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
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 $\tau$ and $\sigma$, where $\sigma$ is a constant type, does there exists a type substitution $S$ such that $S(\tau)$ is a subtype of $\sigma$? 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 7

2 Preliminaries 9  
   2.1 Intersection Types 9  
   2.2 Alternating Turing Machines 10

3 Bounded Combinatory Logic 13  
   3.1 Type System 13  
   3.2 Inhabitation 14

4 Intersection Type Matching 19  
   4.1 Lower Bound 19  
   4.2 Upper Bound 22

5 Matching-based Optimizations 37  
   5.1 Matching Optimization 37  
   5.2 Matching Optimization Using Lookahead 39  
   5.3 Implementation and Example 40

6 Conclusion 45

BIBLIOGRAPHY 45
INTRODUCTION

Bounded combinatory logic with intersection types (bclk 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 Γ ⊢^k e : τ?

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 bclk 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 S, 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 bclk (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 bclk 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 bclk 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(τ) ≤^A σ?
The relation $\leq_A$ 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 $\leq_A$. In [15] it was shown that $\leq_A$ 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.
2.1 Intersection types

Type expressions, ranged over by $\tau, \sigma$, etc., are defined by $\tau ::= a | \tau \to \tau | \tau \cap \tau$ where $a, b, c, \ldots$ range over atoms comprising of type constants, drawn from a finite set including the constant $\omega$, and type variables, drawn from a disjoint denumerable set $V$ ranging over by $a, \beta$ 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 $\Gamma$ is a finite function from term variables to types, written as a finite set of type assumptions of the form $(x : \tau)$. We let $\text{Var}(\tau)$ and $\text{Var}(\Gamma)$ denote the sets of type variables occurring in a type $\tau$ respectively in $\Gamma$.

A type $\tau \cap \sigma$ is said to have $\tau$ and $\sigma$ 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 $\omega$.

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

$$
\begin{align*}
\sigma &\leq_A \omega, \quad \omega \leq_A \omega \to \omega, \quad \sigma \cap \tau \leq_A \sigma, \quad \sigma \cap \tau \leq_A \tau, \quad \sigma \leq_A \sigma \cap \sigma; \\
(\sigma \to \tau) \cap (\sigma \to \rho) &\leq_A \sigma \to \tau \cap \rho; \\
\text{If } \sigma &\leq_A \sigma' \text{ and } \tau \leq_A \tau' \text{ then } \sigma \cap \tau \leq_A \sigma' \cap \tau' \text{ and } \sigma' \to \tau \leq_A \sigma \to \tau'.
\end{align*}
$$

We identify $\sigma$ and $\tau$ when $\sigma \leq_A \tau$ and $\tau \leq_A \sigma$. The distributivity properties $(\sigma \to \tau) \cap (\sigma \to \rho) = \sigma \to (\tau \cap \rho)$ and $(\sigma \to \tau) \cap (\sigma' \to \tau') \leq_A (\sigma \cap \sigma') \to (\tau \cap \tau')$ follow from the axioms of subtyping. Note also that $\tau_1 \to \cdots \to \tau_m \to \omega = \omega$. We say that a type $\tau$ is reduced with respect to $\omega$ 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 $\omega$, 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 $\sigma = \text{tgt}_{\rho} (\tau)$ and $\tau_i = \text{arg}_{\rho} (\tau)$, 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 $\tau$ is organized if it is a (possibly empty) intersection of paths (those are called paths in $\tau$). Every type $\tau$ 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 $\tau = \bigcap_{i \in I} \tau_i$ then take $\overline{\tau} \to \sigma = \bigcap_{i \in I} (\overline{\tau} \to \sigma_i)$. 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 \( \sigma \), we let \( P_m(\sigma) \) denote the set of all paths in \( \sigma \) of length \( m \) or more. We extend the definition to arbitrary \( \tau \) by implicitly organizing \( \tau \), i.e., we write \( P_m(\tau) \) as a shorthand for \( P_m(\tau) \). The path length of a type \( \tau \) is denoted \( ||\tau|| \) and is defined to be the maximal length of a path in \( \tau \).

A substitution is a function \( S : V \to T \), such that \( S \) is the identity everywhere but on a finite subset of \( V \). For a substitution \( S \), we define the support of \( S \), written \( \text{Supp}(S) \), as \( \text{Supp}(S) = \{ a \in V \mid a \not\in S(\alpha) \} \). We may write \( S : V \to T \) when \( V \) is a finite subset of \( V \) with \( \text{Supp}(S) \subseteq V \). A substitution \( S \) is tacitly lifted to a function on types, \( S : T \to T \), 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 \( \bigcap_{i \in I} (\sigma_i \to \tau_i) \cap \bigcap_{j \in J} a_j \leq A a \) then \( a = a_j \), for some \( j \in J \).
2. If \( \bigcap_{i \in I} (\sigma_i \to \tau_i) \cap \bigcap_{j \in J} a_j \leq A \sigma \to \tau \), where \( \sigma \to \tau \neq \omega \), then the set \( \{ i \in I \mid \sigma \leq A \sigma_i \} \) is nonempty and \( \bigcap \{ \tau_i \mid \sigma \leq A \sigma_i \} \leq A \tau \).

### 2.2 Alternating Turing Machines

An alternating Turing machine (ATM) [2] is a tuple \( M = (\Sigma, Q, q_0, q_a, q_r, \Delta) \). The set of states \( Q = Q_\exists \cup Q_\forall \) 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 \( \Sigma = \{ 0, 1, \omega \} \), where \( \omega \) is the blank symbol (used to initialize the tape but not written by the machine). The transition relation \( \Delta \) satisfies

\[
\Delta \subseteq \Sigma \times Q \times \Sigma \times Q \times \{ \text{l, r} \},
\]

where \( h \in \{ \text{l, r} \} \) 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 \( M \) 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 \( \Delta \). 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:"
• An accepting configuration is eventually accepting.
• If $C$ is existential and some successor of $C$ is eventually accepting then so is $C$.
• If $C$ is universal and all successors of $C$ are eventually accepting then so is $C$.

We introduce the following notation for existential and universal states. A command of the form \texttt{choose} $x \in S$ branches from an existential state to successor states in which $x$ gets assigned distinct elements of $S$ (it implicitly rejects if $S = \emptyset$). A command of the form \texttt{forall}(i = 1 \ldots k) S_i$ branches from a universal state to successor states from which each instruction sequence $S_i$ is executed.
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, \( \text{bcl}_k \) 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. 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 \( \tau \), define its level \( \ell(\tau) \) by \( \ell(a) = 0 \) for \( a \in \cup V \), \( \ell(\tau \to \sigma) = 1 + \max\{\ell(\tau), \ell(\sigma)\} \), and \( \ell(\cap_{i=1}^n \tau_i) = \max\{\ell(\tau_i) \mid i = 1, \ldots, n\} \). Now define \( \ell(S) = \max\{\ell(S(\alpha)) \mid \alpha \in V\} \). Notice that the level of a type is independent of the number of components in an intersection.

\[
\begin{align*}
\Gamma, x : \tau & \vdash_k x : S(\tau) \\
\ell(S) \leq k & \quad \text{(var)} \\
\Gamma & \vdash_k (e e') : \tau' \\
\Gamma & \vdash_k (e) : \tau' \quad \text{\(-E\)} \\
\Gamma & \vdash_k (\tau \to \tau') \\
\Gamma & \vdash_k (\tau_1 \cap \tau_2) \\
\Gamma & \vdash_k (\tau_1) \\
\Gamma & \vdash_k (\tau_2) \\
\Gamma & \vdash_k (\tau \leq A \tau') \\
\Gamma & \vdash_k (\tau) \\
\end{align*}
\]

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

A level-\( k \) type is a type \( \tau \) 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 \( T_k \) denote the set of all level-\( k \) types. For a subset \( A \) of atomic types, we let \( T_k(A) \) 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 \( \Gamma \) and \( \tau \), is there an applicative term \( e \) such that \( \Gamma \vdash e : \tau \)?

The cause of the exponentially growing complexity of inhabitation in \( \text{bcl}_k \) (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 \( S : \text{Var}(\Gamma) \to \text{Var}(\Lambda_{\omega}(\Gamma, \tau)) \), where \( \Lambda_{\omega}(\Gamma, \tau) \) denotes the set of atoms occurring in \( \Gamma \) or \( \tau \), together with \( \omega \). This finitizes the inhabitation problem and immediately leads to decidability. Now, given a number \( k \), an environment \( \Gamma \) and a type \( \tau \), define for each \( x \) occurring in \( \Gamma \) the set of substitutions \( S^x_k(\Gamma, \tau, k) = \text{Var}(\Gamma(x)) \to \text{Var}(\Lambda_k(\Gamma, \tau)) \). 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 \( \text{bcl}_k \) 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 \( S^x_k(\Gamma, \tau, k) \) 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 \( \sigma = \bigcap_{i \in I} \sigma_i \), the size of the index set \( I \) can be bounded by the size of the index set \( J \) of \( \sigma \). 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 \((a \to b) \cap (a \to c) \leq a \to (b \cap c)\), 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 \( \tau_i \) are paths and let \( \sigma = \beta_1 \to \ldots \to \beta_n \to p \) where \( p \neq \omega \) is an atom.

We have \( \tau \leq \Lambda \sigma \) if and only if there is an \( i \in I \) with \( \tau_i = \alpha_1 \to \ldots \to \alpha_n \to p \) and \( \beta_j \leq \Lambda \alpha_j \) for all \( j \leq n \).

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

\( \Rightarrow \): We use induction over \( n \).

\( n = 0 \): We have \( \bigcap_{i \in I} \tau_i \leq \Lambda p \) 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 \( \bigcap_{i \in I} \tau_i \leq \Lambda \beta_1 \to \ldots \to \beta_{n+1} \to p \). Lem. 1 further states that the set \( H = \{ k \in K | \beta_1 \leq A_k \sigma_k \} \) is non-empty and \( \bigcap_{h \in H} c_k' \leq \Lambda \beta_2 \to \ldots \to \beta_{n+1} \to p \). Note that each of the \( c_k' \) 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 \( \sigma_k'_{h_0} = a_2 \to \ldots \to a_{n+1} \to p \) where \( \beta_l \leq \Lambda a_l \) for all \( 2 \leq l \leq n + 1 \). Because \( h_0 \in H \) we
Algorithm 0 Alternating Turing machine deciding inhabitation in $\mathsf{bcl}_k$

1: Input: $\Gamma, \tau, k$
2: Output: $\text{INH1}$ accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3: 4: loop:
5: \# CHOOSE $(x : \sigma) \in \Gamma$;
6: \# $\sigma' := \bigcap \{S(\sigma) \mid S \in S^\Gamma(\tau, k)\}$;
7: \# CHOOSE $n \in \{0, \ldots \|\sigma'\|\}$;
8: \# CHOOSE $P \subseteq P_n(\sigma')$;
9: if $\bigcap_{\pi \in P} \text{tgt}_n(\pi) \leq A \tau$ then
10: \quad if $n = 0$ then
11: \quad \quad ACCEPT;
12: \quad else
13: \quad \quad FORALL $(j = 1 \ldots n) \tau := \bigcap_{\pi \in P} \text{arg}_j(\tau)$;
14: \quad \quad GOTO loop;
15: \quad end if
16: \else
17: \quad FAIL;
18: end if
19: end if

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

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

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

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

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

$\square$

For $\sigma = \bigcap_{j \in J} \sigma_j$ (not necessarily organized), it is easy to see that one has $\tau \leq_A \sigma$ iff for all $j$ we have $\tau \leq_A \sigma_j$. 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 \( \omega \). We have \( \tau \leq_A \sigma \) if and only if there exists \( I' \subseteq I \) with \( |I'| \leq |J| \) and \( \bigcap_{i \in I'} \tau_i \leq_A \sigma \).

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

Assume \( \tau \leq_A \sigma \). This implies \( \tau \leq_A \sigma_j \) for all \( j \in J \). Fix \( j \in J \). \( \sigma \) is organized and we write \( \sigma_j = \beta_j^1 \rightarrow \ldots \rightarrow \beta_j^{n_j} \rightarrow P_j \). By the “if”-part of Lem. 2 there is an index \( i_j \in I \) such that \( \tau_j = a_j^1 \rightarrow \ldots \rightarrow a_j^{i_j} \rightarrow P_j \) with \( \beta_j^k \leq_A a_j^k \) 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 \( \bigcap_{i \in I'} \tau_i \leq_A \sigma_j \).

As shown in [6], the key to an algorithm matching the lower bound for \( \text{bcl}_k \) is a path lemma ([6, Lemma 11]) which characterizes inhabitation by the existence of certain sets of paths in instances of types in \( \Gamma \). 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. \( \Gamma \vdash_k x_1 \ldots x_m : \tau \)

2. There exists a set \( P \) of paths in \( P_m(\bigcap\{S(\sigma) \mid S \in S_x^{(\Gamma, \tau, k)}\}) \) such that
   a) \( \bigcap_{\pi \in P} t_{g_m}^m(\pi) \leq_A \tau \);
   b) \( \Gamma \vdash_k e_i : \bigcap_{\pi \in P} \arg_i(\pi) \), for all \( i \leq m \).

3. There exists a set \( \mathcal{S} \subseteq S_x^{(\Gamma, \tau, k)} \) of substitutions with \( |\mathcal{S}| \leq |I| \) and a set \( \mathcal{P}' \subseteq P_m(\bigcap_{S \in \mathcal{S}} S(\sigma)) \) of paths with \( |\mathcal{P}'| \leq |I| \) such that
   a) \( \bigcap_{\pi \in \mathcal{P}'} t_{g_m}^m(\pi) \leq_A \tau \);
   b) \( \Gamma \vdash_k e_i : \bigcap_{\pi \in \mathcal{P}'} \arg_i(\pi) \), for all \( i \leq m \).

Proof: The implication 1. \( \Rightarrow \) 2. follows from Lem. 11 in [7].

We prove 2. \( \Rightarrow \) 3 : Let \( P \) be as in the condition, i.e., \( \bigcap_{\pi \in P} t_{g_m}^m(\pi) \leq_A \tau \). Lemma 3 states that there is \( \mathcal{P}' \subseteq P \) with \( |\mathcal{P}'| \leq |I| \) and \( \bigcap_{\pi \in \mathcal{P}'} t_{g_m}^m(\pi) \leq_A \tau \). For each \( \pi \in \mathcal{P}' \) there exists \( \pi_0 \in S_x^{(\Gamma, \tau, k)} \) such that \( \pi \in P_m(S_0(\sigma)) \). Define \( \mathcal{S} = \{S_0 | \pi \in \mathcal{P}' \} \). It is clear that \( |\mathcal{S}| \leq |I| \) and that \( \mathcal{P}' \subseteq P_m(\bigcap_{S \in \mathcal{S}} S(\sigma)) \). Thus, 3.(a) holds. Fix \( i \leq m \). Because \( \mathcal{P}' \subseteq P \) we have \( \bigcap_{\pi \in \mathcal{P}'} \arg_i(\pi) \leq_A \bigcap_{\pi \in \mathcal{P}'} \arg_i(\pi) \). Since we have \( \Gamma \vdash_k e_i : \bigcap_{\pi \in \mathcal{P}'} \arg_i(\pi) \), rule \( \leq_A \) yields \( \Gamma \vdash_k e_i : \bigcap_{\pi \in \mathcal{P}'} \arg_i(\pi) \).

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

2. There exists a set $S \subseteq S_{\Gamma,x}^{(\tau,k)}$ of substitutions with $|S| \leq |I|$ and a set $P \subseteq P_m(\bigcap_{S \in S} S(\sigma))$ of paths with $|P| \leq |I|$ such that
   a) $\bigcap_{\pi \in P} \mathit{tgt}_m(\pi) \leq_A \tau$;
   b) $\Gamma \vdash_k e_j : \bigcap_{\pi \in P} \mathit{arg}_j(\pi)$, for all $j \leq m$.

Algorithm 1 below is a direct implementation of the path lemma (Cor. 5) and therefore decides inhabitation in $\mathsf{bcl}_k$.

Algorithm 1 $\text{INH1}(\Gamma, \tau, k)$

1: Input: $\Gamma, \tau, k \: \text{wlog: All types in } \Gamma \text{ and } \tau = \bigcap_{i \in I} \tau_i \text{ are organized}$
2: Output: $\text{INH1}$ accepts iff $\exists e$ such that $\Gamma \vdash_k e : \tau$
3:
4: loop:
5: \text{CHOOSE} $(x : \sigma) \in \Gamma$;
6: \text{CHOOSE} $S \subseteq S_{\Gamma,x}^{(\tau,k)}$ with $|S| \leq |I|$;
7: $\sigma' := \bigcap \{S(\sigma) | S \in S\}$;
8: \text{CHOOSE} $n \in \{0, \ldots |\sigma'|\}$;
9: \text{CHOOSE} $P \subseteq P_n(\sigma')$ with $|P| \leq |I|$;
10: if $\bigcap_{\pi \in P} \mathit{tgt}_m(\pi) \leq_A \tau$ then
11: \text{ if } $n = 0$ then
12: ACCEPT;
13: else
14: FORALL $(j = 1 \ldots n)$ $\tau := \bigcap_{\pi \in P} \mathit{arg}_j(\pi)$;
15: GOTO loop;
16: end if
17: else
18: FAIL;
19: end if

Algorithm 0 is identical to Alg. 1 but for the fact that it ignores the restrictions $|S| \leq |I|$ (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).
The work done in lines 5 through 9 of Alg. 1 aims at constructing paths $\pi$ descending from $\sigma$ (using instantiating substitutions) such that the condition $\bigcap_{\pi \in P} \text{tgt}_\mu(\pi) \leq_A \tau$ in line 11 is satisfied. Clearly, it would be an important optimizing heuristic if we could rule out uninteresting paths $\pi$ 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 $\sigma$ very early on, i.e., right after choosing $\sigma$ 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 $\tau$ and $\sigma$ where $\sigma$ does not contain any type variables, is there a substitution $S : V \rightarrow T$ with $S(\tau) \leq_A S(\sigma)$?

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 $\sigma_i$ or $\tau_i$ does not contain any type variables. We say that $C$ is matchable if there is a substitution $S : V \rightarrow T$ such that for all $i$ we have $S(\tau_i) \leq_A S(\sigma_i)$. We say that $S$ matches $C$.

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

We sometimes denote $\text{CMATCH}$ and $\text{cMATCH}$ as matching problems. Furthermore, we write $S(\sigma) \leq_A S(\tau)$ 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 $\tau \leq_A \sigma$ states that $\tau$ is a subtype of $\sigma$.

### 4.1 Lower Bound

In this section we will prove the following theorem:

**Theorem 7** $\text{CMATCH}$ is NP-hard.

**Proof:** We define a reduction $R$ from 3SAT to $\text{CMATCH}$ such that for any formula $F$ in 3-CNF we have: $F$ is satisfiable iff $R(F)$ is matchable.

Let $F = c_1 \land \ldots \land c_m$ where for each $i$ we have $c_i = L_1^i \lor L_2^i \lor L_3^i$ and each $L_j^i$ 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 \( \alpha_x \) and \( \alpha_{\neg x} \). Furthermore, we assume the two type constants 1 and 0. For a given formula \( F \) we obtain the set of constraints \( R(F) \) which contains the following constraints:

- For all \( x \) in \( F \): \(( (1 \rightarrow 1) \cap (0 \rightarrow 0) \) \) \( \leq \) \(( \alpha_x \rightarrow \alpha_x \) \) \( \rightarrow \) \( \alpha_x \)
- For all \( x \) in \( F \): \(( (1 \rightarrow 1) \cap (0 \rightarrow 0) \) \) \( \leq \) \(( \alpha_{\neg x} \rightarrow \alpha_{\neg x} \) \) \( \rightarrow \) \( \alpha_{\neg x} \)
- For all \( x \) in \( F \): \(( 1 \rightarrow 0 \) \) \( \cap \) \(( 0 \rightarrow 1 \) \) \( \leq \) \(( \alpha_x \rightarrow \alpha_{\neg x} \) \)
- For all \( c_i: \alpha_{L_{i1}} \cap \alpha_{L_{i2}} \cap \alpha_{L_{i3}} \leq 1 \)

The first two groups of constraints ensure that every type variable (i.e., \( \alpha_x \) or \( \alpha_{\neg x} \)) can only be instantiated by 1 or 0 (and, in fact, by \( \omega \) — but this does not pose a problem, as we will see). The third group of constraints ensures that for every type variable \( \alpha_x \) the type variable \( \alpha_{\neg x} \) must be assigned the opposite value. The last group of constraints ensures that for every clause of \( F \) at least one type variable is instantiated to 1.

The correctness of the reduction is shown in Lem. 8, below.

\[ \square \]

**Lemma 8** Let \( R \) be defined as above and let \( F \) be a 3SAT formula. The following equivalence holds.

\[ F \text{ satisfiable} \iff R(F) \text{ matchable} \]

**Proof:** For the “only if”-direction let \( v \) be a valuation that satisfies \( F \). We define a substitution \( S_v: \mathbb{V} \rightarrow \mathbb{T} \) as follows:

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

By way of slight notational imprecision the right hand sides of these defining equations consider the truth values \( v(x) \) and \( \neg v(x) \) as types. We claim that \( S_v \) matches \( R(F) \). Observe that every type variable is instantiated to either 1 or 0. The observation directly implies that the first two groups of type constraints in \( R(F) \) are clearly matched by the substitution. Furthermore, the definition of \( S_v \) implies that for any \( x \) we have \( S_v(\alpha_x) \neq S_v(\alpha_{\neg x}) \). Together with the observation above we conclude \( S_v(\alpha_x) = 1 \Rightarrow S_v(\alpha_{\neg x}) = 0 \) and \( S_v(\alpha_x) = 0 \Rightarrow S_v(\alpha_{\neg x}) = 1 \). Therefore the third group of type constraints is matched. Finally, \( v(F) = 1 \) implies that for all clauses \( c_i \) there must be a literal \( L_{i1} \) with \( v(L_{i1}) = 1 \). Because \( \alpha_{L_{i1}} \) is instantiated exactly to \( v(L_{i1}) \) this means that for every constraint in the last group there is a type variable that is instantiated to 1. Namely, we have \( S_v(\alpha_{L_{i1}}) = 1 \) and therefore every constraint in the last group of constraints is matched.
For the “if”-direction, from a substitution \( S \) which matches \( \mathcal{R}(F) \), we must construct a satisfying valuation \( v_S \) for \( F \). The idea of this construction is similar to the construction of \( v_v \) in the “only if”-direction presented above. However, for the following to be well-defined

\[
v_S(x) = S(a_x),
\]

we have to make sure that \( S \) instantiates any type-variable exactly to 1 or to 0 (any other type instantiation would not lead to a well-defined valuation \( v_S \)). However, it is easy to see that a constraint of the first group would also be satisfied if \( S(a_x) = \omega \). It is not clear how to define \( v_S(x) \) in this case. We adapt the definition of \( v_S(x) \) above, as follows:

\[
v_S(x) = \begin{cases} S(a_x) & \text{if } S(a_x) \neq \omega \\ 1 & \text{else} \end{cases}
\]

Note that the valuation \( v_S(x) = 1 \) in the else-case is arbitrary. Any variable which is not forced to be equal to either 0 or 1 can be considered to be a don’t-care for the satisfiability of \( F \). We now have to argue, that \( v_S \) is well-defined and that it satisfies \( F \):

Well-definedness of \( v_S \): We show that every propositional variable \( x \) is mapped to 0 or 1. For those variables where the else-case of the definition applies, this is clear. Otherwise, let \( x \) be a variable with \( S(a_x) \neq \omega \). Because \( S \) matches \( \mathcal{R}(F) \) we know in particular that \(((1 \rightarrow 1) \rightarrow 1) \land ((0 \rightarrow 0) \rightarrow 0) \leq A (S(a_x) \rightarrow S(a_x)) \rightarrow S(a_x)\) holds. Because \( S(a_x) \neq \omega \) we may conclude that \((S(a_x) \rightarrow S(a_x)) \rightarrow S(a_x) \neq \omega \) and therefore Lem. 1 is applicable. It states that a necessary condition for this to hold is \( S(a_x) \rightarrow S(a_x) \leq A (1 \rightarrow 1) \) or \( S(a_x) \rightarrow S(a_x) \leq A (0 \rightarrow 0) \). The first type constraint can hold if and only if \( 1 \leq A S(a_x) \) and \( S(a_x) \leq A 1 \), i.e., \( S(a_x) = 1 \). The second type constraint similarly can hold if and only if \( S(a_x) = 0 \). Thus, only one of the two conditions can be satisfied. Without loss of generality the first one holds and we have \( S(a_x) = 1 \). In this case Lem. 1 further states that \( 1 \leq A S(a_x) \) must hold which is certainly the case. We conclude that for \( a_x \) with \( S(a_x) \neq \omega \) the constraint \(((1 \rightarrow 1) \rightarrow 1) \land ((0 \rightarrow 0) \rightarrow 0) \leq (a_x \rightarrow a_x) \rightarrow a_x \) is matched by \( S \) if and only if \( S \) instantiates \( a_x \) either to 1 or to 0. We see that \( v_S(x) \in \{0,1\} \) for all propositional variables \( x \) occurring in \( F \).

Satisfaction of \( F \): We have to show that \( v_S(c_i) = 1 \) for all \( 1 \leq i \leq m \). Consider \( c_i \) and the corresponding constraint \( a_{L^i_1} \land a_{L^i_2} \land a_{L^i_3} \leq 1 \). For all \( j \in \{1,2,3\} \) we have \( S(a_{L^i_j}) \in \{0,1,\omega\} \). Note that this follows from the argument above if \( L^i_j \) is a positive literal. If \( L^i_j \) is a negative literal then there is a corresponding constraint in the second group of constraints for which an analogous argument can be made. Because \( S \) matches \( a_{L^i_1} \land a_{L^i_2} \land a_{L^i_3} \leq 1 \) we may use the statement of Lem. 1 regarding atoms, to conclude that there must be some \( 1 \leq j \leq 3 \) with \( S(a_{L^i_j}) = 1 \). If \( a_{L^i_j} \) is a positive literal, say \( a_{L^i_j} = x \), then the first case of the definition of \( v_S \) states \( v_S(x) = 1 \) and therefore also \( v_S(c_i) = 1 \).
Otherwise, let $a_{L_i} = \neg x$. If we can show that $S(a_x) = 0$ we are done because in this case we have $v_S(x) = 0$ and therefore $v_S(\neg x) = 1$ and consequently also $v_S(c_i) = 1$. We know $S(a_{L_i}) = S(a_{\neg x}) = 1$ and we consider the constraint of the third group that corresponds to $x$. It is matched by $S$ and thus we have:

$$(1 \to 0) \cap (0 \to 1) \leq_A S(a_x) \to 1$$

Considering cases, it is immediately clear that $(1 \to 0) \cap (0 \to 1) \leq_A S(a_x) \to 1$ would not hold if $S(a_x) \in \{1, \omega\}$. If $S(a_x) = 0$, however, the constraint is clearly satisfied. Thus, $S(a_x) = 0$ is forced in this case by the corresponding constraint in the third group of constraints, and we are done.

This proves the correctness of the reduction, establishing the NP-hardness of cMATCH. □

The following corollary yields a reduction from CMATCH to cMATCH:

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

**Proof:** We reduce CMATCH to cMATCH, by constructing two types $\tau$ and $\sigma$ such that $\tau$ contains type variables and $\sigma$ does not, by placing the types occurring in a constraint of an instance $C$ of CMATCH at corresponding covariant or contravariant positions (depending on which type contains variables) in the type trees of $\tau$ and $\sigma$.

It can be seen that $\tau \leq \sigma$ can be matched if and only if $C$ can be matched. Furthermore, it is clear $\tau$ and $\sigma$ can be constructed such that they are of polynomial size in the number of constraints in $C$. We conclude that 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 $\tau$ is a type variable and $\sigma$ does not contain any type variables or $\sigma$ is a type variable and $\tau$ does not contain any type variables.

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

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

$C$ 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 $\alpha$ be a variable occurring in $C$ and let $\tau_i \leq \alpha$ for $1 \leq i \leq n$ be the constraints in $C$ where $\alpha$ 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 $\alpha$ 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 $\tau_i \leq_A \sigma_j$. Because $C$ is matchable there exists a substitution $S$ such that $\tau_i \leq_A S(\alpha)$ for $1 \leq i \leq n$ and $S(\alpha) \leq_A \sigma_j$ for $1 \leq j \leq m$. By transitivity of $\leq_A$ we get $\tau_i \leq_A S(\alpha) \leq_A \sigma_j$ 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 $\alpha$ occurs on the left hand side of $\leq$. Define the substitution $S_C$ by setting $S_C(\alpha) = \alpha$, if $\alpha$ does not occur in $C$, and $S_C(\alpha) = \bigcap_{j=1}^{m} \sigma_j$ otherwise. Notice that the basic constraints in $C$ do not contain any variables on one side of $\leq$. If $C$ is consistent then we have $\tau_i \leq_A \sigma_j$ for all $i$ and $j$ which is equivalent to $\tau_i \leq_A \bigcap_{j=1}^{m} \sigma_j = S_C(\alpha)$ for all $i$. Thus, $S_C$ matches $C$.

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. Nondeterminism may result from the choices in lines 29, 33, and 43.

2. Failing choices return false.

3. We tacitly assume that memoization is used to make sure that the algorithm never adds the same constraint to $C$ more than once.

4. The reduction with respect to $\omega$ in line 6 means, in particular, that neither $\tau$ nor $\sigma$ contain any paths of the form $\rho_1 \rightarrow \ldots \rightarrow \rho_m \rightarrow \omega$ unless they are syntactically identical to $\omega$.

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 17 and 37 $\sigma$ is not an intersection and therefore a path.

$\square$

Recall that empty intersections equal $\omega$ (thus, $S_C$ is well-defined even if $m = 0$).
6. In order to check matchability of $\bigcap_{i \in I} \tau_i \leq \sigma$ where the $\tau_i$ and $\sigma$ are paths, it suffices to only choose one index $i_0$ in lines 33 and 43, whereas in line 29 it is necessary to choose an index set $I'$. This is illustrated by the following example: $C = \{(a \rightarrow b) \cap (a \rightarrow c) \leq a \rightarrow a, a \leq b \cap c\}$ is matchable with the substitution $a \rightarrow b \cap c$. If, in line 29, the algorithm were only allowed to choose a single index from $I'$ this would result in the addition of the new constraints $a \leq a$ and either $b \leq \alpha$ or $c \leq \alpha$, i.e. (choosing the first component of the intersection — the other case is treated analogously), the resulting set of constraints would be $\{a \leq a, b \leq a, a \leq b \cap c\}$. This set of constraints is not matchable, and it is easy to see that the algorithm would always return `false` even though the initial set of constraints was matchable.

The reason is that, if $\sigma$ is a path whose target is a variable, a substitution may cause $\sigma$ to not be a path any more. Therefore, it is not possible to use Lem. 2, and we have to choose a number of paths from $\tau$.

7. After line 19 ($\tau = \omega$) the only possibility to match $\tau \leq \sigma$ for $\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow A$ is if $A$ is a type variable. It cannot be a constant different from $\omega$ because in this case for all substitutions $S$ we have $\omega \not\leq_A S(\sigma)$. $A$ cannot be $\omega$ either, because then $\sigma$ would have been reduced to $\omega$ in line 6 and the case in line 8 would have been applicable.

---

**Algorithm 2** Matching(C)

1. Input: $C = \{\tau_1 \leq \sigma_1, \ldots, \tau_n \leq \sigma_n\}$ such that for all $i$ at most one of $\sigma_i$ and $\tau_i$ contains variables. Furthermore, all types have to be organized.

2. Output: True if $C$ can be matched otherwise false

3. while $\exists$ non-basic constraint in $C$ do

4. choose a non-basic constraint $c = (\tau \leq \sigma) \in C$

5. reduce $\tau$ and $\sigma$ with respect to $\omega$

6. switch

7. case: $c$ does not contain any variables

8. if $\tau \leq_A \sigma$ then

9. $C := C \{c\}$

10. else

11. return false

12. end if

13. case: $\sigma = \bigcap_{i \in I} \sigma_i$

14. $C := C \{c\} \cup \{\tau \leq \sigma_i | i \in I\}$

15. //Algorithm continued on p. 25

---

**Lemma 14** Algorithm 2 terminates.
case: $\sigma$ contains variables, $\tau$ does not contain any variables

write $\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow A$

if $\tau = \omega$ then

if $A \in V$ then

$C := C \backslash \{c\} \cup \{\omega \leq A\}$

else

return false
endif
endif

case: $\tau$ contains variables, $\sigma$ does not contain any variables

if $\sigma = \omega$ then

$C := C \backslash \{c\}$

else

choose $i_0 \in I_1$

$C := C \backslash \{c\} \cup \{\omega \leq \tau_{i_0,j} | 1 \leq j \leq m\} \cup \{\sigma_{m_i} \leq A_i \}

endif

end switch

end while

if $C$ is consistent then

return true
else

return false
end if
Proof: The reduction-step in line 6 reduces 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 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 \texttt{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 \( \tau_j \leq \sigma_j \) 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 \( \tau \) be a type. The set of arguments \( \arg(\tau) \) of \( \tau \) is inductively defined as follows:

\[
\arg(a) = \emptyset
\]

\[
\arg(\bigcap_{i \in I} \tau_i) = \bigcup_{i \in I} \arg(\tau_i)
\]

\[
\arg(\tau' \rightarrow \tau'') = \{ \tau' \} \cup \arg(\tau') \cup \arg(\tau'')
\]

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

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

If \( \rho \) 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 \( \rho' \) is a subterm of one of the \( \rho_i \) then we know by induction that \( \arg(\rho') \subseteq \arg(\rho_i) \subseteq \bigcup_{i \in \rho_i} \arg(\rho_i) = \arg(\rho) \). Finally, let \( \rho = \rho_1 \rightarrow \rho_2 \). If \( \rho' = \rho_i \) for \( i \in \{1, 2\} \) then it is clear that \( \arg(\rho') \subseteq \{ \rho_i \} \cup \arg(\rho_1) \cup \arg(\rho_2) = \arg(\rho) \). If \( \rho' \) is a subterm of \( \rho_i \) for \( i \in \{1, 2\} \) then an analogous induction argument as in the previous case shows that \( \arg(\rho') \subseteq \arg(\rho) \).

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

Proof: We first show by induction that for every type \( \tau \) 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 \( \arg(\tau) = \arg(\bigcap_{i \in \overline{I}} \tau_i) = \bigcup_{i \in \overline{I}} \arg(\tau_i) = \bigcup_{i \in \overline{I}} \arg(\overline{\tau_i}) = \arg(\bigcap_{i \in \overline{I}} \overline{\tau_i}) = \arg(\tau) \). If \( \tau = \tau' \rightarrow \tau'' \) where \( \overline{\tau} = \bigcap_{i \in I} \tau_i' \) we have \( \arg(\tau) = \{ \tau' \} \cup \arg(\tau') \cup \arg(\tau'') = \{ \tau' \} \cup \arg(\tau') \cup \arg(\overline{\tau'}) = \{ \tau' \} \cup \arg(\tau') \cup \arg(\bigcap_{i \in I} \tau_i') = \{ \tau' \} \cup \arg(\tau') \cup \bigcap_{i \in I} \arg(\tau_i') = \bigcup_{i \in I} \arg(\tau_i' \rightarrow \tau_i'') = \arg(\bigcap_{i \in I} \tau_i' \rightarrow \tau_i'') = \arg(\tau) \). From \( \arg(\tau' \rightarrow \tau'') = \{ \tau' \} \cup \arg(\tau') \cup \arg(\tau'') \) 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 $\rho$ 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 $\rho$ is a subterm of $\tau_i$ or $\sigma_i$ or of an organized argument of $\tau_i$ or $\sigma_i$.

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

Before the first execution of the while-loop it is clear that the property holds for every type occurring in a non-basic 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 non-basic 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 non-basic constraint $c$ is removed from and possibly some new constraints (basic and non-basic) are added to $C$. Thus, it suffices to show that for every type in the new non-basic constraints that are added the property holds. Fix $c = \tau \leq \sigma$ the non-basic constraint that is considered by the algorithm in the current execution of the while-loop. $\tau$ and $\sigma$ have the desired property. We now consider all possible cases. If it is not clear whether a new constraint is basic or non-basic we implicitly do the following arguments only for the new non-basic 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 $\tau$. Because the property held for $\sigma$ and because every $\sigma_i$ is a subterm of $\sigma$ the property also holds for every $\sigma_i$.

Line 17: The constraints added to $C$ in lines 21 and 31 are basic constraints (because $A$ is a variable) and therefore not relevant for this argument. We only have to explain why the constraints added in lines 30 and 34 have the desired property.

We start with the constraint $p_{i_0} \leq A$ that is added in line 34. $p_{i_0}$ is a subterm of $\tau_{i_0}$ which itself is a subterm of $\tau$. Thus, $p_{i_0}$ is a subterm of $\tau$. Because the property holds for $\tau$, i.e., $\tau$ 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 $\sigma$.

Consider $\sigma_j$ in one of the constraints added to $C$ in line 30. $\sigma_j$ an argument of $\sigma$ and we know that $\sigma$ is a subterm of an initial type or of an organized argument of an initial type. In the first case Lem. 16 shows that $\sigma_j$ is also an argument of this initial type. Thus, $\sigma_j$ is an organized argument of an initial type and the property holds. In the second case denote by $\overline{p}$ the organized argument of an initial type which $\sigma$ is a subterm of. Because

2 In line 15 this is the case, for example.
\( \sigma_j \) is an argument of \( \sigma \), Lem. 16 shows that it is also an argument of \( \bar{\rho} \). We may now use Lem. 17 to conclude that \( \sigma_j \) is also an argument of the initial type \( \rho \) was an argument of. Therefore, \( \sigma_j \) is an organized argument of an initial type and we are done. In order to show that the property holds for a type \( \tau_{ij} \) in one of the constraints added to \( C \) in line 30 we follow an analogous argument. Furthermore, the property also holds for the types \( \sigma_j \) and \( \tau_{ji} \) in one of the constraints added to \( C \) in line 34 with the same argument.

Line 37: No new constraint is added to \( C \) in line 39, and therefore nothing has to be argued. We only have to explain why the constraints added in lines 44 and 45 have the desired property. This, however, can be shown in a completely analogous way as above.

\[ \square \]

**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 non-basic 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 non-basic 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 non-basic constraint is considered, it is clear that the loop is iterated at most \( k^2 \) times. \[ \square \]

**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. \[ \square \]

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 \( \omega \) in line 6 can be implemented in linear time if it is done bottom-up, removing every occurrence of \( \omega \) as component in an intersection and replacing empty intersections and arrows of the form \( \rho \rightarrow \omega \) by \( \omega \). 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 non-deterministic 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). This can also 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 non-deterministic polynomial time. \( \square \)

We make some remarks about this proof:

Remark 22

1. The statement of Lem. 21 might come as a surprise since the execution of the while-loop requires a repeated organization of the arguments of the occurring types. This organization is interleaved with decomposition steps. It can be asked why this repeated organization does not result in a normalization [10] of the types involved (which could cause an exponential blowup in the size of the type). The reason why this problem does not occur is motivated by the following small example. We inductively define two families of types:

\[
\tau_0 = c_0 \cap d_0 \\
\tau_l = \tau_{l-1} \rightarrow (c_l \cap d_l) \\
\sigma_l = \sigma_{l-1} \rightarrow \alpha_l
\]

\( \tau_n \) grows exponentially if normalized. 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 \( (\tau_{n-1} \rightarrow c_n) \cap (\tau_{n-1} \rightarrow d_n) \leq \sigma_n \). In the first iteration of the while-loop the case in line 28 applies and a subset of components of the top-level intersection of \( (\tau_{n-1} \rightarrow c_n) \cap (\tau_{n-1} \rightarrow d_n) \) has to be chosen. In order to maximize the size of \( C \) we choose both components which forces the construction of the following constraints: \( c_n \cap d_n \leq \alpha_n \), \( \tau_{n-1} \leq \tau_{n-1} \) and \( \sigma_{n-1} \leq \tau_{n-1} \). 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 \( \tau_{n-1} \) is a top-level intersection) and the constraints \( \sigma_{n-1} \leq \tau_{n-2} \rightarrow c_{n-1} \) and \( \sigma_{n-1} \leq \tau_{n-2} \rightarrow d_{n-1} \) are created. For both constraints the same rule
applies and causes a change of \(C\) according to lines 44 and 45. This leads to the construction of the basic constraints \(a_{n-1} \leq c_{n-1}\) and \(a_{n-1} \leq d_{n-1}\) as well as to the construction of \(\sigma_{n-2} \leq \tau_{n-2}\) and \(\sigma_{n-2} \leq \tau_{n-2}\) and the same argument as above can be repeated.

We conclude that the doubling of the arguments of the \(\tau_i\) 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.

2. A new intersection which possibly does not occur as a subterm in any of the types occurring initially in \(C\) has to be constructed in line 31. 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 basic constraints. However, this construction of new intersections only happens as a consequence to the non-basic constraint that is treated there. As noted above there can be at most a polynomial number of non-basic 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 \(\tau\) be a type and let \(S\) be a substitution. Then \(S(\tau) = S(\tau)^3\).

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

If \(\tau\) is an atom then \(\tau = \tau\) and nothing has to be proved. If \(\tau = \tau' \cap \tau''\) we have \(S(\tau) = S(\tau') \cap S(\tau'') = S(\tau') \cap S(\tau'') = S(\tau' \cap \tau'')[3] = S(\tau)[3]\). If \(\tau = \tau' \rightarrow \tau''\) with \(\tau'' = \bigcap_{i \in I} \tau_i''\) we have \(S(\tau) = S(\tau') \rightarrow S(\tau'') = S(\tau') \rightarrow S(\tau'') = S(\tau') \rightarrow S(\bigcap_{i \in I} \tau_i'') = S(\tau') \rightarrow \bigcap_{i \in I} S(\tau_i'') = \bigcap_{i \in I}(S(\tau') \rightarrow S(\tau_i'')) = \bigcap_{i \in I}(\tau' \rightarrow \tau_i'')) = S(\tau)[3].

The following two lemmas are derived using Lem. 1.

**Lemma 24** Let \(\tau = \bigcap_{i \in I} \tau_{i,1} \rightarrow \ldots \rightarrow \tau_{i,m_i} \rightarrow p_i\) be an organized type and let \(\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow p\) be a type. \(\tau \leq_{A} \sigma\) 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 \(\tau_{i,j} \leq_{A} \sigma\) and such that \(\bigcap_{i \in I'} \tau_{i,m_i} \rightarrow \ldots \rightarrow \tau_{i,m_i} \rightarrow p_i \leq_{A} p\).

**Proof:** We prove both directions by induction over \(m\).

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

\(m \Rightarrow m + 1\): We have \(\tau \leq_{A} \sigma\). We write \(\sigma = \sigma_1 \rightarrow \sigma', i.e., \sigma' = \sigma_2 \rightarrow \ldots \rightarrow \sigma_{m+1} \rightarrow \rho\), and \(\tau = \bigcap_{i \in I} p_j \cap \bigcap_{i \in I'} \tau_{i,1} \rightarrow \tau_i', i.e., J' \subseteq I\) is the subset of \(I\) where \(\tau_j\) is an atom.

---

3 Equality refers to the identification of types \(\sigma\) and \(\sigma'\) if both \(\sigma' \leq_{A} \sigma\) and \(\sigma \leq_{A} \sigma'\) hold.
We prove both directions by induction over the requirements in the lemma.

\(\Leftarrow: m = 0\): We get \(\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \rho\), and \(\tau' = \tau_2 \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow \rho\). Lemma 1 states that there is a nonempty subset \(H\) of \(I''\) such that for all \(i \in H\) we have \(c_i \leq_\Lambda \tau_i\) and \(\cap_{i \in H} \tau'_i \leq_\Lambda \sigma'\). The second inequality can be rewritten as \(\cap_{i \in H} \tau_2 \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow \rho\). Applying the induction hypothesis to this inequality we see that there is a subset \(\tilde{H}\) of \(I''\) such that for all \(i \in \tilde{H}\) we have \(c_i \leq_\Lambda \tau_i\), which is equivalent to \(\tilde{\tau} = \tau_2 \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow \rho\). Because \(\tilde{H} \subseteq H\) for all \(i \in \tilde{H}\) we have \(c_1 \leq_\Lambda \tau_1\), and therefore \(\tilde{H} \subseteq H\) satisfies the requirements in the lemma.

\(\Rightarrow: m + 1\): We write \(\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\) be a path with \(m \leq n\). \(\tau \leq_\Lambda \sigma\) if and only if for all \(1 \leq j \leq m\) we have \(\sigma_j \leq_\Lambda \tau_j\) and \(\rho \leq_\Lambda \sigma_{m+1} \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\).

Proof: We prove both directions by induction over \(m\).

\(\Rightarrow: m = 0\): We have \(\tau \leq_\Lambda \sigma\) and \(\tau = \rho\). This implies \(\rho \leq_\Lambda \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\).

\(\Rightarrow: m + 1\): We write \(\sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\), \(\tau = \tau_1 \rightarrow \ldots \rightarrow \tau_m \rightarrow \rho\), and \(\tau' = \tau_2 \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow \rho\). We assume \(\tau \leq_\Lambda \sigma\).Lemma 1 shows that this is only possible if \(c_1 \leq_\Lambda \tau_1\) and \(\tau' \leq_\Lambda \sigma'\). The second inequality is equivalent to \(\tau_2 \rightarrow \ldots \rightarrow \tau_{m+1} \rightarrow \rho \leq_\Lambda \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\). Applying the induction hypothesis we get \(c_j \leq_\Lambda \tau_j\) for all \(2 \leq j \leq m + 1\) and \(\rho \leq_\Lambda \sigma_{m+2} \rightarrow \ldots \rightarrow \sigma_n \rightarrow p\).

\(\Leftarrow: m = 0\): We have \(\tau = \rho \leq_\Lambda \sigma_1 \rightarrow \ldots \rightarrow \sigma_n \rightarrow p = \sigma\).
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 in this case and $\tau \leq_A \sigma$ holds. Thus, $S$ matches $c$.

**Line 15:** We have $\sigma = \cap_{i \in I} \sigma_i$ and the constraints $\tau \leq \sigma_i$ were added. Because $S$ matches $C'$ we have $S(\tau) \leq_A S(\sigma_i)$ for all $i \in I$. By monotonicity of $\cap$ we get $S(\tau) \leq_A \cap_{i \in I} S(\sigma_i) = S(\cap_{i \in I} \sigma_i) = S(\sigma)$.

**Line 21:** In this case $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_m \rightarrow A$ where $A = \alpha$ is a variable and the constraint $\omega \leq \alpha$ was added. We know that $S$ matches $C'$. This is only possible if $S(\alpha) = \omega$. We have to show that $S$ matches the constraint $\omega \leq \sigma_1 \rightarrow \dots \rightarrow \sigma_m \rightarrow \alpha$. This is clear because $S(\alpha) = \omega$ implies $S(\sigma) = S(\sigma_1) \rightarrow \dots \rightarrow S(\sigma_m) \rightarrow \omega = \omega$.

For the following two cases we have $\sigma = \sigma_1 \rightarrow \dots \rightarrow \sigma_m \rightarrow A$ and $\tau = \cap_{i \in I} \tau_i$ with $\tau_i = \tau_{i,1} \rightarrow \dots \rightarrow \tau_{i,m_i} \rightarrow p_i$.

**Line 30:** $A = \alpha$ 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 $\cap_{i \in I'} \tau_{i,m_i+1} \rightarrow \dots \rightarrow \tau_{i,m_i} \rightarrow p_i \leq \alpha$ were added. Since $S$ matches $C'$ we know $S(\overline{\sigma_j}) \leq_A \overline{\tau_{i,j}}$ and $\cap_{i \in I'} \overline{\tau_{i,m_i+1}} \rightarrow \dots \rightarrow \tau_{i,m_i} \rightarrow p_i \leq_A S(\alpha)$. We want to show $\tau \leq_A S(\sigma)$. We write $S(\sigma) = \sigma_1 \rightarrow \dots \rightarrow \sigma_m \rightarrow \rho$ where $S(\sigma_j) = \sigma_j'$ and $S(\alpha) = \rho$. We have $\sigma_j' = \sigma_j' = S(\sigma_j) = S(\overline{\sigma_j}) \leq_A \overline{\tau_{i,j}}$ where the third equality follows from Lem. 23. We may apply the “if”-part of Lem. 24 to conclude $\tau \leq_A S(\sigma)$.

**Line 34:** $A = p$ is a constant and the constraints $\overline{\sigma_j} \leq \overline{\tau_{i,j}}$ for all $1 \leq j \leq m$ and $p_{i,j} \leq p$ were added. Since $S$ matches $C'$ we know $S(\overline{\sigma_j}) \leq_A \overline{\tau_{i,j}}$ for all $1 \leq j \leq m$ and $p_{i,j} \leq_A p$. The
second inequality implies \( p_{i_0} = p \). On the other hand, using Lem. 23 we get \( S(\sigma) = S(\sigma_i) \leq A \tau_{i_{0,i}} = \tau_{i_{0,i}} \) for all \( 1 \leq j \leq m \). Lemma 2 implies \( \tau \leq S(\sigma) \) and \( \tau \) matches \( c \).

For the remaining cases \( \tau = \bigcap_{i \in I} \tau_i \) where \( \tau_i = \tau_{i,1} \rightarrow \ldots \rightarrow \tau_{i,m_i} \rightarrow A_i \) possibly contains variables and \( \sigma \) does not contain any variables.

Line 39: Because \( \sigma = \omega \), it is clear that \( S(\tau) \leq A \sigma \) holds.

Line 44: We have \( \sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \) and the constraints \( \sigma_i \leq \tau_{i_{0,i}} \) for all \( 1 \leq j \leq m_{i_0} \) and \( A_{i_0} \leq \sigma_{m_{i_0}+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \) were added. Because \( \tau \) matches \( C' \) we know \( \tau_{i_{0,i}} \leq \sigma_{m_{i_0}+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \). This implies \( \sigma_j = \sigma_{m_{i_0}+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \). This implies \( \tau_{i_{0,i}} = S(\sigma_{m_{i_0}+1}) = S(\tau_{i_{0,i}}) \), the last equality again following from Lem. 23. We may apply the “if”-part of Lem. 25 to conclude \( S(\tau) \leq A \sigma \).

\[ \square \]

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 50 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, we have to show that the choice does not result in false (where possible). 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 \( C' = C \setminus \{c\} \) (line 10) and not in false.

\[ 33 \]
Line 14: \( \sigma = \bigcap_{i \in I} \sigma_i \) and \( C' = C \setminus \{ c \} \cup \{ t \leq \sigma_i | i \in I \} \). We have to show \( S(\tau) \leq_A S(\sigma) \) for all \( i \in I \). This holds if and only if \( S(\tau) \leq_A \bigcap_{i \in I} S(\sigma_i) \). But this clearly holds because \( \bigcap_{i \in I} S(\sigma_i) = S(\sigma) \) and \( S \) matches \( c \).

Line 17: \( \sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow A \) contains variables. We distinguish cases:

\( \tau = \omega \): We know that \( S \) matches \( \omega \leq \sigma \). Because \( \sigma \) is an arrow type this is only possible if \( A(\sigma) = \omega \). Because \( A \) cannot be \( \omega \) itself (otherwise \( \sigma \) would have been replaced by \( \omega \) in line 6). Thus, \( A \) must be a variable (therefore it is not possible to enter the case where \( \text{false} \) is returned) that is instantiated with \( \omega \) by \( S \) and \( \omega \leq_A S(\sigma) \) clearly holds.

\( \tau \neq \omega \) and \( A \in V \): We have \( \tau = \bigcap_{i \in I} \tau_i,1 \rightarrow \ldots \rightarrow \tau_i,m_i \rightarrow p_i \). Because \( \tau \leq_A S(\sigma) \) 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 \( S(\tau_j) \leq_A \tau_{i,j} \) and such that \( \bigcap_{i \in I'} \tau_i,m_{i+1} \rightarrow \ldots \rightarrow \tau_i,m_i \rightarrow p_i \leq_A S(\sigma) \). From \( S(\tau_j) \leq_A \tau_{i,j} \) we infer \( S(\tau_j) = S(\sigma_j) = S(\sigma) \leq_A \tau_{i,j} \) where the first equality holds because of Lem. 23. This shows that \( S \) matches all newly introduced constraints.

\( \tau \neq \omega \) and \( A \notin V \): Write \( A = p \) and \( \tau = \bigcap_{i \in I} \tau_i,1 \rightarrow \ldots \rightarrow \tau_i,m_i \rightarrow p_i \). Because \( A = p \) we know that \( S(\sigma) \) is still a path. Thus, using \( \tau \leq_A S(\sigma) \) together with Lem. 2 we conclude that there exists an index \( i_0 \in I \) such that \( m_{i_0} = m \), \( p_{i_0} = p \), and \( S(\tau_j) \leq_A \tau_{i,j} \) for all \( j \leq m \). Therefore, in line 33 we may choose exactly this \( i_0 \) (note that \( i_0 \) is indeed contained in \( I \) because \( m_{i_0} = m \)). Using Lem. 23, we conclude \( S(\tau_j) \leq_A \tau_{i_0,j} \) for all \( j \leq m \) from \( S(\tau_j) \leq_A \tau_{i,j} \) and from \( p_{i_0} = p \) we conclude \( p_{i_0} = S(\sigma) \). Therefore, all newly introduced constraints are matched by \( S \).

Line 37: We distinguish two cases:

\( \sigma = \omega \): No new constraints were added and therefore \( S \) matches \( C' \).

\( \sigma \neq \omega \): We write \( \tau = \bigcap_{i \in I} \tau_i \) where \( \tau_i = \tau_i,1 \rightarrow \ldots \rightarrow \tau_i,m_i \rightarrow A_i \) and \( \sigma = \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \), and we know \( S(\tau) \leq_A \sigma \). We organize \( S(\tau) \) and we write \( \overline{S(\tau)} = \bigcap_{i \in H} \rho_{i,1} \rightarrow \ldots \rightarrow \rho_{i,m_i} \rightarrow b_i \). We apply Lem. 2 to \( S(\tau) \leq_A \sigma \) and conclude that there exists \( h_0 \in H \) with \( b_{h_0} = a \), \( m_{h_0} = m \), and \( \sigma_i \leq_A \rho_{i,0,l} \) for all \( 1 \leq l \leq m \). In particular, this implies that \( \pi_{h_0} = \rho_{h_0,1} \rightarrow \ldots \rightarrow \rho_{h_0,m} \rightarrow a \) is a subtype of \( \sigma \). Furthermore, for \( \pi_{h_0} \) there must be an index \( j_0 \) in \( I \) such that \( \pi_{h_0,j} \) occurs as a component in \( \overline{S(\tau_{j_0})} \). It is clear that \( m_{h_0} \leq m \). 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 \( I \) (cf. line 42), and we may choose \( i_0 = j_0 \) in line 43.

We have to show that \( S \) matches \( \overline{\tau_j} \leq \overline{\tau_{j_0}} \) for all \( 1 \leq l \leq m_{j_0} \) and \( A_{h_0} \leq \sigma_{m_{h_0}+1} \rightarrow \ldots \rightarrow \sigma_m \rightarrow a \). We have \( S(\tau_j) = \overline{S(\tau_j)} \leq_A \pi_{h_0} \leq_A \sigma \). Applying the “only if”-part of Lem. 25 to \( S(\tau_{h_0,1}) \rightarrow \ldots \rightarrow S(\tau_{h_0,m_{h_0}}) \rightarrow S(A_{h_0}) = S(\tau_{j_0}) \leq_A \sigma = \sigma_1 \rightarrow \ldots \rightarrow \)
σ_m → a, we obtain σ_l ≤_A S(τ_{h,l}) for all 1 ≤ l ≤ m_{h} and S(A_{i0}) ≤_A σ_{m_{i0} + 1} → ... → σ_m → a. Using Lem. 23 we get S(τ_{h,l}) ≤_A S(A_{i0}). Altogether this shows that S matches the newly introduced constraints.

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.
Matching can be used to further optimize Alg. 1. First (Sect. 5.1), we filter out types in \( \Gamma \) that cannot contribute to inhabiting \( \tau \) 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 \( \Gamma \). Whenever a combinator \((x : \sigma)\) is chosen from \( \Gamma \) we may assume that all variables occurring in \( \sigma \) are fresh. Thus, indeed, we face matching problems.

5.1 MATCHING OPTIMIZATION

Algorithm 3 below checks for every target of every component \( \sigma_j \) of \( \sigma \) and \( \tau_i \) of the inhabitation goal \( \tau \) whether the constraint consisting of the corresponding target and \( \tau_i \) is matchable. The \( \tau_i \) can be inhabited by different components of \( \sigma \). 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 \( \text{tgt}_{m_j}(\sigma_j) \) in line 8 where \( n > m_j = |\sigma_j| \). Any such call to Alg. 2 in this line is assumed to return false if \( \text{tgt}_{m_j}(\sigma_j) = \alpha \) 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 \( \sigma \) be an organized type and let \( S \) be a level-k substitution.
We have \( |S(\sigma)| \leq |\sigma| + k \).

**Proof:** Let \( \sigma_1 \rightarrow \ldots \rightarrow \sigma_m \rightarrow \alpha \) be a longest path in \( \sigma \) 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 \( |S(\sigma_1) \rightarrow \ldots \rightarrow S(\sigma_m) \rightarrow S(\alpha)| = m + k \). There are two cases: If \( S(\sigma_1) \rightarrow \ldots \rightarrow S(\sigma_m) \rightarrow S(\alpha) \) 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 \( \sigma \), 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 \( \tau \) 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 INH2(Γ, τ, k)

1. Input: Γ, τ, k — wlog: All types in Γ and τ = \( \bigcap_{i \in I} \tau_i \) are organized
2. Output: INH2 accepts iff \( \exists e \) such that \( \Gamma \vdash_k e : \tau \)
3. loop:
4. CHOOSE \((x : \sigma) \in \Gamma; \)
5. write \(\sigma = \bigcap_{j \in J} \sigma_j;\)
6. for all \(i \in I, j \in J, n \leq \|\sigma\| + k\) do
7. candidates \((i, j, n) := \text{Match}(tgt_n(\sigma_j) \leq \tau_i)\)
8. end for
9. \(N := \{n \leq \|\sigma\| + k | \forall i \in I \exists j \in J : \text{candidates}(i, j, n) = \text{true}\}\)
10. CHOOSE \(n \in N;\)
11. for all \(i \in I\) do
12. CHOOSE \(j_i \in J\) with candidates \((i, j_i, n) = \text{true}\)
13. CHOOSE \(S_i \in S_x^{(\Gamma, \tau, k)}\)
14. CHOOSE \(\pi_i \in P_n(S_i(\sigma_j))\)
15. end for
16. if \(\forall i \in I : tgt_n(\pi_i) \leq_A \tau_i\) then
17. if \(n = 0\) then
18. ACCEPT;
19. end if
20. FORALL \((l = 1 \ldots n) \tau := \bigcap_{i \in I} \text{arg}_l(\pi_i);\)
21. GOTO loop;
22. end if
23. else
24. FAIL;
25. end if
26. end if
27. end if

2. There exists \(S \in S_x^{(\Gamma, \tau, k)}\) and a path \(\pi \in P_m(S_x(\sigma))\) such that
   a) \(tgt_m(\pi) \leq_A \tau;\)
   b) \(\Gamma \vdash_k e_l : \text{arg}_l(\pi), \text{for all } l \leq m.\)

3. There exists \(j \in J, S \in S_x^{(\Gamma, \tau, k)}, \text{ and } \pi \in P_m(S(\sigma_j))\) such that
   a) \(tgt_m(\pi) \leq_A \tau;\)
   b) \(\Gamma \vdash_k e_l : \text{arg}_l(\pi), \text{for all } l \leq m.\)

Proof: The implication 1. \(\Rightarrow\) 2. follows from Cor. 5.
We want to prove $2. \implies 3$: Denote by $S'$ and $\pi'$ the substitution and path in condition 2. Because $P_m(S'(\sigma)) = P_m(\bigcap_{j \in J} S'(\sigma_j))$ it is clear that there is an index $j'$ such that $\pi'$ occurs in $S'(\sigma_{j'})$. Choosing $j = j'$, $S = S'$, and $\pi = \pi'$, the conditions clearly hold.

The implication $3. \implies 1.$ 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. $\Gamma \vdash_k x e_1 \ldots e_m : \tau$

2. For all $i \in I$ there exist $j_i \in J$, $S_i \in S_k^{(\Gamma, \pi, k)}$, and $\pi_i \in P_m(S_i(\sigma_{j_i}))$ with
   a) $\text{tg}t_m(\pi_i) \leq A \tau_i$;
   b) $\Gamma \vdash_k e_l : \text{arg}_l(\pi_i)$, for all $l \leq m$.

**Proof:** It is clear that $\Gamma \vdash_k x e_1 \ldots e_m : \tau$ is equivalent to $\Gamma \vdash_k x e_1 \ldots e_m : \tau_i$ 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 $j_i \in J$, $S_i \in S_k^{(\Gamma, \pi, k)}$, and $\pi_i \in P_m(S_i(\sigma_{j_i}))$ such that

1. $\text{tg}t_m(\pi_i) \leq A \tau_i$;

2. $\Gamma \vdash_k e_l : \text{arg}_l(\pi_i)$, 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 that the choice of the path $\pi_i$ of length $n$ in line 15 of Alg. 3 is only meaningful if for all $1 \leq l \leq n$ and all paths $\pi'$ in $\text{arg}_l(\pi_i)$ there exists $(y : \rho) \in \Gamma$ such that $\rho$ has a path $\rho'$ that has a target $\text{tg}t_m(\rho')$ for which $\text{Match}((\text{tg}t_m(\rho') \leq \pi'))$ returns $\text{true}$. If there is no such $(y : \rho)$ then $\bigcap_{l \in \text{arg}_l(\pi_i)}$ cannot be inhabited and the check in line 8 does not succeed for any combination. This check together with a direct test of $\text{tg}t_m(\pi_i) \leq A \tau_i$ is incorporated into the choice of $\pi_i$ 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 \text{INH3}(\Gamma, \tau, k)

1. Input: $\Gamma, \tau, k$ \text{-} \\ volg: All types in $\Gamma$ and $\tau = \bigcap_{i \in I} \tau_i$ are organized
2. Output: \text{INH3} 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 $i, j \in J, n \leq \|\sigma\| + k$ do
8: candidates $(i, j, n) := \text{Match}(\text{tgt}_n(\sigma_j) \leq \tau_i)$
9: end for
10: $N := \{n \leq \|\sigma\| + k \mid \forall_{i \in I} \exists_{j \in J} : \text{candidates}(i, j, n) = \text{true}\}$
11: \text{CHOOSE} $n \in N$;
12: for all $i \in I$ do
13: \text{CHOOSE} $j_i \in J$ with $\text{candidates}(i, j_i, n) = \text{true}$
14: \text{CHOOSE} $S_i \in \mathcal{S}_k(\Gamma, \tau_i, k)$
15: \text{CHOOSE} $\pi_i \in \mathcal{P}_n(\arg_l(\pi_i))$ such that $\text{tgt}_m(\pi_i) \leq \tau_i$ and for all $1 \leq l \leq n$
16: and all $\pi' \in \arg_l(\pi_i)$ there exists $(x : \rho) \in \Gamma$, a path $\rho'$ in $\rho$, and $m$ such
17: that $\text{Match}(\text{tgt}_m(\rho') \leq \pi') = \text{true}$
18: end for
19:
20: if $n = 0$ then
21: ACCEPT;
22: else
23: \text{FORALL} $(l = 1 \ldots n) \tau := \bigcap_{i \in I} \arg_l(\pi_i)$;
24: GOTO loop;
25: end if

5.3 \text{IMPLEMENTATION AND EXAMPLE}

We implemented our inhabitation algorithms in the .NET-framework (C# and F#) for bcl₀ 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 35 \text{We discuss an example, first, which will then be generalized in order to compare the mean execution times of the two previous algorithms.}

1. \text{We consider a small repository for synthesizing functions in the ring } \mathbb{Z}_4. \text{ It contains the identity function and the successor as well as the predecessor functions in } \mathbb{Z}_4. \text{ Furthermore, there is a composition combinator } c \text{ 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 \text{ denotes}
that the corresponding combinator is a function. It is not strictly necessary, however, here we introduce it to avoid self-applications of $c$.

\[
\Gamma = \{ \text{id} : f \cap (0 \to 0) \cap (1 \to 1) \cap (2 \to 2) \cap (3 \to 3), \\
\text{succ} : f \cap (0 \to 1) \cap (1 \to 2) \cap (2 \to 3) \cap (3 \to 0), \\
\text{pred} : f \cap (0 \to 3) \cap (1 \to 0) \cap (2 \to 1) \cap (3 \to 2), \\
c : (f \cap (\alpha \to \beta)) \to (f \cap (\beta \to \gamma)) \to (f \cap (\gamma \to \delta)) \to (\alpha \to \delta) \}
\]

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) $c(\text{id}, \text{succ}, \text{succ})$
b) $c(\text{succ}, \text{id}, \text{succ})$
c) $c(\text{succ}, \text{succ}, \text{id})$
d) $c(\text{pred}, \text{pred}, \text{id})$
e) $c(\text{id}, \text{pred}, \text{pred})$
f) $c(\text{pred}, \text{id}, \text{pred})$

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 $\tau_0$ 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.
2. We generalize the previous example by defining the following parametrized type environment:

\[ \Gamma_n^m := \{ \text{id} : f \cap (0 \rightarrow 0) \cap \ldots \cap ((n-1) \rightarrow (n-1)), \]
\[ \text{succ} : f \cap (0 \rightarrow 1) \cap \ldots \cap ((n-1) \rightarrow 0), \]
\[ \text{pred} : f \cap (0 \rightarrow (n-1)) \cap \ldots \cap ((n-1) \rightarrow (n-2)), \]
\[ c_m : (f \cap (\alpha_0 \rightarrow \alpha_1)) \rightarrow \ldots \rightarrow (f \cap (\alpha_{m-1} \rightarrow \alpha_m)) \rightarrow (\alpha_0 \rightarrow \alpha_m) \} \]

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

\[ \Gamma_n^m \vdash ? : (0 \rightarrow 2) \cap (1 \rightarrow 3) \cap \ldots \cap ((n-1) \rightarrow 1) \]

Table 5.1 compares the number of new inhabitation goals (\#ig) to be generated as well as the mean execution time (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,
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.

<table>
<thead>
<tr>
<th>n</th>
<th>m</th>
<th>Algorithm</th>
<th>#ig</th>
<th>ET/ms</th>
<th>sd/ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>73</td>
<td>84.25</td>
<td>0.375</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>96.75</td>
<td>4.25</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>43905</td>
<td>29631</td>
<td>127.5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>55</td>
<td>121</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>$4 \times 10^7$</td>
<td>$5.9 \times 10^7$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2188</td>
<td>364</td>
<td>8.5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>$1.3 \times 10^{14*}$</td>
<td>$1.9 \times 10^{14*}$</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>33</td>
<td>197</td>
<td>0.5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>$6.6 \times 10^{18*}$</td>
<td>$9.8 \times 10^{18*}$</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3889</td>
<td>2270</td>
<td>12.5</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental results for $\Gamma^m_n$
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 $\mathbf{bcl}_{k}$ 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 $\leq_{A}$. 
BIBLIOGRAPHY


