Finite Combinatory Logic with Intersection Types

Jakob Rehof[1] and Paweł Urzyczyn[2⋆]

[1] Technical University of Dortmund
Department of Computer Science
jakob.rehof@cs.tu-dortmund.de
[2] University of Warsaw
Institute of Informatics
urzy@mimuw.edu.pl

Abstract. Combinatory logic is based on modus ponens and a schematic (polymorphic) interpretation of axioms. In this paper we propose to consider expressive combinatory logics under the restriction that axioms are not interpreted schematically but ,,literally", corresponding to a monomorphic interpretation of types. We thereby arrive at *finite combinatory logic*, which is strictly finitely axiomatisable and based solely on modus ponens. We show that the provability (inhabitation) problem for finite combinatory logic with intersection types is Exptime-complete with or without subtyping. This result contrasts with the general case, where inhabitation is known to be Expspace-complete in rank 2 and undecidable for rank 3 and up. As a by-product of the considerations in the presence of subtyping, we show that standard intersection type subtyping is in Ptime. From an application standpoint, we can consider intersection types as an expressive specification formalism for which our results show that functional composition synthesis can be automated.

1 Introduction

Under the Curry-Howard isomorphism, combinatory terms correspond to Hilbert-style proofs, based on two principles of deduction: modus ponens and a schematic interpretation of axioms. The latter can be expressed as a meta-rule of instantiation or as a rule of substitution. The schematic interpretation corresponds to considering types of combinators as implicitly polymorphic.[3] Because of the schematic interpretation of axioms, such logics are not strictly finite.

In the present paper we propose to consider *finite combinatory logics*. Such logics arise naturally from combinatory logic by eliminating the schematic interpretation of axioms. This corresponds to a monomorphic interpretation of combinator types, and leaves modus ponens as the sole principle of deduction. We consider combinatory logic with intersection types [4]. In general, the provability problem for this logic is undecidable. We prove that provability in finite

⋆ Partly supported by MNiSW grant N N206 355836.

[3] The schematic viewpoint is sometimes referred to as *typical ambiguity*.

combinatory logic is decidable and EXPTIME-complete. We show, in fact, that all of the problems: emptiness, uniqueness and finiteness of inhabitation are in EXPTIME by reduction to alternating tree automata. We then consider the provability problem in the presence of standard intersection type subtyping. We show that the subtyping relation is in PTIME (previous work only appears to establish an exponential algorithm). We prove that provability remains EXPTIME-complete in the presence of subtyping, and also that uniqueness and finiteness of inhabitation remain in EXPTIME. In contrast to the case without subtyping, which is solved by reduction to tree automata, the upper bounds are achieved using polynomial space bounded alternating Turing machines.

From an application standpoint, we are interested in finite combinatory logic with intersection types as an expressive specification logic for automatic function composition synthesis. Under the Curry-Howard isomorphism, validity is the question of inhabitation: Given a set Γ of typed combinators (function symbols), does there exist an applicative combination e of combinators in Γ having a given type τ? An algorithm for inhabitation in finite combinatory logic can be used to synthesize e from the specification given by Γ and τ.

Due to space limitations some proofs have been left out. All details can be found in the technical report [16] accompanying this paper.

Finite combinatory logic with intersection types. The strictly finite standpoint considered here is perhaps especially interesting for highly expressive logics for which validity is undecidable. Intersection type systems [3, 15] belong to a class of propositional logics with enormous expressive power. Intersection types capture several semantic properties of lambda terms, and their type reconstruction problem has long been known to be undecidable, since they characterize exactly the set of strongly normalizing terms [15]. Intersection types have also been used more generally to capture semantic properties of programs, for example in the context of model checking [13, 7]. The inhabitation problem for the λ-calculus with intersection types was shown to be undecidable in [18] (see also [17] for connections to λ-definability). More recently, the borderline between decidability and undecidability was clarified in [10, 19] by rank restrictions (as defined in [11]), with inhabitation in rank 2 types being shown EXPSPACE-complete and undecidable from rank 3 and up.[4] Combinatory logic with intersections types was studied in [4], where it was shown that the combinatory system is complete in that it is logically equivalent to the λ-calculus by term (proof) translations in both directions. It follows that provability (inhabitation) for combinatory logic with intersections types is undecidable.

Application perspective. Finite combinatory logic can be used as a foundation for automatic composition synthesis via inhabitation: Given a set Γ of typed functions, does there exist a composition of those functions having a given type? Under this interpretation, the set Γ represents a library of functions, and the intersection type system provides a highly expressive specification logic

[4] Other restrictions have also been considered, which arise by limiting the logic of intersection [9, 20].

for which our results show that the problem of typed function composition synthesis can be automated. Our approach provides an alternative to synthesis via proof counting (following [2]) for more complex term languages (e.g., [22]). Most notably, finite combinatory logic is naturally applicable to synthesis from libraries of reusable components which has been studied recently [12], leading to a 2EXPTIME-complete synthesis problem. Our setting is different, since our framework is functional (rather than reactive), it is based on a different specification format (types rather than temporal logic), so we obtain an EXPTIME-complete problem (using the identity APSPACE = EXPTIME). We gain algorithmic control over the solution space to the synthesis problem, since it follows from the reduction that not only the problem of inhabitation but also the problems of finiteness and uniqueness of inhabitation are solvable in EXPTIME. The incorporation of subtyping is important here, since it would allow synthesis specifications to take advantage of hierarchical structure, such as inheritance or taxonomies.

Although synthesis problems are by nature computationally complex, the exploitation of abstraction in component libraries and type structure allows new ways of scaling down problem size. An implementation of the algorithms described here is currently underway in the context of a component synthesis framework, and future work will address experimental evaluation.

2 Preliminaries

Trees: Let $\mathbf{A}$ and $\mathbf{L}$ be non-empty sets. A *labeled tree* over the *index set* $\mathbf{A}$ and the *alphabet of labels* $\mathbf{L}$ is understood as a partial function $T : \mathbf{A}^* \to \mathbf{L}$, such that the domain $dm(T)$ is non-empty and prefix-closed. The *size* of T, denoted $|T|$, is the number of nodes in (the domain of) T. The *depth* of a tree T, written $\|T\|$, is the maximal length of a member of $dm(T)$. Note that $\|T\| \leq |T|$. For $w \in dm(T)$, we write $T[w]$ for the subtree of T rooted at w, given by $T[w](w') = T(ww')$.

Terms: Applicative terms, ranged over by e etc., are defined by:

$$e \ ::= \ x \mid (e\ e'),$$

where x, y, z etc. range over a denumerable set of variables. We write $e \equiv e'$ for syntactic identity of terms. We abbreviate $(\ldots((e\ e_1)\ e_2)\ldots e_n)$ as $(e\ e_1\ e_2 \ldots e_n)$ for $n \geq 0$ (for $n = 0$ we set $(e\ e_1 \ldots e_n) \equiv e$). Notice that any applicative term can be written uniquely as $(x\ e_1 \ldots e_n)$.

An alternative to the above standard understanding of expressions as words, is to represent them as finite labeled trees over the index set $\mathbb{N}$ and the alphabet $V \cup (\mathbb{N} - \{0\})$. A variable x is seen as a one-node tree where the only node is labeled x. An expression $(x\ e_1 \ldots e_n)$ is a tree, denoted $@_n(x, e_1, \ldots, e_n)$, with root labeled $@_n$ and $n+1$ subtrees $x, e_1, \ldots, e_n$, which are addressed from left to right from the root by the indices $0, 1, \ldots, n$. That is, if $e = @_n(x, e_1, \ldots, e_n)$, we have $e(\varepsilon) = @_n$, $e(0) = x$ and $e(i) = e_i$ for $i = 1, \ldots, n$. In a term of the form $@_n(x, e_1, \ldots, e_n)$ we call the tree node x the *operator* of node $@_n$. We say that a variable x *occurs along an address* $w \in dm(e)$ if for some prefix u of w we have $u0 \in dm(e)$ with $e(u0) = x$.

Types: Type expressions, ranged over by τ, σ etc., are defined by

$$\tau \ ::= \ \alpha \mid \tau \to \tau \mid \tau \cap \tau$$

where α ranges over a denumerable set TV of type variables. We let T denote the set of type expressions. As usual, types are taken modulo idempotency ($\tau \cap \tau = \tau$), commutativity ($\tau \cap \sigma = \sigma \cap \tau$), and associativity ($(\tau \cap \sigma) \cap \rho = \tau \cap (\sigma \cap \rho)$). A type *environment* Γ is a finite set of type assumptions of the form $x : \tau$, and we let $dm(\Gamma)$ and $rn(\Gamma)$ denote the domain and range of Γ.

A type $\tau \cap \sigma$ is said to have τ and σ as *components*. For an intersection of several components we sometimes write $\bigcap_{i=1}^{n} \tau_i$ or $\bigcap_{i \in I} \tau_i$. We stratify types by letting $A, B, C, \ldots$ range over types with no top level conjunction, that is

$$A \ ::= \ \alpha \mid \tau \to \tau$$

We can write any type τ as $\bigcap_{i=1}^{n} A_i$ for suitable $n \geq 1$ and A_i.

Types as trees: It is convenient to consider types as labeled trees. A *type tree* τ is a labeled tree over the index set $\mathbf{A} = \mathbb{N} \cup \{l, r\}$ and the alphabet of labels $\mathbf{L} = TV \cup \{\to, \cap\}$, satisfying the following conditions for $w \in \mathbf{A}^*$:

- if $\tau(w) \in TV$, then $\{a \in \mathbf{A} \mid wa \in dm(\tau)\} = \emptyset$;
- if $\tau(w) = \to$, then $\{a \in \mathbf{A} \mid wa \in dm(\tau)\} = \{l, r\}$;
- if $\tau(w) = \cap$, then $\{a \in \mathbf{A} \mid wa \in dm(\tau)\} = \{1, \ldots, n\}$, for some n.

Trees are identified with types in the obvious way. Observe that a type tree is LOGSPACE-computable from the corresponding type and vice versa. Recall that types are taken modulo idempotency, commutativity and associativity, so that a type has in principle several tree representations. In what follows we always assume a certain fixed way in which a type is interpreted. By representing A as $\bigcap_{i=1}^{1} A_i$ we can assume w.l.o.g. that types τ are *alternating*, i.e., that all $w \in dm(\tau)$ alternate between members of $\mathbb{N}$ and $\{l, r\}$ beginning and ending with members of $\mathbb{N}$, and we call such w *alternating addresses*. We define *right-addresses* to be members of $\mathbf{A}^*$ with no occurrence of l. For an alternating type τ, we define $\mathbf{R}_n(\tau)$ to be the set of alternating right-addresses in $dm(\tau)$ with n occurrences of r. Observe that alternating right-addresses have the form $n_0 r n_1 r \ldots r n_k$ with $n_j \in \mathbb{N}$ for $0 \leq j \leq k$, in particular note that $\mathbf{R}_0(\tau) = \{1, \ldots, k\}$, for some k. For $w \in \mathbf{R}_n$, $n \geq 1$ and $1 \leq j \leq n$, we let $w \downarrow j$ denote the address that arises from chopping off w just before the j'th occurrence of r in w, and we define $L(j, w) = (w \downarrow j)l$. The idea is that alternating right-addresses in τ determine "target types" of τ, i.e., types that may be assigned to applications of a variable of type τ. For example, if $\tau = \cap\{\alpha_1 \cap \alpha_2 \to \beta_1 \cap \beta_2\}$, then we have $1r1, 1r2 \in \mathbf{R}_1(\tau)$ with $\tau[1r1] = \beta_1$ and $\tau[1r2] = \beta_2$. Moreover, $\tau[L(1, 1r1)] = \tau[1l] = \alpha_1 \cap \alpha_2$. Now suppose that $x : \tau$ and observe that an expression $(x\ e)$ can be assigned any of types β_1 and β_2, provided $e : \alpha_1 \cap \alpha_2$.

Definition 1 (*arg*, *tgt*). The *j'th argument of τ at w* is $arg(\tau, w, j) = \tau[L(j, w)]$, and the *target of τ at w* is $tgt(\tau, w) = \tau[w]$.

Definition 2 (Sizes). The *size* (resp. *depth*) of a term or type is the size (resp. depth) of the corresponding tree. The *size* of an environment Γ is the sum of the cardinality of the domain of Γ and the sizes of the types in Γ, i.e., $|\Gamma| = |dm(\Gamma)| + \sum_{x \in dm(\Gamma)} |\Gamma(x)|$. Also we let $\| \Gamma \|$ denote the maximal depth of a member of $rn(\Gamma)$.

Type assignment: We consider the applicative restriction of the standard intersection type system, which arises from that system by removing the rule ($\to$I) for function type introduction. The system is shown in Figure 1 and is referred to as finite combinatory logic with intersection types, FCL($\cap$). We consider the following decision problem:

Inhabitation problem for FCL($\cap$)*:* Given an environment Γ and a type τ, does there exist an applicative term e such that $\Gamma \vdash e : \tau$?

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \ (\text{var}) \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \ (\to\text{E})$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash e : \tau_1 \cap \tau_2} \ (\cap\text{I}) \qquad \frac{\Gamma \vdash e : \tau_1 \cap \tau_2}{\Gamma \vdash e : \tau_i} \ (\cap\text{E})$$

Fig. 1: Finite combinatory logic FCL($\cap$)

The difference between our type system and full combinatory logic based on intersection types [4, 21] is that we do not admit a schematic (polymorphic) interpretation of the axioms (types in Γ) but take each of them monomorphically, "as is". Moreover, we do not consider a fixed base of combinators, since we are not here interested in combinatory completeness. The study of combinatory logics with intersection types usually contain, in addition, standard rules of subtyping which we consider in Section 4.

As has been shown [4], the standard translation of combinatory logic into the λ-calculus can be adapted to the intersection type system, preserving the main properties of the λ-calculus type system, including characterization of the strongly normalizing terms (see also [21, Section 2], which contains a compact summary of the situation). In particular, it is shown in [4, Theorem 3.11], that the combinatory system is logically equivalent to the λ-calculus with intersection types. By composing this result with undecidability of inhabitation in the λ-calculus [18], it immediately follows that inhabitation is undecidable for the full combinatory system.

We note that the system FCL($\cap$) is sufficiently expressive to uniquely specify any given expression: for every applicative term e there exists an environment Γ_e and a type τ_e such that e types uniquely in Γ_e at τ_e, i.e., $\Gamma_e \vdash e : \tau_e$ and for any term e' such that $\Gamma_e \vdash e' : \tau_e$ it is the case that $e \equiv e'$. This property is of interest for applications in synthesis. We refer to [16] for more details.

3 EXPTIME-completeness of inhabitation

An EXPTIME-lower bound for applicative inhabitation follows easily from the known result (apparently going back to H. Friedman 1985 and published later by others, see [8]) that the composition problem for a set of functions over a finite set is EXPTIME-complete, because an intersection type can be used to represent a finite function. It may be instructive to observe that the problem (without subtyping) can be understood entirely within the theory of tree automata. In [16], we give an alternative, simple reduction from the intersection non-emptiness problem for tree automata. The proof works for types of rank 2 or more.

For the upper bound we prove that the inhabitation problem is reducible to the non-emptiness problem for polynomial-sized alternating top-down tree automata [5], and hence it is in APSPACE = EXPTIME. Representing the inhabitation problem by alternating tree automata, we can conclude that also finiteness and uniqueness of inhabitation are solvable in EXPTIME.

Let us point out that the presence of intersection makes an essential difference in complexity. It can easily be shown [16] that the inhabitation problem for the $\cap$-free fragment of FCL($\cap$) is solvable in polynomial time.

3.1 EXPTIME upper bound

We use the definition of alternating tree automata from [5]. We have adapted the definition accordingly, as follows. Below, $\mathcal{B}^+(S)$ denotes the set of positive propositional formulae over the set S, i.e., formulae built from members of S as propositional variables, conjunction, and disjunction.

Definition 3 (Alternating tree automaton). An *alternating tree automaton* is a tuple $\mathcal{A} = (Q, \mathcal{F}, q_0, \Delta)$, where q_0 is an initial state, $\mathcal{F}$ is a label alphabet, $I \subseteq Q$ is a set of initial states, and Δ is a mapping with $dm(\Delta) = Q \times \mathcal{F}$, such that $\Delta(q, @_n) \in \mathcal{B}^+(Q \times \{0, \ldots, n\})$ and $\Delta(q, x) \in \{true, false\}$.

We use the notation $\langle q, i\rangle$ for members of the set $Q \times \{0, \ldots, n\}$. For example, for a transition $\Delta(q, f) = \langle q', i\rangle \wedge \langle q'', i\rangle$, the automaton moves to both of the states q' and q'' along the i'th child of f (universal transition), whereas $\Delta(q, f) = \langle q', i\rangle \vee \langle q'', i\rangle$ is a nondeterministic (existential) transition. In the first case, it is necessary that both $\langle q', i\rangle$ and $\langle q'', i\rangle$ lead to accept states, in the second case it suffices that one of the successor states leads to an accept state. More generally, let $Accept(q, e)$ be true iff the automaton accepts a tree e starting from state q. Then $Accept(q, e)$ can be described as the truth value of $\Delta(q, f)$, where f is the label at the root of e, under the interpretation of each $\langle\, q', i\,\rangle$ as $Accept(q', e_i)$. We refer the reader to [5, Section 7.2] for full details on alternating tree automata.

Recognizing inhabitants. Given Γ and τ, we define a tree automaton $\mathcal{A}(\Gamma, \tau)$, assuming the alternating representation of types and term syntax based on the tree constructors $@_n$, as introduced in Section 2. The property of $\mathcal{A}(\Gamma, \tau)$ is:

$$\Gamma \vdash e : \tau \;\text{ if and only if }\; e \in \mathbf{L}(\mathcal{A}(\Gamma, \tau)).$$

The definition is shown in Figure 2. The constructor alphabet of our automaton $\mathcal{A}(\Gamma,\tau)$ is $\Sigma = \{@_n\} \cup dm(\Gamma)$. The states of $\mathcal{A}(\Gamma,\tau)$ are either "search states" of the form $\textsc{src}(\tau')$ or "check states" of the form $\textsc{chk}(A,B,x)$, where $x \in dm(\Gamma)$ and τ', A and B are subterms of types occurring in Γ (recall that types ranged over by A, B etc. do not have top level intersections, and τ are alternating types).

The initial state of $\mathcal{A}(\Gamma,\tau)$ is $\textsc{src}(\tau)$. The transition relation of $\mathcal{A}(\Gamma,\tau)$ is given by the two transition rules shown in Figure 2, for $0 \leq n \leq \| \Gamma \|$.

$$\textsc{src}(\textstyle\bigcap_{i=1}^{k} A_i) \xrightarrow{@_n} \bigvee_{x \in dm(\Gamma)} \bigwedge_{i=1}^{k} \bigvee_{w \in \mathbf{R}_n(\Gamma(x))} (\bigwedge_{j=1}^{n} \langle \textsc{src}(arg(\Gamma(x),w,j)) \ , \ j \rangle \wedge \langle \textsc{chk}(A_i, tgt(\Gamma(x),w), x) \ , \ 0 \rangle)$$

$$\textsc{chk}(A,B,x) \xrightarrow{y} \textit{true}, \text{ if } A = B \text{ and } x = y, \text{ otherwise } \textit{false}$$

Fig. 2: Alternating tree automaton for $\textsc{fcl}(\cap)$

Intuitively, from search states of the form $\textsc{src}(\tau)$ the automaton recognizes n-ary inhabitants (for $0 \leq n \leq \| \Gamma \|$) of τ by nondeterministically choosing an operator x and verifying that an expression with x as main operator inhabits all components of τ. Given the choice of x, a right-path w is guessed in every component, and the machine makes a universal move to inhabitation goals of the form $\textsc{src}(arg(\Gamma(x),w,j))$, one for each of the argument types of τ as determined by the address w. In these goals, the type $arg(\Gamma(x),w,j)$ denotes the j'th such argument type, and the goal is coupled with the child index j, for $j = 1,\ldots,n$. Moreover, it is checked from the state $\textsc{chk}(A_i, tgt(\Gamma(x),w),x)$ that the target type of x at position w is identical to A_i.

Correctness. In order to prove correctness of our representation in alternating tree automata we need a special generation lemma (Lemma 4), in which we assume the alternating representation of general types τ.

Lemma 4 (Path Lemma). *The assertion $\Gamma \vdash (x\, e_1 \ldots e_n) : A$ holds if and only if there exists $w \in \mathbf{R}_n(\Gamma(x))$ such that*

1. *$\Gamma \vdash e_j : arg(\Gamma(x),w,j)$, for $j = 1,\ldots,n$;*
2. *$A = tgt(\Gamma(x),w)$.*

Proof. The proof from right to left is obvious. The proof from left to right is by induction with respect to $n \geq 0$. The omitted details can be found in [16]. □

Proposition 5. *$\Gamma \vdash e : \tau$ if and only if $e \in \mathbf{L}(\mathcal{A}(\Gamma,\tau))$*

Proof. For $e = @_n(x, e_1, \ldots, e_n)$ let $child_0(e)$ be x, and let $child_j(e)$ be e_j, when $j = 1,\ldots,n$. Let $\tau = \bigcap_{i=1}^{k} A_i$. By Lemma 4, we have $\Gamma \vdash e : \tau$ if and only if

$$\exists x\, \forall i\, \exists w\, \forall j\, [(\Gamma \vdash child_j(e) : arg(\Gamma(x),w,j)) \wedge A_i = tgt(\Gamma(x),w) \wedge x = child_0(e)].$$

The automaton $\mathcal{A}(\Gamma,\tau)$ is a direct implementation of this specification. □

As a consequence of the tree automata representation we immediately get decision procedures for emptiness, finiteness and uniqueness of inhabitation, via known results for the corresponding properties of tree automata.

Theorem 6. *In system* FCL($\cap$) *the following problems are in* EXPTIME.

1. *Emptiness of inhabitation;*
2. *Finiteness of inhabitation;*
3. *Uniqueness of inhabitation.*

In addition, problem (1) is EXPTIME*-complete.*

Proof. The size of $\mathcal{A}(\Gamma, \tau)$ is polynomial in the size of the problem, since its state set is composed of variables and subterms of types occurring in Γ and τ. Moreover, the number of transitions is bounded by $\| \Gamma \|$. The results now follow from Propostion 5 together with the polynomial time solvability of the respective problems for deterministic tree automata and the fact that alternation can be eliminated with an exponential blow-up in the size of the automaton [5]. □

4 Subtyping

In this section we prove that the inhabitation problem remains EXPTIME-complete in the presence of subtyping (Section 4.2), while the problems of uniqueness and finiteness remain in in EXPTIME. We also show (Section 4.1) that the subtyping relation itself is decidable in polynomial time.

We extend the type language with the constant ω and stratify types as follows. Type variables and ω are called *atoms* and are ranged over by a, b, etc. Function types $(A, B, \dots)$, and general types $(\tau, \sigma, \dots)$ are defined by

$$A ::= a \mid \tau \to \tau \qquad\qquad \tau ::= a \mid \tau \to \tau \mid \bigcap_{i \in I} \tau_i$$

Subtyping $\leq$ is the least preorder (reflexive and transitive relation), satisfying:

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

We identify σ and τ when $\sigma \leq \tau$ and $\tau \leq \sigma$.

4.1 PTIME decision procedure for subtyping

We show that the intersection type subtyping relation $\leq$ is decidable in polynomial time. It appears that very little has been published about algorithmic properties of the relation so far. The only result we have found is the decidability result of [6] from 1982. The key characterization in [6] (also to be found in [9, Lemma 2.1]) is that for *normalized* types A_i, B_j ($i \in I$, $j \in J$) one has

$$\bigcap_{i \in I} A_i \leq \bigcap_{j \in J} B_j \;\Leftrightarrow\; \forall j \exists i \; A_i \leq B_j \,.$$

Normalization, in turn, may incur an exponential blow-up in the size of types. Hence, a direct algorithm based on the above leads to an exponential procedure.

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

Lemma 7. *Let a_j, for $j \in J$, be atoms.*

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

It is not too difficult to construct a PSPACE-algorithm by a direct recursive implementation of Lemma 7. We can however do better using memoization techniques in a combined top-down and bottom-up processing of type trees.

Theorem 8. *The subtyping relation $\leq$ is in* PTIME.

Proof. Given types τ_0 and σ_0, to decide if $\tau_0 \leq \sigma_0$, we describe a procedure for answering *simple questions* of the form $\tau \leq \sigma$ or $\tau \geq \sigma$ (where τ and σ are, respectively, subterms of τ_0 and σ_0) and *set questions* of the form $\bigcap_i \tau_i \leq \sigma$ or $\tau \geq \bigcap_i \sigma_i$ (where τ_i and σ_i, but not necesarily the intersections, are subterms of τ_0, σ_0). The problem is that the total number of set questions is exponential.

We introduce memoization, where it is assumed that when a question is asked, all simple questions (but not all set questions) concerning smaller types have been already answered and the answers have been recorded.

To answer a simple question $\tau \leq \sigma$, we must decide if $\tau \leq \theta$ for all components θ of the $\cap$-type σ. Let $\tau = \bigcap_{j\in J} a_j \cap \bigcap_{i\in I}(\varrho_i \to \mu_i)$. If θ is an atom different from ω, we only check if it occurs among the a_j's. Otherwise, $\theta = \xi \to \nu$ and we proceed as follows:

- Determine the set $K = \{i \in I \mid \xi \leq \varrho_i\}$ by inspecting the memoized answers;
- Recursively ask the set question $\bigcap_{i\in K} \mu_i \leq \nu$.

After all inequalities $\tau \leq \theta$ have been verified, the answer $\tau \leq \sigma$ or $\tau \not\leq \sigma$ is stored in memory.

For a set question $\bigcap_i \tau_i \leq \sigma$, write the $\bigcap_i \tau_i$ as $\bigcap_{j\in J} a_j \cap \bigcap_{i\in I}(\varrho_i \to \mu_i)$. Then we proceed as in the simple case, except that the result is not memoized. The algorithm is formalized in our report [16].

To answer a single question (simple or set) one makes as many recursive calls as there are components of the right-hand side. The right-hand sides ("targets") of the recursive calls are different subterms of the original target. Further recursive calls address smaller targets, and the principal observation is that *all recursive calls incurred by any fixed question have different targets*. Therefore the total number of calls to answer a single question is at most linear. Since linear time is enough to handle a single call, we have a quadratic bound to decide a simple question, and $\mathcal{O}(n^4)$ total time. □

4.2 Inhabitation with subtyping

We extend the system FCL($\cap$) with subtyping. The resulting system is denoted FCL($\cap, \leq$) and is shown in Figure 3. (The axiom $\Gamma \vdash e : \omega$ is left out for simplicity, since it has no interesting consequences for the theory developed in this paper.)

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \ (\text{var}) \qquad \frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \ (\to\text{E})$$

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

Fig. 3: FCL($\cap, \leq$)

We now prove that inhabitation remains EXPTIME-complete in the presence of subtyping. We begin with a path characterization of typings.

Lemma 9. *For any τ and $w \in \mathbf{R}_n(\tau)$ one has:*

$$\tau \leq \mathit{arg}(\tau, w, 1) \to \cdots \to \mathit{arg}(\tau, w, n) \to \mathit{tgt}(\tau, w).$$

Proof. By induction on $n \geq 0$. Details omitted. □

Lemma 10 (Path Lemma). *The assertion $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$ holds if and only if there exists a subset S of $\mathbf{R}_n(\Gamma(x))$ such that*

1. *$\Gamma \vdash e_j : \bigcap_{w \in S} \mathit{arg}(\Gamma(x), w, j)$, for $j = 1, \ldots, n$;*
2. *$\bigcap_{w \in S} \mathit{tgt}(\Gamma(x), w) \leq \tau$.*

Proof. Part ($\Leftarrow$) is an easy consequence of Lemma 9. Part ($\Rightarrow$) goes by induction with respect to the derivation of $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$. The omitted details can be found in the full report [16]. □

Tree automaton. Using Lemma 10 it is possible to specify, as in the subtyping-free case, an alternating tree automaton $\mathcal{A}(\Gamma, \tau)$ of the following property (details are given in [16]):

$$\Gamma \vdash e : \tau \quad \text{if and only if} \quad e \in \mathbf{L}(\mathcal{A}(\Gamma, \tau)).$$

Unfortunately, Lemma 10 requires that we handle *subsets* S of right-addresses rather than single addresses, and hence the size of the modified automaton could be exponential in the size of the input. As a consequence, this solution would only imply a *doubly exponential* upper bound for the emptiness, finiteness and uniqueness problems of inhabitation (recall that an additional exponential blow-up is incurred by determinization, following the proof of Theorem 6). However, together with Theorem 6 this reduction to tree automata implies:

Corollary 11. *In system* FCL($\cap$) *and* FCL($\cap, \leq$) *the tree languages of inhabitants given by* $\mathbf{L}(\Gamma, \tau) = \{e \mid \Gamma \vdash e : \tau\}$ *are regular.*

Turing machine. We can achieve an exponential upper bound for FCL($\cap, \leq$) by a direct implementation of the algorithm implicit in Lemma 10 on a Turing machine. Notice that we cannot apply the same approach as with tree automata, namely define a machine recognizing inhabitants for a fixed pair Γ and τ, and then check if the language accepted by the machine is nonempty. Indeed, emptiness is undecidable even for LOGSPACE machines. But we can construct a polynomial space bounded alternating Turing machine $\mathcal{M}$ such that

$$\mathcal{M} \text{ accepts } (\Gamma, \tau) \quad \text{ if and only if } \quad \Gamma \vdash e : \tau, \text{ for some } e.$$

The main point is that each choice of the set $S \subseteq \mathbf{R}_n(\Gamma(x))$ can be stored in linear space on the tape of the Turing machine.

The definition shown in Figure 4 specifies an alternating Turing machine $\mathcal{M}$ which accepts on input Γ and τ if and only if there exists an applicative inhabitant of τ in the environment Γ. Recall from e.g. [14, section 16.2] that the state set Q of an alternating Turing machine $\mathcal{M} = (Q, \Sigma, \Delta, s)$ is partitioned into two subsets, $Q = Q_\exists \cup Q_\forall$. States in $Q_\exists$ are referred to as existential states, and states in $Q_\forall$ are referred to as universal states. A configuration whose state is in $Q_\forall$ is accepting if and only if all its successor configurations are accepting, and a confugiration whose state is in $Q_\exists$ is accepting if and only if at least one of its successor configurations is accepting.

The alternating Turing machine defined in Figure 4 directly implements Lemma 10. We use shorthand notation for existential states (CHOOSE ...) and universal states ($\bigwedge \ldots$), where the branching factors depend on parameters of the input (Figure 4, lines 2–4, 5). For example, in line 2, the expression CHOOSE $x \in dm(\Gamma)\ldots$ denotes an existential state from which the machine transitions to a number of successor states that depends on Γ and writes a member of $dm(\Gamma)$ on a designated portion of the tape for later reference. We can treat the notation as shorthand for a loop, in standard fashion. Similarly, line 8 denotes a universal state ($\bigwedge_{j=1}^{n} \ldots$) from which a loop is entered depending on n. In line 5, we call a decision procedure for subtyping, relying on Lemma 8.

The variable τ holds the value of the current inhabitation goal, processed as follows. First, values for x, n and S are existentially generated (lines 2–4). It is then tested (line 5) that the current goal type τ is a supertype of the target intersection and, if so, the machine transitions to the universal state (line 8). In every branch of computation created at step 8, the goal τ is set to the appropriate argument type, and the computation resumes again from line 2. The computation stops when there is no arguments to process (line 6).

Theorem 12. *The inhabitation problem in* FCL($\cap, \leq$) *is* EXPTIME*-complete.*

Proof. Inhabitation remains EXPTIME-hard in the presence of subtyping, and the proof remains essentially the same. Therefore it is omitted, see [16] for details.

Membership in EXPTIME follows from the fact that the Turing machine $\mathcal{M}$ shown in Figure 4 solves the inhabitation problem in alternating polynomial

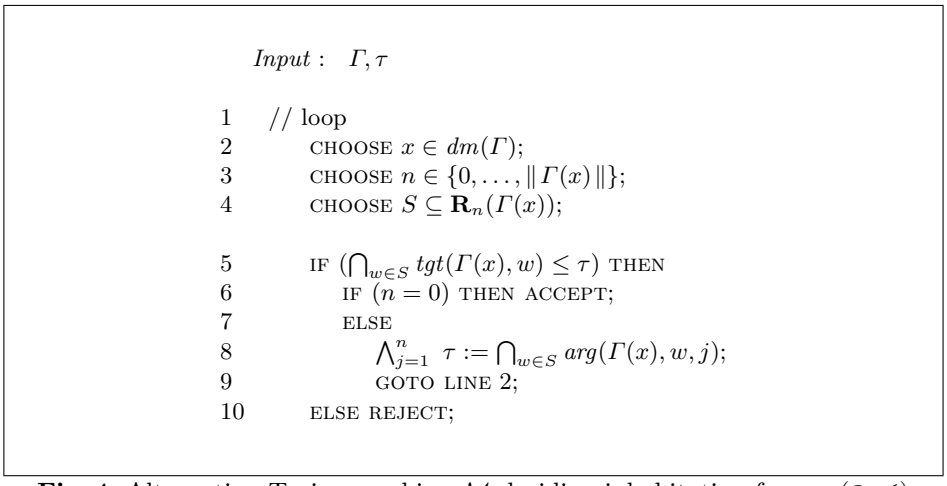

Fig. 4: Alternating Turing machine $\mathcal{M}$ deciding inhabitation for FCL$(\cap, \leq)$

space. Correctness (the machine accepts (Γ, τ) iff $\Gamma \vdash e : \tau$ for some e) follows easily from Lemma 10 (the machine is a direct implementation of the lemma). To see that the machine is polynomial space bounded, notice that the variable S holds a subset of $\mathbf{R}_n$ for $n \leq \|\Gamma\|$, hence it consumes memory linear in the size of the input. Moreover, Lemma 8 shows that the subtyping relation is decidable in polynomial time (and space). Finally, notice that the variable τ holds types that are intersections of distinct strict subterms of the type $\Gamma(x)$, and there is only a linear number of subterms. □

Uniqueness of inhabitation with subtyping. In order to show that uniqueness of inhabitation in system FCL$(\cap, \leq)$ is in EXPTIME we modify the inhabitation algorithm using alternation to search for more than one inhabitant. Figure 5 shows an alternating polynomial space Turing machine accepting Γ and τ if and only if there exists more than one inhabitant e with $\Gamma \vdash e : \tau$ in system FCL$(\cap, \leq)$. In this machine, alternation is used in two different ways, to search for inhabitants of argument types (as was already done in Figure 4) as well as to search for two different inhabitants. The machine solves the *ambiguity problem* (given Γ, τ, does there exist *more than one* term e with $\Gamma \vdash e : \tau$?) by nondeterministically guessing an inhabitant (lines 3–5) and a point from which another inhabitant can be "split" off (lines 6–9). See [16] for full details.

Theorem 13. *In system* FCL$(\cap, \leq)$*, uniqueness of inhabitation is in* EXPTIME*.*

Finiteness of inhabitation with subtyping Lemma 10 (Path Lemma) yields an alternative proof system for FCL$(\cap, \leq)$, consisting only of the following rule schema (P), where $S \subseteq \mathbf{R}_n(\Gamma(x))$:

$$\frac{\Gamma \vdash_P e_j : \bigcap_{w\in S} arg(\Gamma(x), w, j)\ \ (j = 1, \ldots, n) \qquad \bigcap_{w\in S} tgt(\Gamma(x), w) \leq \tau}{\Gamma \vdash_P @_n(x, e_1, \ldots, e_n) : \tau} \quad \text{(P)}$$

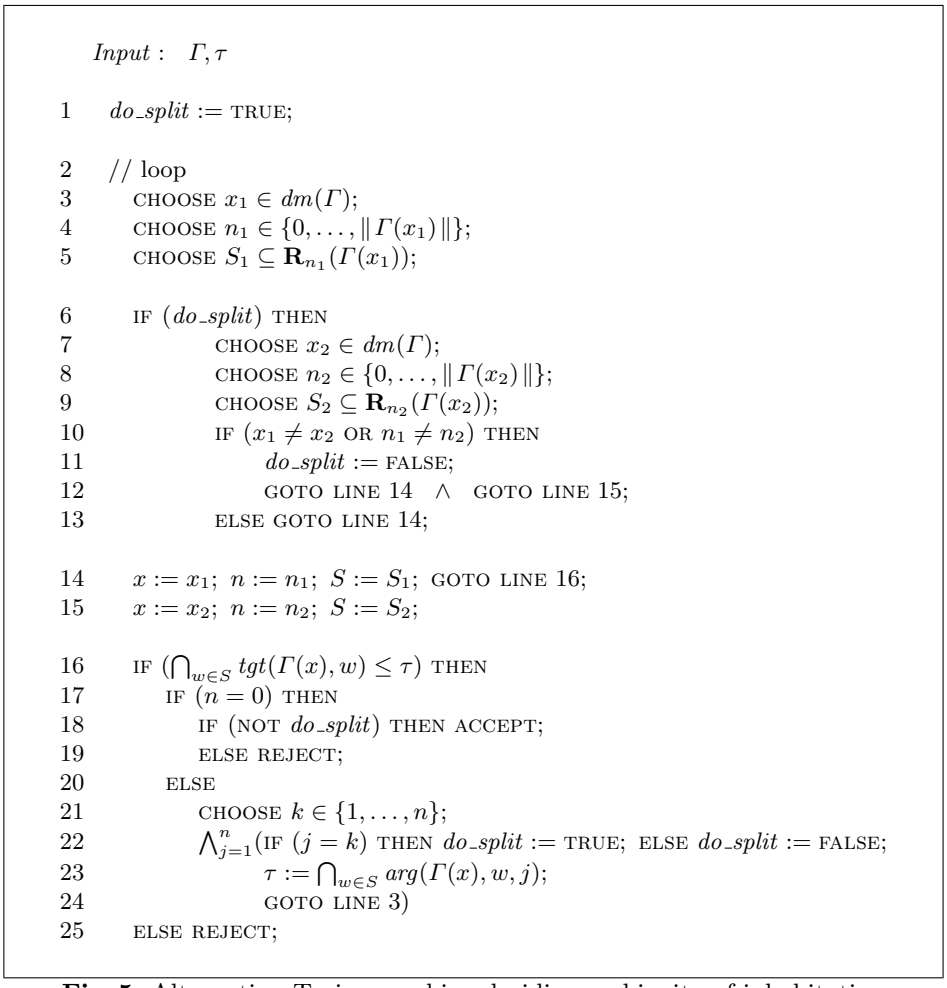

Fig. 5: Alternating Turing machine deciding ambiguity of inhabitation

Type judgements derivable by (P) are written as $\Gamma \vdash_P e : \tau$. Notice that rule (P) becomes an axiom scheme when $n = 0$.

Lemma 14. *$\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$ if and only if $\Gamma \vdash_P @_n(x, e_1, \ldots, e_n) : \tau$.*

Proof. Immediate by induction using Lemma 10. □

Clearly, a proof Π of $\Gamma \vdash_P e : \tau$ assigns a type to every subterm e' of e, applying rule (P), with some set S. If $e' = @_n(x, e_1, \ldots, e_n)$ then the triple (x, n, S) is called the *Π-stamp at e*. We say that a proof Π of $\Gamma \vdash_P e : \tau$ is *cyclic*, if there is a subterm e' of e and a proper subterm e'' of e' have the same Π-stamp.

Lemma 15. *Let Π be a proof of $\Gamma \vdash_P e : \tau$. If the depth of e exceeds $|\Gamma| \cdot 2^{|\Gamma|} \cdot |dm(\Gamma)|$ then the proof Π is cyclic.*

Lemma 16. *There are infinitely many inhabitants e with $\Gamma \vdash e : \tau$ if and only if there exists a cyclic proof of $\Gamma \vdash_P e' : \tau$ for some e'.*

Theorem 17. *In system* FCL$(\cap, \leq)$*, finiteness of inhabitation is in* EXPTIME.

Proof. Consider the algorithm shown in Figure 6. It is an alternating polynomial space Turing machine accepting Γ and σ if and only if there exists an inhabitant e with a cyclic proof of $\Gamma \vdash_P e : \sigma$. By Lemma 16, therefore, infiniteness of inhabitation is in EXPTIME for system FCL$(\cap, \leq)$, and hence finiteness of inhabitation is in EXPTIME. Full details can be found in [16]. □

Input : Γ, τ

```
1   count := 0;
2   inf := TRUE;

3   CHOOSE y ∈ dm(Γ);
4   CHOOSE m ∈ {0, ... ‖Γ(y)‖};
5   CHOOSE Q ⊆ R_m(Γ(y));

6   // loop
7       CHOOSE x ∈ dm(Γ);
8       CHOOSE n ∈ {0, ..., ‖Γ(x)‖};
9       CHOOSE S ⊆ R_n(Γ(x));

10      IF (inf AND (x, n, S) = (y, m, Q)) THEN count := count + 1;

11      IF (⋂_{w∈S} tgt(Γ(x), w) ≤ τ) THEN
12          IF (n = 0) THEN
13              IF (inf) THEN
14                  IF (count = 2) THEN ACCEPT ELSE REJECT;
15              ELSE ACCEPT
16          ELSE
17              CHOOSE k ∈ {1, ..., n};
18              ⋀_{j=1}^{n} (IF (j = k) THEN inf := TRUE ELSE inf := FALSE;
20                  τ := ⋂_{w∈S} arg(Γ(x), w, j);
21                  GOTO LINE 7)
22      ELSE REJECT;
```

Fig. 6: Alternating Turing machine deciding infiniteness of inhabitation

Acknowledgement The authors thank Boris Düdder and Moritz Martens for helpful comments.

References

1. H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.
2. Ch. Ben-Yelles. *Type Assignment in the Lambda-Calculus: Syntax and Semantics*. PhD thesis, Department of Pure Mathematics, University College of Swansea, September 1979.
3. M. Coppo and M. Dezani-Ciancaglini. An extension of basic functionality theory for lambda-calculus. *Notre Dame Journal of Formal Logic*, 21:685–693, 1980.
4. M. Dezani-Ciancaglini and R. Hindley. Intersection types for combinatory logic. *Theoretical Computer Science*, 100(2):303–324, 1992.
5. H. Comon et al. *Tree Automata Techniques and Applications*. Available online at `http://tata.gforge.inria.fr`, November 18, 2008.
6. J. R. Hindley. The simple semantics for Coppo-Dezani-Sallé types. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 212–226. Springer, 1982.
7. Naoki Kobayashi and C.-H. Luke Ong. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS*, pages 179–188. IEEE Computer Society, 2009.
8. M. Kozik. A finite set of functions with an EXPTIME-complete composition problem. *Theoretical Computer Science*, 407:330–341, 2008.
9. T. Kurata and M. Takahashi. Decidable properties of intersection type systems. In *TLCA*, volume 902 of *LNCS*, pages 297–311. Springer, 1995.
10. Dariusz Kuśmierek. The inhabitation problem for rank two intersection types. In *TLCA*, volume 4583 of *LNCS*, pages 240–254. Springer, 2007.
11. D. Leivant. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages*, pages 88–98. ACM, January 1983.
12. Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In *FOSSACS*, volume 5504 of *LNCS*, pages 395–409. Springer, 2009.
13. Mayur Naik and Jens Palsberg. A type system equivalent to a model checker. In *ESOP*, volume 3444 of *LNCS*, pages 374–388. Springer, 2005.
14. Ch. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
15. G. Pottinger. A type assignment for the strongly normalizable lambda-terms. In J. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 561–577. Academic Press, 1980.
16. J. Rehof and P. Urzyczyn. Finite combinatory logic with intersection types. Technical Report 834, Dept. of Computer Science, Technical University of Dortmund, `http://ls14-www.cs.tu-dortmund.de/index.php/Datei:TR-834.pdf`, 2011.
17. S. Salvati. Recognizability in the simply typed lambda-calculus. In *WoLLIC*, volume 5514 of *LNCS*, pages 48–60. Springer, 2009.
18. P. Urzyczyn. The emptiness problem for intersection types. *Journal of Symbolic Logic*, 64(3):1195–1215, 1999.
19. P. Urzyczyn. Inhabitation of low-rank intersection types. In *TLCA*, volume 5608 of *LNCS*, pages 356–370. Springer, 2009.
20. P. Urzyczyn. The logic of persistent intersection. *Fundamenta Informaticae*, 103:303–322, 2010.
21. B. Venneri. Intersection types as logical formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.
22. J. B. Wells and B. Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, volume 3573, pages 262–277. Springer, 2005.