leq_{\mathbf{A}} \bigcap_{i\in I} S(\sigma_i)$. But this clearly holds because $\bigcap_{i\in I} S(\sigma_i) = S(\sigma)$ and S matches c.[7]

For the remaining cases we have $\sigma = \sigma_1 \to \ldots \to \sigma_m \to a$ and $\tau = \bigcap_{i\in I} \tau_i$ with $\tau_i = \tau_{i,1} \to \ldots \to \tau_{i,m_i} \to p_i$.

Line 19: σ contains variables and τ does not. We distinguish cases:

$a \in \mathbb{V}$: We know that S matches $\tau \leq \sigma$. If $S(a) = \omega$, in line 21 we choose the first option. Then, the only new constraint is $\omega \leq a$ which is clearly matched by S.

Otherwise, i.e., $S(a) \neq \omega$, the second option is chosen. Because $\tau \leq_{\mathbf{A}} S(\sigma)$ holds we may apply the "only if"-part of Lem. 24 to conclude that there is a nonempty subset I' of I such that for all $i \in I'$ and all $1 \leq j \leq m$ we have $\overline{S(\sigma_j)} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ and such that $\bigcap_{i\in I'} \tau_{i,m+1} \to \ldots \to \tau_{i,m_i} \to p_i \leq_{\mathbf{A}} S(a)$. This implies in particular that for all $i \in I'$ we have $m_i \geq m$, and thus we may choose $I' \subseteq I_2 \cup I_3$ in line 23. From $\overline{S(\sigma_j)} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ we infer $S(\overline{\sigma_j}) = S(\sigma_j) = \overline{S(\sigma_j)} \leq_{\mathbf{A}} \overline{\tau_{i,j}}$ where the first equality holds because of Lem. 23. This shows that S matches all newly introduced constraints.

$a \notin \mathbb{V}$: Because $a \notin \mathbb{V}$ we know that $S(\sigma)$ is still a path. Thus, using $\tau \leq_{\mathbf{A}} S(\sigma)$ together with Lem. 2 we conclude that there exists an index $i_0 \in I$ such that $m_{i_0} = m$, $p_{i_0} = a$, and $S(\sigma_j) \leq_{\mathbf{A}} \tau_{i_0,j}$ for all $j \leq m$. Therefore, in line 27 we may choose exactly this i_0 (note that i_0 is indeed contained in I_2 because $m_{i_0} = m$). Using Lem. 23, we conclude $S(\overline{\sigma_j}) \leq_{\mathbf{A}} \overline{\tau_{i_0,j}}$ for all $j \leq m$ from $S(\sigma_j) \leq_{\mathbf{A}} \tau_{i_0,j}$ and from $p_{i_0} = a$ we conclude $p_{i_0} = S(a)$. Therefore, all newly introduced constraints are matched by S.

Line 30: We know $S(\tau) \leq_{\mathbf{A}} \sigma$. Write $\overline{S(\tau)} = \bigcap_{h\in H} \rho_{h,1} \to \ldots \to \rho_{h,n_h} \to b_h$. Applying Lem. 2 to $\overline{S(\tau)} \leq_{\mathbf{A}} \sigma$, we conclude that there exists $h_0 \in H$ with $b_{h_0} = a$, $n_{h_0} = m$, and $\sigma_l \leq_{\mathbf{A}} \rho_{h_0,l}$ for all $1 \leq l \leq m$. Note that with $\pi_{h_0} = \rho_{h_0,1} \to \ldots \to \rho_{h_0,m} \to a$ this implies $\pi_{h_0} \leq_{\mathbf{A}} \sigma$. For π_{h_0} there must be an index $j_0 \in I$ such that π_{h_0} occurs as a component in $\overline{S(\tau_{j_0})}$. It is clear that $m_{j_0} \leq m$. Otherwise all paths in this type would be of length greater than m (neither a substitution nor an organization may reduce the length of a path). Thus, j_0 as above is contained in $I_1 \cup I_2$ (cf. line 18), and we may choose $i_0 = j_0$ in line 31.

We have to show that S matches $\overline{\sigma_l} \leq \overline{\tau_{j_0,l}}$ for all $1 \leq l \leq m_{j_0}$ and $p_{j_0} \leq \sigma_{m_{j_0}+1} \to \ldots \to \sigma_m \to a$. Because π_{h_0} is a component in $\overline{S(\tau_{j_0})}$, it is clear that $\pi_{h_0} \leq_{\mathbf{A}} \sigma$ implies $\overline{S(\tau_{j_0})} \leq_{\mathbf{A}} \sigma$ and, thus, also $S(\tau_{j_0}) \leq_{\mathbf{A}} \sigma$. Applying the "only if"-part of Lem. 25 to $S(\tau_{j_0,1}) \to \ldots \to S(\tau_{j_0,m_{j_0}}) \to S(p_{j_0}) = S(\tau_{j_0}) \leq_{\mathbf{A}} \sigma = \sigma_1 \to \ldots \to \sigma_m \to a$, we obtain $\sigma_l \leq_{\mathbf{A}} S(\tau_{j_0,l})$ for all $1 \leq l \leq m_{j_0}$ and $S(p_{j_0}) \leq_{\mathbf{A}} \sigma_{m_{j_0}+1} \to \ldots \to \sigma_m \to a$. We get $\overline{\sigma_l} = \sigma_l \leq_{\mathbf{A}} S(\tau_{j_0,l}) = S(\overline{\tau_{j_0,l}})$, the last equality following from Lem. 23. Altogether this shows that S matches the newly introduced constraints.

7 For the special case $\sigma = \omega$, i.e., $I = \emptyset$, no new constraints are added.

□

This lemma together with Lem. 12 proves the following corollary:

Corollary 29 *Algorithm 2 is complete.*

Proof: We assume that the initial set C of constraints is matchable. We have to show that there is an execution sequence of the algorithm that results in **true**. Using Lem. 28 in an inductive argument it can be shown that for every iteration of the **while**-loop it is possible to make the nondeterministic choice in such a way that the iteration results in a matchable set of constraints. Thus, there is an execution sequence of the **while**-loop that results in a matchable set of basic constraints. Lemma 12 shows that this set is consistent and therefore the algorithm returns **true**. □

Corollary 30 *Algorithm 2 is correct.*

Proof: Immediate from Cor.(s) 27 and 29. □

Summarizing the results we get the main theorem of this chapter:

Theorem 31 cMATCH *is* NP*-complete.*

Proof: Immediate from Cor.(s) 9 and 30 and Lem. 21. □

5 MATCHING-BASED OPTIMIZATIONS

Matching can be used to further optimize Alg. 1. First (Sect. 5.1), we filter out types in Γ that cannot contribute to inhabiting τ due to a failing matching condition. Second (Sect. 5.2), we further filter the types by a lookahead check, against new inhabitation targets, for a necessary matching condition.

Note that variables occurring in all inhabitation goals can be considered to be constants, because we may only instantiate variables occurring in Γ. Whenever a combinator $(x : \sigma)$ is chosen from Γ we may assume that all variables occurring in σ are fresh. Thus, indeed, we face matching problems.

5.1 MATCHING OPTIMIZATION

Algorithm 3 below checks for every target of every component σ_j of σ and τ_i of the inhabitation goal τ whether the constraint consisting of the corresponding target and τ_i is matchable. The τ_i can be inhabited by different components of σ. The number n of arguments, however, has to be the same for all i. This condition leads to the construction of N in line 10. Note that we may need to compute $tgt_n(\sigma_j)$ in line 8 where n is larger than $m_j := \|\sigma_j\|$. Any such call to Alg. 2 in this line is assumed to return false if $tgt_{m_j}(\sigma_j)$ is a type constant. If $tgt_{m_j}(\sigma_j) = \alpha$, then we check matchability of $\alpha \leq \tau_i$. The following lemma shows that, indeed, it suffices to only consider n with $n \leq \|\sigma\| + k$ in lines 7 and 10.

Lemma 32 *Let σ be an organized type and let S be a level-k substitution.*
We have $\|S(\sigma)\| \leq \|\sigma\| + k$.

Proof: Let $\sigma_1 \to \ldots \to \sigma_m \to \alpha$ be a longest path in σ whose target is a variable. Note that $m \leq \|\sigma\|$. The worst case is that $S(\alpha)$ is a path of length k. In this case we have $\|S(\sigma_1) \to \ldots \to S(\sigma_m) \to S(\alpha)\| = m + k$. There are two cases: If $S(\sigma_1) \to \ldots \to S(\sigma_m) \to S(\alpha)$ is a longest path in $S(\sigma)$, then we have $\|S(\sigma)\| = m + k \leq \|\sigma\| + k$. Otherwise we a longest path in $S(\sigma)$ must have been created by instantiating a longest path in σ, and such a path must be longer than m (and it does not have a variable in the target!). We conclude $\|S(\sigma)\| = \|\sigma\| \leq \|\sigma\| + k$. □

We need an adaptation of Lem. 5 to prove the correctness of Alg. 3.

Lemma 33 *Let τ be a path and let $x : \sigma \in \Gamma$ where $\sigma = \bigcap_{j \in J} \sigma_j$ is organized.*
The following are equivalent conditions:

1. $\Gamma \vdash_k x\ e_1 \ldots e_m : \tau$

Algorithm 3 $\mathrm{INH2}(\Gamma, \tau, k)$

1: *Input:* Γ, τ, k — *wlog:* All types in Γ and $\tau = \bigcap_{i \in I} \tau_i$ are organized
2: *Output:* INH2 accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3:
4: *loop:*
5: CHOOSE $(x : \sigma) \in \Gamma$;
6: write $\sigma = \bigcap_{j \in J} \sigma_j$
7: **for all** $i \in I, j \in J, n \leq \|\sigma\| + k$ **do**
8: $\quad \mathrm{candidates}(i, j, n) := \mathrm{Match}(tgt_n(\sigma_j) \leq \tau_i)$
9: **end for**
10: $N := \{n \leq \|\sigma\| + k \mid \forall_{i \in I} \exists_{j \in J} : \mathrm{candidates}(i, j, n) = \textbf{true}\}$
11: CHOOSE $n \in N$;
12: **for all** $i \in I$ **do**
13: $\quad$ CHOOSE $j_i \in J$ with $\mathrm{candidates}(i, j_i, n) = \textbf{true}$
14: $\quad$ CHOOSE $S_i \in \mathcal{S}_x^{(\Gamma, \tau_i, k)}$
15: $\quad$ CHOOSE $\pi_i \in \mathbb{P}_n(\overline{S_i(\sigma_{j_i})})$
16: **end for**
17:
18: **if** $\forall_{i \in I} : tgt_n(\pi_i) \leq_{\mathbf{A}} \tau_i$ **then**
19: $\quad$ **if** $n = 0$ **then**
20: $\quad\quad$ ACCEPT;
21: $\quad$ **else**
22: $\quad\quad$ FORALL $(l = 1 \ldots n)$ $\tau := \bigcap_{i \in I} \overline{arg_l(\pi_i)}$;
23: $\quad\quad$ GOTO *loop*;
24: $\quad$ **end if**
25: **else**
26: $\quad$ FAIL;
27: **end if**

2. *There exists* $S \in \mathcal{S}_x^{(\Gamma, \tau, k)}$ *and a path* $\pi \in \mathbb{P}_m(\overline{S_\tau(\sigma)})$ *such that*
   a) $tgt_m(\pi) \leq_{\mathbf{A}} \tau$;
   b) $\Gamma \vdash_k e_l : arg_l(\pi)$, *for all* $l \leq m$.

3. *There exists* $j \in J$, $S \in \mathcal{S}_x^{(\Gamma, \tau, k)}$, *and* $\pi \in \mathbb{P}_m(\overline{S(\sigma_j)})$ *such that*
   a) $tgt_m(\pi) \leq_{\mathbf{A}} \tau$;
   b) $\Gamma \vdash_k e_l : arg_l(\pi)$, *for all* $l \leq m$.

Proof: The implication 1. $\Rightarrow$ 2. follows from Cor. 5.

We want to prove 2. $\Rightarrow$ 3 : Denote by S' and π' the substitution and path in condition 2. Because $\mathbb{P}_m(\overline{S'(\sigma)}) = \mathbb{P}_m(\bigcap_{j\in J}\overline{S'(\sigma_j)})$ it is clear that there is an index j' such that π' occurs in $\overline{S'(\sigma_{j'})}$. Choosing $j = j'$, $S = S'$, and $\pi = \pi'$, the conditions clearly hold.

The implication 3. $\Rightarrow$ 1. follows from a suitable application of the type rules. □

We get the following corollary:

Corollary 34 *Let $\tau = \bigcap_{i\in I}\tau_i$ be organized and let $(x : \sigma) \in \Gamma$ where $\sigma = \bigcap_{j\in J}\sigma_j$ is also organized. The following are equivalent conditions:*

1. $\Gamma \vdash_k x\ e_1 \dots e_m : \tau$
2. *For all $i \in I$ there exist $j_i \in J$, $S_i \in \mathcal{S}_x^{(\Gamma,\tau_i,k)}$, and $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ with*
   a) $tgt_m(\pi_i) \leq_{\mathbf{A}} \tau_i$;
   b) $\Gamma \vdash_k e_l : \bigcap_{i\in I} arg_l(\pi_i)$, *for all $l \leq m$.*

Proof: It is clear that $\Gamma \vdash_k x\ e_1 \dots e_m : \tau$ is equivalent to $\Gamma \vdash_k x\ e_1 \dots e_m : \tau_i$ for all $i \in I$. An application of the equivalence of conditions *1.* and *3.* of Lem. 33 shows that this is equivalent to the following condition: For all $i \in I$ there exist $j_i \in J$, $S_i \in \mathcal{S}_x^{(\Gamma,\tau_i,k)}$, and $\pi_i \in \mathbb{P}_m(\overline{S_i(\sigma_{j_i})})$ such that

1. $tgt_m(\pi_i) \leq_{\mathbf{A}} \tau_i$;
2. $\Gamma \vdash_k e_l : arg_l(\pi_i)$, for all $l \leq m$.

With these choices this condition is equivalent to condition *2.* of the corollary. □

Algorithm 3 is a direct realization of condition *2.* of the corollary above. This proves the correctness of the algorithm. Because cMATCH is in NP the complexity of Alg. 3 remains unchanged. Again, a combinatorial consideration shows that this optimization can lead to very large speed-ups since it prevents the consideration of useless substitutions.

5.2 MATCHING OPTIMIZATION USING LOOKAHEAD

Finally, we describe an optimization of Alg. 3 which has turned out experimentally to be immensely powerful, causing speed-ups of up to 16 orders of magnitude in some examples. The idea is to formulate a necessary condition whose violation shows that a newly generated inhabitation question cannot be solved. In this case the combination of choices of j_i, S_i, and π_i can be rejected *before* any new inhabitation goal is instantiated. Basically, the condition states that the choice of the path π_i of length n in line 15 of Alg. 3 is only meaningful if *for all* $1 \leq l \leq n$ and *all* paths π' in $\overline{arg_l(\pi_i)}$ there exists $(y : \rho) \in \Gamma$ such that ρ has a path ρ' that has a target $tgt_m(\rho')$ for some m for which $\mathrm{Match}(tgt_m(\rho') \leq \pi')$ returns **true**. If there is no such $(y : \rho)$, then $\bigcap_{i\in I}\overline{arg_l(\pi_i)}$ cannot be inhabited and the check in line 8 does not succeed for any combination. We need an auxiliary lemma:

Lemma 35 *Let τ be a type and let $\rho = \rho_1 \to \ldots \to \rho_r \to a$ be a path. Let S be a substitution. Let $\pi \in \mathbb{P}_m(\overline{S(\rho)})$ such that $tgt_m(\pi) \leq_{\mathbf{A}} \tau$.*

There exist $h \leq r$ and a substitution S' such that $S'(tgt_h(\rho)) \leq_{\mathbf{A}} \tau$. Furthermore, $l(S') \leq l(S)$ and any constants occurring in the image of S' also occur in the image of S.

Proof: We distinguish two cases:

$a \in \mathbb{V}$: Write $a = \alpha$, i.e., $\rho_1 \to \ldots \to \rho_r \to \alpha$. Again we distinguish two cases:

- $m \leq r$: Then we can write $tgt_m(\pi) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b \leq_{\mathbf{A}} \tau$, where $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$. Choosing $h = m$ and $S' = S$, we get $S'(tgt_h(\rho)) = S(tgt_m(\rho)) = S(tgt_m(\rho)) = S(\rho_{m+1} \to \ldots \to \rho_r \to \alpha) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to S(\alpha) \leq_{\mathbf{A}} S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b \leq_{\mathbf{A}} \tau$ where the first inequality follows because $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$.
- $m > r$: Then, we can write $\pi = S(\rho_1) \to \ldots \to S(\rho_r) \to \pi_{r+1} \to \ldots \to \pi_s \to b$, where $\pi_{r+1} \to \ldots \to \pi_s \to b$ is a component in $\overline{S(\alpha)}$. We get $tgt_m(\pi) = \pi_{m+1} \to \ldots \to \pi_s \to b \leq_{\mathbf{A}} \tau$. We choose $h = r$ and we define $S' = \{\alpha \mapsto \pi_{m+1} \to \ldots \to \pi_s \to b\}$. Then we have $S'(tgt_h(\rho)) = S'(\alpha) = \pi_{m+1} \to \ldots \to \pi_s \to b \leq_{\mathbf{A}} \tau$.

$a \notin \mathbb{V}$: We have $\overline{S(\rho)} = S(\rho_1) \to \ldots \to S(\rho_r) \to a$. Thus, $m \leq r$ and $tgt_m(\pi) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to a \leq_{\mathbf{A}} \tau$. Choosing $h = m$ and $S' = S$, we get $S'(tgt_h(\rho)) = S(tgt_m(\rho)) = S(\rho_{m+1} \to \ldots \to \rho_r \to a) = S(\rho_{m+1}) \to \ldots \to S(\rho_r) \to a \leq_{\mathbf{A}} \tau$.

It is clear that $l(S') \leq l(S)$ and that the statement about the constants in the image of S' holds. □

The following lemma formalizes the necessary condition for a type τ to be inhabited:

Lemma 36 *Assume $\Gamma \vdash_k e : \tau$, where τ is a path and all types in Γ are organized.*

There exist $(y : \rho) \in \Gamma$, a path ρ' in ρ, $S' \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $h \leq \|\rho'\|$ such that $S'(tgt_h(\rho')) \leq_{\mathbf{A}} \tau$.

Proof: We write $e = x\ e_1 \ldots e_m$ and we use the only-if direction of Corollary 34 to conclude that there exists an $(x : \sigma) \in \Gamma$, $j \in J$, $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $\pi \in \mathbb{P}_m(\overline{S(\sigma_j)})$ with $tgt_m(\pi) \leq_{\mathbf{A}} \tau$. Thus, setting $(y : \rho) = (x : \sigma)$, we know that there exist $(y : \rho) \in \Gamma$, a path ρ' in ρ (note that $\rho' = \sigma_j$), $S \in \mathcal{S}_x^{(\Gamma,\tau,k)}$, and $\pi \in \mathbb{P}_m(\overline{S(\rho')})$ with $tgt_m(\pi) \leq_{\mathbf{A}} \tau$. Applying Lemma 35 we get the statement of the lemma. □

Correctness of the ATM in Fig. 4 follows because it is a direct implementation of the condition of Corollary 34 and combines it with the necessary condition stated in Lemma 36. Note that in line 15 we incorporate the check of the necessary condition and of $tgt_n(\pi_i) \leq_{\mathbf{A}} \tau_i$ (condition 2.a) of Corollary 34) into the choice of π_i in Alg. 4. Note that this makes the **if**-block in line 18 and the corresponding FAIL-statement in line 26 of Alg. 3 obsolete.

Algorithm 4 INH3(Γ, τ, k)

Input: Γ, τ, k — *wlog:* All types in Γ and $\tau = \bigcap_{i\in I} \tau_i$ are organized
Output: INH3 accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$

loop:
CHOOSE $(x : \sigma) \in \Gamma$;
write $\sigma = \bigcap_{j\in J} \sigma_j$
for all $i \in I, j \in J, n \leq \|\sigma\| + k$ **do**
  $\mathrm{candidates}(i, j, n) := \mathrm{Match}(tgt_n(\sigma_j) \leq \tau_i)$
end for
$N := \{n \leq \|\sigma\| + k \mid \forall_{i\in I} \exists_{j\in J} : \mathrm{candidates}(i, j, n) = \textbf{true}\}$
CHOOSE $n \in N$;
for all $i \in I$ **do**
  CHOOSE $j_i \in J$ with $\mathrm{candidates}(i, j_i, n) = \textbf{true}$
  CHOOSE $S_i \in \mathcal{S}_x^{(\Gamma,\tau_i,k)}$
  CHOOSE $\pi_i \in \mathbb{P}_n(\overline{S_i(\sigma_{j_i})})$ such that $tgt_n(\pi_i) \leq_{\mathbf{A}} \tau_i$ and for all $1 \leq l \leq n$
  and all $\pi' \in \overline{arg_l(\pi_i)}$ there exists $(x : \rho) \in \Gamma$, a path ρ' in ρ, and m such
  that $\mathrm{Match}(tgt_m(\rho') \leq \pi') = \textbf{true}$
end for

if $n = 0$ **then**
  ACCEPT;
else
  FORALL $(l = 1 \ldots n)$ $\tau := \bigcap_{i\in I} \overline{arg_l(\pi_i)}$;
  GOTO *loop*;
end if

5.3 IMPLEMENTATION AND EXAMPLE

We implemented our inhabitation algorithms in the .NET-framework (C# and F#) for BCL_0 and conducted experiments. We briefly discuss the results by means of a few examples. These results, in particular, illustrate the impact of the lookahead-strategy.

Example 37 *We discuss an example, first, which will then be generalized in order to compare the mean execution times of the two previous algorithms.*

1. *We consider a small repository for synthesizing functions in the ring $\mathbb{Z}_4$. It contains the identity function and the successor as well as the predecessor functions in $\mathbb{Z}_4$. Furthermore, there is a composition combinator c that computes the composition of three functions. All functions are coded by an intersection type that basically lists their function table. The type constant f denotes*

that the corresponding combinator is a function. It is not strictly necessary, however, here we introduce it to avoid self-applications of c.

$$\begin{aligned}\Gamma = \{&id : f \cap (0 \to 0) \cap (1 \to 1) \cap (2 \to 2) \cap (3 \to 3),\\ &succ : f \cap (0 \to 1) \cap (1 \to 2) \cap (2 \to 3) \cap (3 \to 0),\\ &pred : f \cap (0 \to 3) \cap (1 \to 0) \cap (2 \to 1) \cap (3 \to 2),\\ &c : \big(f \cap (\alpha \to \beta)\big) \to \big(f \cap (\beta \to \gamma)\big) \to \big(f \cap (\gamma \to \delta)\big) \to (\alpha \to \delta)\}\end{aligned}$$

The implementation of Alg. 4 solved the inhabitation question

$$\Gamma \vdash_0 ? \ : \ (0 \to 2) \cap (1 \to 3) \cap (2 \to 0) \cap (3 \to 1),$$

i.e., it synthesized functions realizing the addition of 2 *over* $\mathbb{Z}_4$*, in less than two seconds on a computer with a quad core 2.0 GHz CPU and 8 GB RAM. The implementation produces all six inhabitants which are:*

a) $c(id, succ, succ)$

b) $c(succ, id, succ)$

c) $c(succ, succ, id)$

d) $c(pred, pred, id)$

e) $c(id, pred, pred)$

f) $c(pred, id, pred)$

Figure 5.1 depicts the graphical output produced by our implementation applied to this example, enumerating the six inhabitants. The inner nodes represent functional combinators whose children are the arguments. Leaves are 0-ary combinators, which, in this example, are the three functions id, succ, and pred.

We estimate the number of new inhabitation questions Alg. 3 would have to generate: Ignoring the type constant f *for simplicity, a level-0 substitution can map a variable into* $2^4 - 1$ *types (every nonempty subset of* $\{0, 1, 2, 3\}$ *represents an intersection). Thus, there are* $15^4 = 50\,625$ *substitutions. In lines 12–16 such a substitution has to be chosen four times. This results in at least* $(15^4)^4 \approx 6.6 \times 10^{18}$ *new inhabitation goals. Even for this rather small example the* 2-EXPTIME*-bound makes Alg. 1 infeasible.*

It is easy to see that many of the 50 625 *possible substitutions will not help to inhabit* $(0 \to 2) \cap (1 \to 3) \cap (2 \to 0) \cap (3 \to 1)$*. For example, trying to inhabit the component* $(0 \to 2)$*, no instantiation of c where* $\alpha \mapsto 0$ *and* $\delta \mapsto 2 \cap \tau_0$ *where* τ_0 *is any level-0 type constructed from* 0, 1, *and* 3*, does not hold will fail. This check is incorporated by the first condition of the choice in line 15 of Alg. 4. It reduces the possible number of* 50 625 *solutions. However, the lookahead strategy checking the arguments and therefore eliminating infeasible substitutions in advance has the greatest impact because it greatly reduces the combinations of these substitutions that have to be considered. In total, an implementation of Alg. 4 constructed* 3889 *inhabitation questions to synthesize all solutions to the inhabitation question above.*

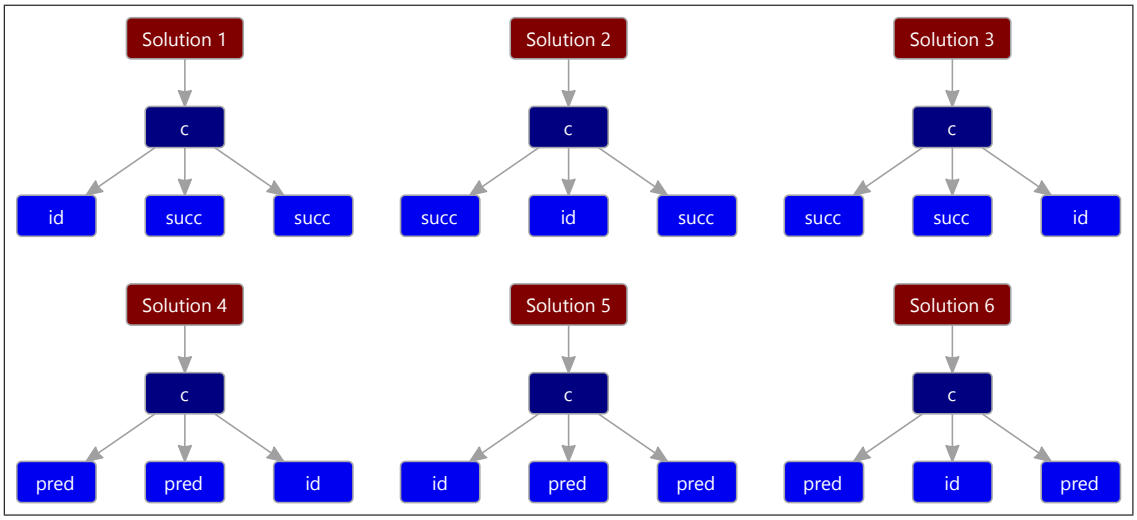

Figure 5.1: Solutions generated by the implementation of Alg. 4

2. *We generalize the previous example by defining the following parametrized type environment:*

$$\begin{aligned}\Gamma_n^m := \{&id : f \cap (0 \to 0) \cap \ldots \cap \big((n-1) \to (n-1)\big),\\ &succ : f \cap (0 \to 1) \cap \ldots \cap \big((n-1) \to 0\big),\\ &pred : f \cap \big(0 \to (n-1)\big) \cap \ldots \cap \big((n-1) \to (n-2)\big),\\ &c_m : \big(f \cap (\alpha_0 \to \alpha_1)\big) \to \ldots \to \big(f \cap (\alpha_{m-1} \to \alpha_m)\big) \to (\alpha_0 \to \alpha_m)\}\end{aligned}$$

We ask the following inhabitation question (this time, synthesizing addition of 2 *in* $\mathbb{Z}_n$*):*

$$\Gamma_n^m \vdash_0 ? \; : \; (0 \to 2) \cap (1 \to 3) \cap \ldots \cap \big((n-1) \to 1\big)$$

Table 5.1 compares the number of new inhabitation goals (#ig) to be generated as well as the mean execution time ($\overline{ET}$) and the standard deviation (sd) of Alg.(s) 3 and 4 for some values of n and m. We aggregated the information over a sample of four. In some cases, as can be seen by the foregoing discussion, we can only estimate the corresponding numbers for Alg. 3, because it is not possible to wait for the result. The corresponding entries marked by an asterix are estimates for the least number of inhabitation goals to be generated respectively for the mean execution time.

A few comments about the figures are to be made. One might ask why Alg. 4 is slower than Alg. 3 for $n = m = 2$ *even though the number of inhabitation goals to be generated is much smaller. This can be explained by the fact that the lookahead-optimization itself requires some computational effort which for very small examples may be significant. However, with increasing values for n and m the improvement is obvious. Furthermore, the estimated figures are only very rough lower bounds. First,*

n	m	Algorithm	#ig	$\overline{\text{ET}}$/ms	sd/ms
2	2	3	73	84.25	0.375
2	2	4	9	96.75	4.25
3	2	3	43 905	29 631	127.5
3	2	4	55	121	1
3	3	3	$4 \times 10^{7*}$	$5.9 \times 10^{7*}$	-
3	3	4	2188	364	8.5
4	2	3	$1.3 \times 10^{14*}$	$1.9 \times 10^{14*}$	-
4	2	4	33	197	0.5
4	3	3	$6.6 \times 10^{18*}$	$9.8 \times 10^{18*}$	-
4	3	4	3889	2270	12.5

Table 5.1: Experimental results for Γ_n^m

we only estimated the inhabitation goals that have to be generated in the very first step. Second, for the execution time we only used a linear model to estimate the execution time required for one inhabitation goal. This assumption is not very realistic, because it should be expected that the execution time per goal increases exponentially with larger values for n and m.

We would like to point out that even for this rather small example, the figures illustrate the explosiveness of the 2-EXPTIME*-complexity of the algorithms.*

6 CONCLUSION

This technical report contains the detailed proofs accompanying the paper of the same title. We provide an NP-completeness proof for the intersection type matching problem. Amongst others we use this result to incrementally formulate various optimizations for the ATM deciding inhabitation in BCL_k that was presented in [6].

Future work includes more specific optimizations and more experiments. For example, our experiments suggest that any optimization that reduces the number of substitutions that have to be generated can have a great impact (the earlier in the algorithm this number can be reduced the better). For example, considering multistep-lookahead (looking several steps ahead) might further improve the runtime of the algorithm for many practical applications. Of independent theoretical interest is satisfiability over $\leq_{\mathbf{A}}$.

BIBLIOGRAPHY

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

[2] Chandra, A. K., Kozen, D. C., and Stockmeyer, L. J. Alternation. *Journal of the ACM 28*, 1 (1981), 114–133.

[3] Coppo, M., and Dezani-Ciancaglini, M. An extension of basic functionality theory for lambda-calculus. *Notre Dame Journal of Formal Logic 21* (1980), 685–693.

[4] Dezani-Ciancaglini, M., and Hindley, R. Intersection Types for Combinatory Logic. *Theoretical Computer Science 100*, 2 (1992), 303–324.

[5] Düdder, B., Garbe, O., Martens, M., Rehof, J., and Urzyczyn, P. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for GUI Synthesis. In *Proceedings of ITRS'12* (2012).

[6] Düdder, B., Martens, M., Rehof, J., and Urzyczyn, P. Bounded Combinatory Logic. In *Proceedings of CSL'12* (2012), vol. 16 of *LIPIcs*, Schloss Dagstuhl, pp. 243–258.

[7] Düdder, B., Martens, M., Rehof, J., and Urzyczyn, P. Bounded Combinatory Logic (Extended Version). Tech. Rep. 840, Faculty of Computer Science (TU Dortmund), 2012. http://ls14-www.cs.tu-dortmund.de/index.php/Jakob_Rehof_Publications#Technical_Reports.

[8] Düdder, B., Martens, M., Rehof, J., and Urzyczyn, P. Using Inhabitation in Bounded Combinatory Logic with Intersection Types for Synthesis. Tech. Rep. 842, Faculty of Computer Science (TU Dortmund), 2012. http://ls14-www.cs.tu-dortmund.de/index.php/Jakob_Rehof_Publications#Technical_Reports.

[9] Frey, A. Satisfying Subtype Inequalities in Polynomial Space. *Theor. Comput. Sci. 277*, 1-2 (2002), 105–117.

[10] Hindley, J. R. The Simple Semantics for Coppo-Dezani-Sallé Types. In *International Symposium on Programming* (1982), M. Dezani-Ciancaglini and U. Montanari, Eds., vol. 137 of *LNCS*, Springer, pp. 212–226.

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

[12] Niehren, J., Priesnitz, T., and Su, Z. Complexity of Subtype Satisfiability over Posets. In *Proceedings of ESOP'05* (2005), vol. 3444 of *LNCS*, Springer, pp. 357–373.

[13] Papadimitriou, C. H. *Computational Complexity*. Addison-Wesley, 1994.

[14] Rehof, J. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, Department of Computer Science, 1998. http://ls14-www.cs.tu-dortmund.de/images/a/ac/ComplexityOfSimpleSubtypingSystems.pdf.

[15] Rehof, J., and Urzyczyn, P. Finite Combinatory Logic with Intersection Types. In *Proceedings of TLCA'11* (2011), vol. 6690 of *LNCS*, Springer, pp. 169–183.

[16] Statman, R. Intuitionistic Propositional Logic Is Polynomial-space Complete. *Theoretical Computer Science 9* (1979), 67–72.

[17] Tiuryn, J. Subtype Inequalities. In *Proceedings of LICS'92* (1992), IEEE Computer Society, pp. 308–315.

[18] Urzyczyn, P. The Emptiness Problem for Intersection Types. *Journal of Symbolic Logic 64*, 3 (1999), 1195–1215.

Forschungsberichte
der Fakultät für Informatik
der Technischen Universität Dortmund

ISSN 0933-6192

Anforderungen an:
Dekanat Informatik | TU Dortmund
D-44221 Dortmund