technische universität
dortmund

department of
computer science

TECHNICAL REPORTS IN COMPUTER SCIENCE

Technische Universität Dortmund

Finite Combinatory Logic with Intersection Types (Extended Version)

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

Paweł Urzyczyn
University of Warsaw
Institute of Informatics
urzy@mimuw.edu.pl

(Partly supported by grant N N206 355836 from the Ministry of Science and Higher Education)

Number: 834
February 2011

Technische Universität Dortmund — Fakultät für Informatik
Otto-Hahn-Str. 14, 44227 Dortmund

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 consider the provability (inhabitation) problem for finite combinatory logic with intersection types and show that it 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.

CONTENTS

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.[1] 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 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.

In the remainder of this introduction we shall attempt to explain these perspectives in more detail.

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, 16] 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 [16]. Intersection types have also been used more generally to capture

1 In other parlance, the schematic viewpoint is sometimes referred to as *typical ambiguity*.

semantic properties of programs, for example in the context of model checking [14, 7]. Recently, 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 still, the borderline between decidability and undecidability was clarified in [11, 19] by rank restrictions (as defined in [12]), with inhabitation in rank 2 types being shown EXPSPACE-complete and undecidable from rank 3 and up.[2] 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

Combinatory logic is largely concerned with the completeness of a combinatory base, and, as mentioned above, combinatory logic with intersection types is complete with respect to the λ-calculus with intersection types. Finite combinatory logic is necessarily incomplete in this sense but decidable.

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 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., [23]). Most notably, however, finite combinatory logic is naturally applicable to the currently important synthesis problem from libraries of reusable components [13]. In contrast to the LTL-based approach of [13], which leads to a 2EXPTIME-complete synthesis problem, our approach is based on an entirely different specification format (types rather than temporal logic) and leads to an EXPTIME-complete problem, by reduction to alternation in polynomial space (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 Other restrictions have also been considered, which arise by limiting the logic of intersection [10, 20].

2 PRELIMINARIES

Trees

Let A and $\mathbf{L}$ be non-empty sets. A *labeled tree* over the *index set* 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 *at* w with domain

$$dm(T[w]) = \{w' \in \mathbf{A}^* \mid ww' \in dm(T)\}$$

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 in the usual way.

An intersection type $\tau \cap \sigma$ is said to have τ and σ as *components*. We sometimes write $\bigcap_{i=1}^{n} \tau_i$ for an intersection with n components ($n \geq 1$). Whenever convenient, we use the equivalent notation $\bigcap_{i \in I} \tau_i$, where I is an initial segment of $\mathbb{N}$. 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. Hence, any function type can be written as $\tau \to \bigcap_{i=1}^{n} A_i$.

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, with $\tau = \cap\{\alpha_1 \cap \alpha_2 \to \beta_1 \cap \beta_2\}$, 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 2.1 and is referred to as FCL$(\cap)$ (finite combinatory logic with intersection types). We consider the following decision problem:

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

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

Figure 2.1: Finite combinatory logic FCL$(\cap)$

Inhabitation problem for FCL$(\cap)$

Given an environment Γ *and a type* τ*, does there exist an applicative term* e *such that* $\Gamma \vdash e : \tau$ *?*

The difference between our type system and full combinatory logic based on intersection types [4, 22] 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 [22, 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, in the sense of the following proposition. This property, though simple, is obviously interesting with a view to applications in synthesis.

Proposition 3 (Unique specification) *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 $e \equiv e'$.*

Proof: For environments, Γ and Γ', define $\Gamma \oplus \Gamma'$, such that $dm(\Gamma \oplus \Gamma') = dm(\Gamma) \cup dm(\Gamma')$, as follows. If $x \in dm(\Gamma) \cap dm(\Gamma')$ then $(\Gamma \oplus \Gamma')(x) = \Gamma(x) \cap \Gamma'(x)$; otherwise let $(\Gamma \oplus \Gamma')(x)$ be $\Gamma(x)$ or $\Gamma'(x)$, depending on which one is defined. Define the environment Γ_e by induction:

$$\Gamma_{(x\, e_1 \ldots e_m)} \;=\; \{x : \alpha_{e_1} \to \cdots \to \alpha_{e_m} \to \alpha_{(x\, e_1 \ldots e_m)}\} \;\cap\; \bigoplus_{i=1}^{m} \Gamma_{e_i}\,,$$

where, for each e, the variable α_e is fresh. By induction with respect to $n \geq 0$, one proves that if $e = (x\, e_1 \ldots e_n)$ is a subterm of a term e_0 then $\Gamma_{e_0} \vdash e' : \alpha_e$ implies $e' \equiv e$. □

3 EXPTIME-COMPLETENESS OF INHABITATION

We show that the applicative inhabitation problem is EXPTIME-complete. We prove EXPTIME-hardness in Section 3.1 and membership in EXPTIME in Section 3.2. Our lower bound is seen to hold already for types of rank 2 (see definition below), by reduction from the intersection non-emptiness problem for finite bottom-up tree automata. An EXPTIME-lower bound follows easily from the known result (apparently going back to H. Friedman 1985 and published later by others, see [9]) 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. Since we find it instructive to observe that the problem (without subtyping) can be understood entirely within the theory of tree automata, we give here an alternative, simple reduction from the intersection non-emptiness problem for tree automata. This problem is known to be EXPTIME-complete by a proof method that generalizes the PSPACE-completeness proof of Kozen [8] for the intersection non-emptiness problem for finite word automata. Incidentally, Kozen used a similar method to show what might be considered a unary version of our problem. The proof for finite tree automata is given in detail in Veanes' report [21], although previous proof sketches by several other authors exist (see references in [21]).

Our upper bound proof again uses the theory of tree automata: We prove that the inhabitation problem is reducible to the non-emptiness problem for polynomial-sized alternating top-down tree automata [5]. Hence, we can show that the inhabitation problem is in APSPACE = EXPTIME. Representing the inhabitation problem by alternating tree automata, we can conclude that not only emptiness of inhabitation but 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. Let FCL stand for the $\cap$-free fragment of FCL($\cap$).

Proposition 4 *The inhabitation problem for* FCL *is solvable in polynomial time.*

Proof: Given Γ and τ, the algorithm systematically generates all types inhabited in Γ and stores them in memory. First, all types in $rn(\Gamma)$ are listed as inhabited. Then the following is repeated, until either τ is discovered or no more inhabited types can be generated:

- Select x with $\Gamma(x) = \rho_1 \to \cdots \to \rho_m \to \sigma$ such that types $\rho_1, \ldots, \rho_m$ are already listed, but σ is not. Add σ to the list.

The list only contains subterms of types from Γ, and the number of such terms is linear in the size of the input. The number of times ($\bullet$) is repeated is also linear, so the whole procedure can be completed in polynomial time. □

We prove EXPTIME-hardness of applicative inhabitation by reduction from intersection non-emptiness of finite bottom-up tree automata [21].

A *simple arrow* is a type of the form $\alpha_1 \to \cdots \to \alpha_n \to \alpha$, where $\alpha_1, \ldots, \alpha_n$ are type variables. A *simple intersection* is a type of form $\tau_1 \cap \ldots \cap \tau_m$, where $m \geq 1$ and each τ_i is either a variable or a simple arrow.

Lemma 5 (Generation) *Let $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$, where every type in Γ is a simple intersection. Then τ is a simple intersection with $\tau = \bigcap_{i=1}^{k} \tau^i$ (for some k, τ^i) and*

- $\Gamma(x) = \bigcap_{i=1}^{k}(\alpha_1^i \to \cdots \to \alpha_n^i \to \tau^i) \cap \sigma$, *for some σ;*
- $\Gamma \vdash e_j : \alpha_j^i$, *for $i = 1 \ldots k$, and $j = 1 \ldots n$.*

Proof: Induction with respect to $n \geq 0$.

For $n = 0$, it must be the case that τ arises from $\Gamma(x)$ by a (possibly empty) series of applications of $(\cap E)$ and $(\cap I)$. It follows from the assumption that all types in Γ are simple intersections that $\Gamma(x) = (\bigcap_{i \in I} \tau^i) \cap \sigma$ (for some I, σ) where the τ^i are simple arrows, and τ is a simple intersection of some subset of the τ^i.

For $n > 0$, the type derivation must involve a series of outermost applications of $(\to E)$, with premises $\Gamma \vdash (x\, e_1 \ldots e_{n-1}) : \alpha_n^i \to \tau^i$ and $\Gamma \vdash e_n : \alpha_n^i$ (for $i = 1 \ldots k$), and such that τ arises from the τ^i by a series of applications of $(\cap E)$ and $(\cap I)$, and where we have, by induction hypothesis (for some σ):

$$\Gamma(x) = \bigcap_{i=1}^{k}(\alpha_1^i \to \cdots \to \alpha_{n-1}^i \to \sigma^i) \cap \sigma$$

with $\sigma^i = \alpha_n^i \to \tau^i$ and $\Gamma \vdash e_j : \alpha_j^i$, for $i = 1, \ldots, k$ and $j = 1, \ldots, n-1$. The lemma now follows easily. □

Tree automata

We consider finite bottom-up tree automata as defined as in Veanes' report [21].

Assume that the sets of states $Q_1, \ldots, Q_k$ of automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$ are disjoint, with sets of final states $F_1, \ldots, F_n$. Elements of the union $Q_1 \cup \ldots \cup Q_n$ are used below as type variables, and function symbols (ranged over by x) in the signature of trees will serve as term variables. It is convenient to assume[1] that F_i are singletons, $F_i = \{s_i\}$.

Every "rule" R of the form $x(q_1, \ldots, q_m) \Rightarrow q$, where x is an m-ary function symbol, is encoded as type $\sigma_R = q_1 \to \cdots \to q_m \to q$. Define an environment Γ so that $\Gamma(x)$ is the intersection $\sigma_{R_1} \cap \ldots \cap \sigma_{R_d}$, where $R_1, \ldots, R_d$ are all the rules for x used in $\mathcal{A}_1, \ldots, \mathcal{A}_k$. The size of Γ is linear in the size of the input $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and all types in Γ are simple intersections. The *product automaton* $\mathcal{A}_1 \times \cdots \times \mathcal{A}_k$ is defined over tuples of states, written $\langle q^1, \ldots, q^k \rangle$, in the standard way, as is a *run* of an automaton. Notation $\theta(T)$ denotes the state assigned by run θ to the root of a tree T.

1 The assumption can be lifted: if $F_i = \{s_1^i, \ldots, s_{p_i}^i\}$ then we can take new type variables $s^1, \ldots, s^k$, and add to Γ a declaration $\$: \tau$, where $\$$ is fresh, and τ is an intersection of all types $s_j^i \to s^i$.

Lemma 6 *The following conditions are equivalent:*

a) There exist a tree T and a run θ of the product automaton $\mathcal{A}_1 \times \cdots \times \mathcal{A}_k$ such that $\theta(T) = \langle\, q^1, \ldots, q^k \,\rangle$.

b) Type $q^1 \cap \ldots \cap q^k$ is inhabited in Γ.

Proof: ($a \Rightarrow b$) Induction with respect to the size of T. The base case is when T consists of a single node labeled by a constant x. A run is possible when each automaton $\mathcal{A}_i$ has a rule of the form $x \Rightarrow q^i$. This means that $x : q^1 \cap \ldots \cap q^k \cap \sigma$ is in Γ, for some σ, and thus x inhabits the type $q^1 \cap \ldots \cap q^k$.

If T is of the form $x(T_1, \ldots, T_m)$, then we have m runs $\theta_1, \ldots, \theta_m$ on trees $T_1, \ldots, T_m$. Suppose that $\theta_j(T_j) = \langle\, p_j^1, \ldots, p_j^k \,\rangle$. This means that every $\mathcal{A}_i$ has a rule $x(p_1^i, \ldots, p_m^i) \Rightarrow q^i$, and that the type of x, declared in Γ, must contain all the components $p_1^i \to \cdots \to p_m^i \to q^i$. By the induction hypothesis, types $p_j^1 \cap \ldots \cap p_j^k$ are inhabited by some terms e_j, in particular we have $\Gamma \vdash e_j : p_j^i$, for each i. Therefore $(x\, e_1 \ldots e_m)$ has all types q^i in Γ.

($b \Rightarrow a$) Induction with respect to size of the inhabitant e. If e is an variable x, then it must be the case that $x : q^1 \cap \ldots \cap q^k \cap \sigma$, for some σ, i.e., there are appropriate "atomic rules" available, and a one node run must exist.

If $e = (x\, e_1 \ldots e_m)$, for some m, then we inspect the declaration of x in Γ. It consists of components of the form $\sigma_R = p_1 \to \cdots \to p_\ell \to q$, where ℓ is the arity of x as a function symbol. It follows from Lemma 5 that $\ell = m$ and there are rules R^i encoded by components of $\Gamma(x)$ of the form $\sigma_{R^i} = p_1^i \to \cdots \to p_\ell^i \to q^i$ ($i = 1, \ldots, k,\ j = 1, \ldots, m$). In addition, $\Gamma \vdash e_j : p_j^i$ must hold for all i, j. Since the sets of states of the automata are disjoint, rule R^i, i.e., rule $x(p_1^i, \ldots, p_m^i) \Rightarrow q^i$, must belong to the automaton $\mathcal{A}_i$.

By the induction hypothesis, we have some trees T_j and runs θ_j with $\theta_j(T_j) = \langle\, p_j^1, \ldots, p_j^k \,\rangle$. It follows that there is an appropriate run on $x(T_1, \ldots, T_m)$. □

Corollary 7 *Type $s^1 \cap \cdots \cap s^k$ is inhabited in Γ if and only if $\mathbf{L}(\mathcal{A}_1) \cap \ldots \cap \mathbf{L}(\mathcal{A}_k) \neq \emptyset$.*

Proof: Immediate from Lemma 6.[2] □

Definition 8 (Rank) Following [12], the rank of a type τ, denoted $\mathit{rank}(\tau)$, is defined by setting $\mathit{rank}(\tau) = 0$, if τ is a simple type, and otherwise

$$\mathit{rank}(\tau \to \sigma) = \max\{\mathit{rank}(\tau) + 1, \mathit{rank}(\sigma)\};$$

$$\mathit{rank}(\sigma \cap \tau) = \max\{1, \mathit{rank}(\sigma), \mathit{rank}(\tau)\}.$$

Assuming that Γ consists of declarations $x_i : \tau_i$, for $i = 1, \ldots, n$, the *rank* of a pair (Γ, τ) is the maximum of $\mathit{rank}(\tau)$ and $1 + \max\{\mathit{rank}(\tau_i) \mid i \leq n\}$, i.e., the same as $\mathit{rank}(\tau_1 \to \cdots \to \tau_n \to \tau)$.

Corollary 9 (EXPTIME-hardness)
The inhabitation problem for for FCL($\cap$) is EXPTIME-hard in rank 2.

Proof: Types used in the environment Γ are all of rank 1. Therefore, in Corollary 7, the rank of the pair $(\Gamma, s^1 \cap \cdots \cap s^k)$ is 2. □

2 If there are multiple final states, we observe that the only way to inhabit $s^1 \cap \cdots \cap s^k$ is by using the additional symbol \$, cf. footnote 1.

We show that the applicative inhabitation problem is in EXPTIME. Together with Corollary 9 we thereby prove the problem EXPTIME-complete. Our upper bound proof again uses tree automata, this time polynomial-sized top-down alternating tree automata. By known results for this class of automata, our algorithm immediately implies EXPTIME-decidability of all of the problems: emptiness, finiteness and uniqueness of inhabitation.

3.2.1 *Tree automaton*

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 10 (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 intended 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 3.1. 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 3.1, for $0 \leq n \leq \|\Gamma\|$.

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

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

Figure 3.1: Alternating tree automaton for $\textsc{fcl}(\cap)$

We use shorthand notation to specify members of $\mathcal{B}^+(Q \times \{0, \ldots, n\})$. For an index set $I = \{i_1, \ldots, i_m\}$, we write $\bigvee_{i \in I} \varphi(i)$ to denote the disjunction $\varphi(i_1) \vee \ldots \vee \varphi(i_m)$, and similarly for the notation $\bigwedge_{i \in I} \varphi(i)$.

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.

3.2.2 *Correctness*

In order to prove correctness of our representation in alternating tree automata from Section 3.2.1, we need a special generation lemma (Lemma 12), in which we assume the alternating representation of general types τ. We begin with a more standard version of generation. Recall that A, B range over types with no top level intersection.

Lemma 11 (Generation) *The assertion $\Gamma \vdash (e_1\ e_2) : A$ holds if and only if*

1. $\Gamma \vdash e_1 : (\tau \to \tau')$;
2. $\Gamma \vdash e_2 : \tau$,

for some τ, τ' and some σ with $\tau' = A \cap \sigma$.

Proof: Since A is not an intersection, we can assume that the derivation ends in an application of $(\to E)$ possibly followed by applications of $(\cap E)$. The lemma follows immediately, since it states the most general conditions under which such an application of $(\to E)$ is possible. □

The following characterizes type derivations in terms of alternating right addresses (paths).

Lemma 12 (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 goes by induction with respect to $n \geq 0$. For $n = 0$, we have $\Gamma \vdash x : A$. Hence, for some τ we have $x : \tau \in \Gamma$ with $\tau[i_0] = A$ for some $i_0 \in \mathbb{N}$. Take $w = i_0$, and conditions (1–2) hold.

For the induction step let $\Gamma \vdash ((x\, e_1 \ldots e_n)\ e_{n+1}) : A$. By Lemma 11 this is equivalent to

- $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau \to \tau'$;
- $\Gamma \vdash e_{n+1} : \tau$;
- $\tau' = A \cap \rho$,

for some τ, τ', ρ. Clearly, we have $A = \tau'[j_0]$, for some i_0 and j_0. By the induction hypothesis, $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau \to \tau'$ implies $\tau \to \tau' = tgt(\Gamma(x), w_n) = \Gamma(x)[w_n]$, and $\Gamma \vdash e_j : arg(\Gamma(x), w_n, j)$, for all $j \in \{1, \ldots, n\}$. Now take $w_{n+1} = w_n r j_0$. Then w_{n+1} is a right address, since w_n is one. Moreover, we have $A = \tau'[j_0] = (\tau \to \tau')[rj_0] = \Gamma(x)[w_n][rj_0] = \Gamma(x)[w_{n+1}] = tgt(\Gamma(x), w_{n+1})$, as desired. Finally, we have

$$\Gamma(x)[L(n+1, w_{n+1})] = \Gamma(x)[(w_{n+1} \downarrow n+1)l] = \Gamma(x)[w_n l] = (\tau \to \tau')[l] = \tau,$$

whence $\Gamma \vdash e_{n+1} : arg(\Gamma(x), w_{n+1}, n+1)$. □

Proposition 13 *$\Gamma \vdash e : \tau$ 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, for $j = 1, \ldots, n$. Let $\tau = \bigcap_{i=1}^{k} A_i$. By Lemma 12, we have $\Gamma \vdash e : \tau$ if and only if [3]

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

The automaton $\mathcal{A}(\Gamma, \tau)$ is evidently 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 14 *In system* FCL($\cap$) *the problems*

1. *emptiness of inhabitation*
2. *finiteness of inhabitation*
3. *uniqueness of inhabitation*

are solvable in EXPTIME.

3 Quantifier ranges omitted for simplicity.

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 Proposítion 13 together with the polynomial time solvability of the emptiness problem (see [5], Theorem 1.7.4), of the finiteness problem (see [5], Theorem 1.7.6), the uniqueness problem (see [5], Theorem 1.7.10) for deterministic tree automata and the fact that alternation can be transformed away modulo an exponential blow-up in the size of the automaton (see [5], Theorem 7.4.1). □

Corollary 15 *In system* FCL$(\cap)$ *the inhabitation problem is* EXPTIME*-complete.*

Proof: By Corollary 9 and Theorem 14. □

4 SUBTYPING

In this section we prove that the inhabitation problem remains EXPTIME-complete in the presence of subtyping (Section 4.2); for the problems of uniqueness (Section 4.3) and finiteness (Section 4.4) of inhabitation we are also able to show membership 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 special atom ω and stratify the type language as follows. Type variables are ranged over by α, β. *Atoms*, ranged over by a, b are defined by

$$a ::= \alpha \mid \omega$$

Function types, ranged over by A, B are defined by

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

General types, ranged over by τ, σ are defined by

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

The inequality $\leq$ is defined as the least preorder (reflexive and transitive relation), satisfying:

$$\sigma \leq \omega, \quad \omega \leq \omega \to \omega, \quad \sigma \cap \tau \leq \sigma, \quad \sigma \cap \tau \leq \tau, \quad \sigma \leq \sigma \cap \sigma;$$

$$(\sigma \to \tau) \cap (\sigma \to \rho) \leq \sigma \to \tau \cap \rho;$$

If $\sigma \leq \sigma'$ and $\tau \leq \tau'$ then $\sigma \cap \tau \leq \sigma' \cap \tau'$ 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 been able to find is the decidability result of [6] from 1982. The key characterization in [6] (also to be found in [10, 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, essentially uses expansion of types under the distributive axiom $A \to (B_1 \cap B_2) = (A \to B_1) \cap (A \to B_2)$, applied left to right, which may incur an exponential blow-up in the size of normalized types. Hence, a direct algorithm based on the characterization would lead to an exponential procedure.

The PTIME algorithm is based on the following property of subtyping, sometimes called *beta-soundness*, and probably first stated in [1]. Note that the converse is trivially true.

Lemma 16 *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 $\bigcap\{\tau_i \mid \sigma \leq \sigma_i\} \leq \tau$.*

Proof: Induction with respect to the definition of $\leq$. In the case of transitivity note that every type is an intersection of arrows and atoms. □

It is not too difficult to construct a PSPACE-algorithm by a direct recursive implementation of Lemma 16. We can however do better using memoization techniques in a combined top-down and bottom-up processing of type trees.

Polynomial-time algorithm

Let us fix types τ_0 and σ_0. We are asking if $\tau_0 \leq \sigma_0$. Recall that we identify types τ and σ with alternating trees whose leaves are labeled by atoms and internal nodes are labeled by $\cap$ or $\to$. The internal odd-level nodes are labeled by arrows, and have two children each. The even-level nodes are labeled by $\cap$, and can have an arbitrary number of children, possibly one. The root is at level zero and it is therefore a $\cap$-node.

The obvious idea is that to verify the inequality $\tau_0 \leq \sigma_0$ we may have to ask if $\tau \leq \sigma$ or if $\tau \geq \sigma$, where τ and σ are subterms of τ_0 and σ_0, respectively. In this case, τ and σ are represented by $\cap$-nodes at the same level in the corresponding trees. The number of such pairs is quadratic, but that is not all. For instance, to check if $\tau \leq \sigma$ we may have to ask questions of the form $\bigcap_i \tau_i \leq \sigma$, where τ_i are $\cap$-successors of τ. The problem is that the total number of such questions is exponential.

Theorem 17 *The subtyping relation $\leq$ is in* PTIME.

Proof: We describe a procedure for answering *simple questions at level k* of the form $\tau \leq \sigma$ or $\tau \geq \sigma$ (where τ and σ are at level k) and *set questions at level k* of the form $\bigcap_i \tau_i \leq \sigma$ or $\tau \geq \bigcap_i \sigma_i$ (where τ_i and σ_i are at level $k+1$). We introduce memoization, where it is assumed that when a question at level k is considered, all simple questions (but not all set questions) at levels $k+1$ and more have been already answered and the answers have been recorded.

Simple questions: 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.

Set questions: Take a set question $\bigcap_i \tau_i \leq \sigma$, and observe that types τ_i are intersections. Write the left-hand side as $L = \bigcap_{j\in J} a_j \cap \bigcap_{i\in I}(\varrho_i \to \mu_i)$. To decide

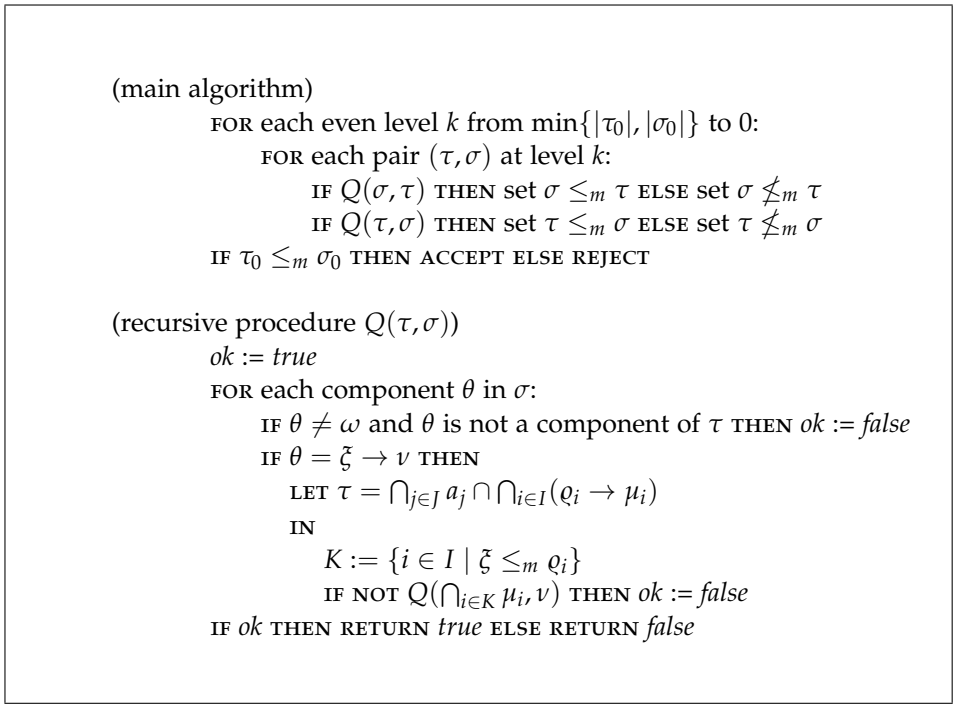

Figure 4.1: PTIME procedure for deciding the relation $\leq$. Notation $\tau \leq_m \sigma$ stands for the memoized relation between nodes τ and σ.

if $L \leq \theta$, where θ is a component of σ, we proceed as in the simple case, except that the result is not memoized. The algorithm is summarized in Figure 4.1.

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 nodes in S, all of them exactly two levels below the original target. Further recursive calls will address targets at lower even levels, 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 an $\mathcal{O}(n^4)$ total time. □

4.2 INHABITATION WITH SUBTYPING

We extend the system FCL($\cap$) with subtyping. The resulting system is denoted FCL($\cap, \leq$) (finite combinatory logic with intersection types and subtyping) and is shown in Figure 4.2. (Normally, the axiom $\Gamma \vdash e : \omega$ is added together with the type ω. We leave it out for simplicity, since it has no interesting consequences for the theory developed in this paper.)

We now prove that inhabitation remains EXPTIME-complete in the presence of subtyping. We first show the upper bound, which is the main part of our proof. Before stating our algorithm witnessing the upper bound, we need a path characterization of typings.

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

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

Figure 4.2: FCL$(\cap, \leq)$

Lemma 18 *For any τ and $w \in \mathbf{R}_n(\tau)$ one has:*

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

Proof: By induction on $n \geq 0$. Assume that $\tau = \bigcap_{i\in I}(\sigma_i \to \sigma_i') \cap \bigcap_{j\in J} a_j$.

If $n = 0$ then $tgt(\tau, w)$ is in the set $\{\sigma_i \to \sigma_i' \mid i \in I\} \cup \{a_j \mid j \in J\}$ and the claim follows from the axiom $\tau_1 \cap \tau_2 \leq \tau_i$.

For $n > 0$, we have $w = i_0 r w'$, for some $i_0 \in I$, and $w' \in \mathbf{R}_{n-1}(\tau[i_0 r])$. Then $arg(\tau, w, 1) = \sigma_{i_0}$, and $arg(\tau, w, j) = arg(\sigma_{i_0}', w', j-1)$, for $j = 2, \ldots, n$, and $tgt(\tau, w) = tgt(\sigma_{i_0}', w')$. By the induction hypothesis, we have

$$\sigma_{i_0}' \leq arg(\sigma_{i_0}', w', 1) \to \cdots \to arg(\sigma_{i_0}', w', n-1) \to tgt(\sigma_{i_0}', w')$$

and hence also

$$\sigma_{i_0} \to \sigma_{i_0}' \leq \sigma_{i_0} \to arg(\sigma_{i_0}', w', 1) \to \cdots \to arg(\sigma_{i_0}', w', n-1) \to tgt(\sigma_{i_0}', w')$$

which shows the claim. □

Lemma 19 (Path lemma) *$\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$ if and only if there exists a subset S of $\mathbf{R}_n(\Gamma(x))$ such that*

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

Proof: $(\Rightarrow)$ Induction with respect to the derivation of $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau$. If the last rule is (var) then $n = 0$ and $\Gamma(x) = \tau$. Write $\tau = (\bigcap_{i\in I} \sigma_i \to \sigma_i') \cap \bigcap_{j\in J} a_j$ and choose $S = I \cup J$.

If the last rule of the derivation is $(\leq)$, then the claim follows easily by induction hypothesis and transitivity of $(\leq)$.

If the last rule is $(\cap I)$, then $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau_1$ and $\Gamma \vdash (x\, e_1 \ldots e_n) : \tau_2$, with $\tau_1 \cap \tau_2 = \tau$. By the induction hypothesis, there are $S_1, S_2 \subseteq \mathbf{R}_n(\Gamma(x))$ with $\Gamma \vdash e_j : \bigcap_{w\in S_i} arg(\Gamma(x), w, j)$ and $\bigcap_{w\in S_i} tgt(\Gamma(x), w) \leq \tau_i$, for $i = 1, 2$ and $j = 1, \ldots, n$. Let $S = S_1 \cup S_2$. Then it follows by $(\cap I)$ that $\Gamma \vdash e_j : \bigcap_{w\in S} arg(\Gamma(x), w, j)$, for $j = 1 \ldots n$, and by monotonicity of $\cap$ that $\bigcap_{w\in S} tgt(\Gamma(x), w) \leq \tau_1 \cap \tau_2 = \tau$, as desired.

If the last rule of the derivation is $(\to E)$, then we have

$$\Gamma \vdash (x\, e_1 \ldots e_{n-1}) : \tau' \to \tau \text{ and } \Gamma \vdash e_n : \tau',$$

for some τ'. By the induction hypothesis there exists $S_{n-1} \subseteq \mathbf{R}_{n-1}(\Gamma(x))$ such that

(I1) $\Gamma \vdash e_j : \bigcap_{w \in S_{n-1}} arg(\Gamma(x), w, j)$ for $j = 1 \ldots n-1$;

(I2) $\bigcap_{w \in S_{n-1}} tgt(\Gamma(x), w) \leq \tau' \to \tau$.

Let

$$P = \{w \in S_{n-1} \mid tgt(\Gamma(x), w) = \sigma \to \sigma' \text{ for some } \sigma, \sigma' \text{ with } \tau' \leq \sigma\}.$$

For $p \in P$, write $tgt(\Gamma(x), p) = \sigma_p \to \sigma'_p$. By (I2) and Lemma 16, we have $P \neq \emptyset$ and

$$\tau' \leq \bigcap_{p \in P} \sigma_p \quad \text{and} \quad \bigcap_{p \in P} \sigma'_p \leq \tau.$$

Write $\sigma'_p = \bigcap_{i_p \in I_p} A_{i_p}$ for suitable I_p. Now define the set of addresses

$$S = \{pri_p \mid p \in P, i_p \in I_p\}.$$

Then

$$\bigcap_{w \in S} arg(\Gamma(x), w, n) = \bigcap_{p \in P} \sigma_p \quad \text{and} \quad \bigcap_{w \in S} tgt(\Gamma(x), w) = \bigcap_{p \in P} \sigma'_p .$$

Since $\Gamma \vdash e_n : \tau'$, we thereby have $\Gamma \vdash e_n : \bigcap_{w \in S} arg(\Gamma(x), w, n)$ by rule $(\leq)$, and also $\bigcap_{w \in S} tgt(\Gamma(x), w) \leq \tau$, as desired.

($\Leftarrow$) Observe that by Lemma 18, we have

$$\Gamma(x) \leq arg(\tau, w, 1) \to \cdots \to arg(\tau, w, n) \to tgt(\tau, w),$$

for $w \in \mathbf{R}_n(\Gamma(x))$. The claim now follows from the assumption together with obvious applications of the type rules. □

Tree automaton

Using Lemma 19 it is possible to specify an alternating tree automaton $\mathcal{A}(\Gamma, \tau)$ by making a simple modification of the automaton shown in Figure 3.1 (Section 3.2.1) and with the corresponding property for system $\text{FCL}(\cap, \leq)$:

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

It is instructive to consider this simple solution. First, relying on Lemma 19, we can no longer restrict ourselves to choosing a single alternating right-address $w \in \mathbf{R}_n(\Gamma(x))$, as is done from the nondeterministic state $\bigvee_{w \in \mathbf{R}_n(\Gamma(x))}(\ldots)$ in the automaton of Figure 3.1. Instead, Lemma 19 requires that we choose a *subset* S of right-addresses, which can be done by modifying the nondeterministic state to $\bigvee_{S \subseteq \mathbf{R}_n(\Gamma(x))}(\ldots)$. Secondly, we modify the states $\text{SRC}(arg(\Gamma(x), w, j))$ and $\text{CHK}(A_i, tgt(\Gamma(x), w), x)$ such that the set S is taken into account, resulting in the states $\text{SRC}(\bigcap_{w \in S} arg(\Gamma(x), w, j))$ and $\text{CHK}(A_i, \bigcap_{w \in S} tgt(\Gamma(x), w), x)$, respectively. It is not difficult, using Lemma 19, to see that the resulting alternating tree automaton satisfies the property mentioned above.

The problem with this solution is that the set of states in the disjunction $\bigvee_{S \subseteq \mathbf{R}_n(\Gamma(x))}(\ldots)$, 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 14). As we will now proceed to show, we can do better, by passing to a Turing machine representation. However, together with Theorem 14 the reduction to tree automata implies:

Corollary 20 *In system* $\textsc{fcl}(\cap)$ *and* $\textsc{fcl}(\cap, \leq)$ *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 on the inhabitation problem for $\textsc{fcl}(\cap, \leq)$ by a direct implementation of the algorithm implicit in Lemma 19 on a Turing machine. Notice that we cannot apply the same approach as in Section 3.2.1, 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, the 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 the choice of the set $S \subseteq \mathbf{R}_n(\Gamma(x))$ is done using the tape of the Turing machine as stateful intermediate storage. By overwriting the tape, storing S upon each nondeterministic choice of S (which is linear in the input), exponentially many states (which are explicit in the tree automaton) can be explored under a polynomial space bound.

The definition shown in Figure 4.3 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. [15, 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 4.3 directly implements Lemma 19. We use shorthand notation for existential states (choose $\ldots$) and universal states ($\bigwedge \ldots$), where the branching factors depend on parameters of the input (Figure 4.3, 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 as a subroutine, relying on Lemma 17.

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).

Input : Γ, τ

1 // loop
2 CHOOSE $x \in dm(\Gamma)$;
3 CHOOSE $n \in \{0, \ldots, \|\Gamma(x)\|\}$;
4 CHOOSE $S \subseteq \mathbf{R}_n(\Gamma(x))$;

5 IF $(\bigcap_{w \in S} tgt(\Gamma(x), w) \leq \tau)$ THEN
6 IF $(n = 0)$ THEN ACCEPT;
7 ELSE
8 $\bigwedge_{j=1}^{n}\ \tau := \bigcap_{w \in S} arg(\Gamma(x), w, j)$;
9 GOTO LINE 2;
10 ELSE REJECT;

Figure 4.3: Alternating Turing machine $\mathcal{M}$ deciding inhabitation for FCL$(\cap, \leq)$

Theorem 21 *The inhabitation problem in system* FCL$(\cap, \leq)$ *is* EXPTIME*-complete.*

Proof: Membership in EXPTIME follows from the fact that the Turing machine $\mathcal{M}$ shown in Figure 4.3 solves the inhabitation problem in alternating polynomial space. Correctness (the machine accepts (Γ, τ) iff $\Gamma \vdash e : \tau$ for some e) follows easily from Lemma 19 (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 17 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.

To see that the inhabitation problem remains EXPTIME-hard in the presence of subtyping, we prove the same property as stated in Lemma 6 using the same reduction. The implication $(a \Rightarrow b)$ from Lemma 6 obviously remains valid, so we only need to consider the implication $(b \Rightarrow a)$. In order to prove this, we use Lemma 19, as follows. Suppose that $\Gamma \vdash (x\, e_1 \ldots e_n) : \bigcap_{i=1}^{k} q_i$. By construction of Γ, we can write

$$\Gamma(x) = \bigcap_{h \in H} p_h^1 \to \cdots \to p_h^\ell \to q_h, \quad \text{for some } H. \tag{4.1}$$

By Lemma 19, there exist m and $S \subseteq \mathbf{R}_m(\Gamma(x))$ such that

$$\bigcap_{w \in S} tgt(\Gamma(x), w) \quad \leq \quad \bigcap_{i=1}^{k} q_i \tag{4.2}$$

and

$$\Gamma \vdash e_j : \bigcap_{w \in S} arg(\Gamma(x), w, j), \quad \text{for } \ j = 1, \ldots, m. \tag{4.3}$$

By (4.2), the form of $\Gamma(x)$ according to (4.1), and Lemma 16, we must have $\ell = m$ and, for all $i = 1 \ldots k$, there are $w_i \in S$ with $q_i = tgt(\Gamma(x), w_i)$. Moreover, it follows from (4.3) that $\Gamma \vdash e_j : p^j_{h_i}$ for $i = 1, \ldots, k$, and $j = 1, \ldots, m$, with $arg(\Gamma(x), w_i, j) = p^j_{h_i}$. Hence, there are rules R^i encoded by types of the form $\sigma_{R^i} = p^1_{h_i} \to \cdots \to p^\ell_{h_i} \to q_i$, with $h_i \in H$. This shows that the inductive argument of the implication $(b \Rightarrow a)$ from Lemma 6 can be repeated in the presence of subtyping. □

4.3 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 4.4 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 4.3) as well as to search for two different inhabitants.

Theorem 22 *In system* FCL$(\cap, \leq)$*, uniqueness of inhabitation is in* EXPTIME*.*

Proof: The alternating Turing machine shown in Figure 4.4 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. The idea is that if two inhabitants exist, then there must be a topmost node in the syntax trees of the inhabitants where they differ, either by having different variables in the operator position (as in $@_n(x \ldots)$ vs. $@_n(y \ldots)$) or by having operators with different arities (as in $@_n(x \ldots)$ vs. $@_m(x \ldots)$, $n \neq m$). It is sufficient to require that such a split should exist in at least one universal branch of the computation. The boolean variable *do_split* acts as a "hot potato" which forces at least one universal branch accepting the input to "split" off, constructing a second, different inhabitant. The universal branch manifesting ambiguity is chosen nondeterministically (lines 21–22), and the construction of a second inhabitant happens in lines 6–13. The split can happen nondeterminitically at any point. The nondeterministic choice (lines 7–9) fixes values for a candidate alternative inhabitant (given by x_2, n_2, S_2). These values might not constitute a split (if $x_1 = x_2$ and $n_1 = n_2$, line 10), in which case computation proceeds without a split at this point. If, on the other hand, a candidate second inhabitant is guessed (lines 7–9) exhibiting a difference (line 10), the obligation to perform further splits is revoked (line 11) and the machine makes a universal transition (line 12) to two states (lines 14 and 15), in which the variables x, n and S are initialized accordingly with two sets of values, x_1, n_1, S_1 and x_2, n_2, S_2. If a leaf node is reached on a branch of computation where the obligation to split has been revoked (line 18, the condition *do_split* = FALSE indicating that the obligation has been met), the machine accepts. Universal branches are also used (lines 22–23) to search for inhabitants of argument types, as usual. □

```
Input :  Γ, τ

1    do_split := TRUE;

2    // loop
3      CHOOSE x1 ∈ dm(Γ);
4      CHOOSE n1 ∈ {0, ..., ‖Γ(x1)‖};
5      CHOOSE S1 ⊆ R_n1(Γ(x1));

6      IF (do_split) THEN
7            CHOOSE x2 ∈ dm(Γ);
8            CHOOSE n2 ∈ {0, ..., ‖Γ(x2)‖};
9            CHOOSE S2 ⊆ R_n2(Γ(x2));
10           IF (x1 ≠ x2 OR n1 ≠ n2) THEN
11               do_split := FALSE;
12               GOTO LINE 14  ∧  GOTO LINE 15;
13           ELSE GOTO LINE 14;

14     x := x1; n := n1; S := S1; GOTO LINE 16;
15     x := x2; n := n2; S := S2;

16     IF (⋂_{w∈S} tgt(Γ(x), w) ≤ τ) THEN
17       IF (n = 0) THEN
18         IF (NOT do_split) THEN ACCEPT;
19         ELSE REJECT;
20       ELSE
21         CHOOSE k ∈ {1, ..., n};
22         ⋀_{j=1}^{n} (IF (j = k) THEN do_split := TRUE; ELSE do_split := FALSE;
23               τ := ⋂_{w∈S} arg(Γ(x), w, j);
24               GOTO LINE 3)
25     ELSE REJECT;
```

Figure 4.4: Alternating Turing machine deciding ambiguity of inhabitation

4.4 FINITENESS OF INHABITATION WITH SUBTYPING

We prove that finiteness of inhabitation in system FCL$(\cap, \leq)$ is in EXPTIME. The idea is to define a notion of cyclic type proofs such that infinitely many inhabitants exist for Γ, τ in system FCL$(\cap, \leq)$ if and only if there exists a cyclic proof for Γ, τ. We then define an alternating polynomial space Turing machine accepting Γ, τ if and only if there exists a cyclic proof for Γ, τ.

Lemma 19 (Path Lemma) yields an alternative proof system for FCL$(\cap, \leq)$, consisting only of the following rule schema (P), where $S \subseteq \mathbf{R}_n(\Gamma(x))$:

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

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

Proof: Immediate by induction using Lemma 19. □

The following definition introduces an analogue of Church-style terms.

Definition 24 (Decorated term) Let Π denote a proof of $\Gamma \vdash_P e : \tau$. The *decorated term* associated with Π, denoted $T_\Pi(e)$, is the term arising from e by labeling each application node $@_n$ (a root of a subterm e' of e) with the set S used in Π in the unique application of rule (P) where e' is the subject in the conclusion. Leaf nodes x are decorated with $\Gamma(x)$. For example, a decoration of the term $e = @_n(x, e_1, \ldots, e_n)$ could be $@_n^S(x^{\alpha_1 \to \cdots \to \alpha_n \to \alpha}, \overline{e}_1, \ldots, \overline{e}_n)$, where $S = \{1r \cdots 1r1\}$ (n occurrences of r), and where the $\overline{e}_i$ are appropriate decorations of e_i.

Definition 25 (Cyclic proof) Let Π denote a proof of $\Gamma \vdash_P e : \tau$. We say that Π is *cyclic* if there exists an address $w \in dm(T_\Pi(e))$ and a proper prefix w' of w such that the label S and the type of the subterm at node $T_\Pi(e)(w)$ are the same as the label and type at node $T_\Pi(e)(w')$. For example, the decorated term

$$@_1^{\{1r1\}}(x^{\alpha \to \alpha}, @_1^{\{1r1\}}(z^{\alpha \to \alpha}, y^\alpha))$$

arises from a cyclic proof with witnessing address $w = 1$ and prefix $w' = \varepsilon$.

Lemma 26 *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.*

Proof: Consider an address w in $dm(e)$ longer than $|\Gamma| \cdot 2^{|\Gamma|} \cdot |dm(\Gamma)|$. Consider the operators occurring along the address w. Since there are at most $|dm(\Gamma)|$ distinct variables in Γ and hence also at most that many distinct variables in e, there must be a variable $x \in dm(\Gamma)$ such that x occurs at least $|\Gamma| \cdot 2^{|\Gamma|}$ times along w. Pick such an x and denote the distinct operator occurrences of x along w as x_i, for $i = 1 \ldots |\Gamma| \cdot 2^{|\Gamma|}$. Each parent node $@_{n_i}$ of the x_i in $T_\Pi(e)$ is decorated with a set S_i. Since there are at most $|\Gamma| \cdot 2^{|\Gamma|}$ distinct such sets S_i, it follows that, for at least two distinct indices $p, q \in \{1, \ldots, |\Gamma| \cdot 2^{|\Gamma|}\}$ it must be that $S_p = S_q$, and hence Π is cyclic. □

Lemma 27 *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'.*

Proof: Assume there are infinitely many inhabitants e with $\Gamma \vdash e : \tau$. According to Lemma 23, for each such e, we have $\Gamma \vdash_P e : \tau$. Since there are infinitely many such e, there are inhabitants of arbitrary depth. Lemma 26 then implies that, for sufficiently deep inhabitants e', the proofs of $\Gamma \vdash_P e' : \tau$ are cyclic.

Assume there exists an inhabitant e' with a cyclic proof of $\Gamma \vdash_P e' : \tau$. Since a cyclic proof can be arbitrarily "pumped" by substituting the topmost node of a circle for its repeating node, there must be infinitely many e such that $\Gamma \vdash_P e : \tau$. By Lemma 23, there are therefore infinitely many inhabitants e with $\Gamma \vdash e : \tau$. □

Theorem 28 *In system* FCL$(\cap, \leq)$*, finiteness of inhabitation is in* EXPTIME

Proof: Consider the algorithm shown in Figure 4.5. 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 27, therefore, infiniteness of inhabitation is in EXPTIME for system FCL$(\cap, \leq)$, and hence finiteness of inhabitation is in EXPTIME.

The machine works by nondeterministically choosing a "pivot" up front (lines 3–5), in the form of a variable y and a set $Q \subseteq \mathbf{R}_m(\Gamma(y)$ which will be the witness of a cyclic proof in accordance with Definition 25. It then counts (using variable *count*) how many times the chosen variable y and set Q are generated in the subsequent choice of an inhabitant, aiming for a repetition ($count = 2$, line 14). The variable *inf* is used to control nondeterministically along which universal branches a cycle will be discovered. □

```
      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;
```

Figure 4.5: Alternating Turing machine deciding infiniteness of inhabitation

ACKNOWLEDGMENTS

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

BIBLIOGRAPHY

[1] Barendregt, H., Coppo, M., and Dezani-Ciancaglini, M. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic 48*, 4 (1983), 931–940.

[2] Ben-Yelles, C. *Type Assignment in the Lambda-Calculus: Syntax and Semantics*. PhD thesis, Department of Pure Mathematics, University College of Swansea, September 1979.

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

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

[5] et al., H. C. *Tree Automata Techniques and Applications*. Available online at `http://tata.gforge.inria.fr`, November 18, 2008.

[6] Hindley, J. R. The simple semantics for Coppo-Dezani-Sallé types. In *International Symposium on Programming* (1982), vol. 137 of *LNCS*, Springer, pp. 212–226.

[7] Kobayashi, N., and Ong, C.-H. L. A type system equivalent to the modal mu-calculus model checking of higher-order recursion schemes. In *LICS* (2009), IEEE Computer Society, pp. 179–188.

[8] Kozen, D. Lower bounds for natural proof systems. In *Symposium on Foundations of Computer Science (FOCS)* (1977), IEEE, pp. 254–266.

[9] Kozik, M. A finite set of functions with an exptime-complete composition problem. *Theoretical Computer Science 407* (2008), 330–341.

[10] Kurata, T., and Takahashi, M. Decidable properties of intersection type systems. In *TLCA* (1995), vol. 902 of *LNCS*, Springer, pp. 297–311.

[11] Kuśmierek, D. The inhabitation problem for rank two intersection types. In *TLCA* (2007), vol. 4583 of *LNCS*, Springer, pp. 240–254.

[12] Leivant, D. Polymorphic type inference. In *Proc. 10th ACM Symp. on Principles of Programming Languages* (January 1983), ACM, pp. 88–98.

[13] Lustig, Y., and Vardi, M. Y. Synthesis from component libraries. In *FOSSACS* (2009), vol. 5504 of *LNCS*, Springer, pp. 395–409.

[14] Naik, M., and Palsberg, J. A type system equivalent to a model checker. In *ESOP* (2005), vol. 3444 of *LNCS*, Springer, pp. 374–388.

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

[16] Pottinger, G. A type assignment for the strongly normalizable lambda-terms. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Hindley and J. Seldin, Eds. Academic Press, 1980, pp. 561–577.

[17] Salvati, S. Recognizability in the simply typed lambda-calculus. In *WoLLIC* (2009), vol. 5514 of *LNCS*, Springer, pp. 48–60.

[18] Urzyczyn, P. The emptiness problem for intersection types. *Journal of Symbolic Logic 64*, 3 (1999), 1195–1215.

[19] Urzyczyn, P. Inhabitation of low-rank intersection types. In *TLCA* (2009), vol. 5608 of *LNCS*, Springer, pp. 356–370.

[20] Urzyczyn, P. The logic of persistent intersection. *Fundamenta Informaticae 103* (2010), 303–322.

[21] Veanes, M. On computational complexity of basic decision problems of finite tree automata. Tech. Rep. 133, Uppsala University, Computer Science Department, `research.microsoft.com/pubs/78382/upmail0133.pdf`, 1997.

[22] Venneri, B. Intersection types as logical formulae. *Journal of Logic and Computation 4*, 2 (1994), 109–124.

[23] Wells, J. B., and Yakobowski, B. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR* (2005), vol. 3573, Springer, pp. 262–277.

Forschungsberichte
der Fakultät für Informatik
der Technischen Universität Dortmund

ISSN 0933-6192

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