The Inhabitation Problem for Rank Two Intersection Types[⋆]

Dariusz Kuśmierek

Warsaw University, Institute of Informatics
Banacha 2, 02-097 Warsaw, Poland
daku@mimuw.edu.pl

Abstract. We prove that the inhabitation problem for rank two intersection types is decidable, but (contrary to a common belief) EXPTIME-hard. The exponential time hardness is shown by reduction from the in-place acceptance problem for alternating Turing machines.

Keywords: lambda calculus, intersection types, type inhabitation problem, alternating Turing machine.

Introduction

Type inhabitation problem is usually defined as follows: "does there exist a closed term T of a given type τ (in an empty environment)".

In the simply typed system the inhabitation problem is PSPACE-complete (see [7]). The intersection types system studied in the current paper involves types of the form $\alpha \cap \beta$. Intuitively, a term can be assigned the type $\alpha \cap \beta$ if and only if it can be assigned the type α and also the type β.

Undecidability of the general inhabitation problem for intersection types was shown by P. Urzyczyn in [8].

Several weakened systems were studied, and proved to be decidable. T. Kurata and M. Takahashi in [2] proved the decidability of the problem in the system $\lambda(E\cap, \leq)$ which does not use the rule $(I\cap)$.

D. Leivant in [3] defines the rank of an intersection type. The notion of rank provides means for classification and a measure of complexity of the intersection types.

One can notice, that the construction in [8] uses types of rank four. The decidability of the inhabitation for rank three is still an open problem. The problem for rank two was so far believed to be decidable in polynomial space (see [8]).

Our result contradicts this belief. We prove the inhabitation problem for rank two to be EXPTIME-hard by a reduction from the halting problem for Alternating Linear Bounded Automata (ALBA in short). The idea of the reduction is as follows. For a given ALBA and a given word of length n we construct a type of the form $\alpha_1 \cap \ldots \cap \alpha_n \cap \alpha_{n+1} \cap \alpha_{n+2}$. For $i = 1 \ldots n$, the component α_i represents the behaviour of the i-th cell of the tape, α_{n+1} represents changes in the position of the head of the machine, and the last part α_{n+2} simulates changes of

[⋆] Partly suported by the Polish government grant 3 T11C 002 27.

the machine state. The $\cap$ operator is used here to hold and process information about the whole configuration of the automaton.

The fact that the problem for rank two is EXPTIME-hard only demonstrates how difficult the still open problem for rank three may be.

1 Basics

We consider a lambda calculus with types defined by the following induction:

– Type variables are types;
– If α and β are types, then $\alpha \to \beta$ and $\alpha \cap \beta$ are also types.

We assume that the operator $\cap$ is associative, commutative and idempotent. The type inference rules for our system are as follows:

$$(\mathrm{VAR}) \qquad \Gamma \vdash x\colon \sigma \qquad \text{if } (x\colon \sigma) \in \Gamma$$

$$(E \to) \quad \frac{\Gamma \vdash M\colon \alpha \to \beta \quad \Gamma \vdash N\colon \alpha}{\Gamma \vdash (MN)\colon \beta} \qquad (I \to) \quad \frac{\Gamma, (x\colon \alpha) \vdash M\colon \beta}{\Gamma \vdash \lambda x.M\colon \alpha \to \beta}$$

$$\frac{\Gamma \vdash M\colon \alpha \cap \beta}{\Gamma \vdash M\colon \alpha} \; (E\cap) \; \frac{\Gamma \vdash M\colon \alpha \cap \beta}{\Gamma \vdash M\colon \beta} \qquad (I\cap) \; \frac{\Gamma \vdash M\colon \alpha \quad \Gamma \vdash M\colon \beta}{\Gamma \vdash M\colon \alpha \cap \beta}$$

Definition 1. Following Leivant ([3]) we define the *rank* of a type τ:

$rank(\tau) = 0$, if τ is a simple type (without "$\cap$");
$rank(\tau \cap \sigma) = max(1, rank(\tau), rank(\sigma))$;
$rank(\tau \to \sigma) = max(1 + rank(\tau), rank(\sigma))$, when $rank(\tau) > 0$ or $rank(\sigma) > 0$.

2 Decidability of the Inhabitation Problem

2.1 The Algorithm

Definition 2. An *environment* Γ is a set of *declarations* of the form $(x\colon \alpha)$, where x is a variable and α is a type.

Definition 3. A variable x is *k-ary* in an environment Γ, if Γ includes a declaration $(x\colon \alpha)$, such that

$$\alpha = \beta_1 \to \cdots \to \beta_k \to \beta_{k+1}$$
$$\text{or}$$
$$\alpha = \gamma \cap (\beta_1 \to \cdots \to \beta_k \to \beta_{k+1}).$$

In other words, the variable is k-ary, if one can apply it to some k arguments.

Definition 4. A *constraint* is a pair (Γ, τ), denoted by $\Gamma \vdash X\colon \tau$, where Γ is an environment and τ is a type.

Definition 5. A *task* is a set of constraints

$$Z = [\Gamma_1 \vdash X\colon \tau_1,\ \ldots,\ \Gamma_n \vdash X\colon \tau_n],$$

where all the environments $\Gamma_1 \ldots \Gamma_n$ share the same domain of variables, and where the types $\tau_1 \ldots \tau_n$ are not intersections (meaning that $\tau_i \neq \alpha_i \cap \beta_i$).
A *solution* of the task Z is a term M such that for each $i = 1 \ldots n$ we have $\Gamma_i \vdash M\colon \tau_1$.

To ilustrate our algorithm, we first consider an example. Let us find an inhabitant in a β-normal form for a type $T = \tau_1 \cap \tau_2$, where

$$\tau_1 = ((((\alpha \to B) \to A) \cap ((\beta \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D)) \\ \to (\alpha \to \beta \to \alpha \to \beta \to E) \to A),$$

$$\tau_2 = ((((\beta \to B) \to A) \cap ((\alpha \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D)) \\ \to (\beta \to \alpha \to \alpha \to \beta \to E) \to A).$$

We have to find a term M which can be assigned both the type τ_1 and τ_2. The following must hold: $\emptyset \vdash M\colon \tau_1$; $\emptyset \vdash M\colon \tau_2$. From the structure of τ_1 and τ_2 we can see that M has to be an abstraction $M = \lambda XY.N$. Now we will try to find N such that: $\Gamma_1 \vdash N\colon A$; $\Gamma_2 \vdash N\colon A$, where

$$\Gamma_1 = \{X\colon (((\alpha \to B) \to A) \cap ((\beta \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D), \\ Y\colon \alpha \to \beta \to \alpha \to \beta \to E\},$$

$$\Gamma_2 = \{X\colon (((\beta \to B) \to A) \cap ((\alpha \to C) \to B) \cap ((\alpha \to D) \to C) \cap ((\beta \to E) \to D), \\ Y\colon \beta \to \alpha \to \alpha \to \beta \to E\}.$$

We notice now, that N has to be X applied to some argument (of different types in different environments). We have $N = X(\lambda x.P)$. And we search now for P such that: $\Gamma_1, x\colon \alpha \vdash P\colon B$; $\Gamma_2, x\colon \beta \vdash P\colon B$. Then again $P = X(\lambda y.Q)$, where $\Gamma_1, x\colon \alpha, y\colon \beta \vdash Q\colon C$; $\Gamma_2, x\colon \beta, y\colon \alpha \vdash Q\colon C$. We continue with $Q = X(\lambda v.R)$, and $\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha \vdash R\colon D$; $\Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha \vdash R\colon D$. Finally $R = X(\lambda z.S)$ and $\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha, z\colon \beta \vdash S\colon E$; $\Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha, z\colon \beta \vdash S\colon E$. Now we see that $S = YS_1S_2S_3S_4$, and we have to solve four tasks:

$$\begin{array}{ll}
\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha, z\colon \beta \vdash S_1\colon \alpha; & \Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha, z\colon \beta \vdash S_1\colon \beta, \\
\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha, z\colon \beta \vdash S_2\colon \beta; & \Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha, z\colon \beta \vdash S_2\colon \alpha, \\
\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha, z\colon \beta \vdash S_3\colon \alpha; & \Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha, z\colon \beta \vdash S_3\colon \alpha, \\
\Gamma_1, x\colon \alpha, y\colon \beta, v\colon \alpha, z\colon \beta \vdash S_4\colon \beta; & \Gamma_2, x\colon \beta, y\colon \alpha, v\colon \alpha, z\colon \beta \vdash S_4\colon \beta.
\end{array}$$

We see that the only possible assignment is $S_1 = x, S_2 = y, S_3 = v, S_4 = z$. Hence we found our inhabitant $M = \lambda XY.X(\lambda x.X(\lambda y.X(\lambda v.X(\lambda z.Yxyvz))))$.

Notice that, when searching for the inhabitants, we need always to consider simulteanously both environments. For each S_i we had always exactly two variables of the right type in each environment (because in every environment there are two variables of the type α and two of the type β). But only one variable had the right type in both environments. While constructing an inhabitant for a type $\tau_1 \cap \tau_2$ we had to build a common solution for both parts.

We start the decidability proof by presenting the nondeterministic algorithm. This algorithm can be easily converted into a deterministic one, but at a considerable loss of clarity. Also we remind reader that the exact procedure described below does not have the termination property. Hovewer we prove later on (see Section 2.3), that for each input type of rank at most two, our algorithm (after a few simple modifications) must find a solution in a bounded number of steps or repeat a configuration.

Definition 6. *The Algorithm.*

Our algorithm uses the "intersection removal" operation *Rem* defined as follows:

$Rem(\Gamma \vdash X\colon \tau) = \{\Gamma \vdash X\colon \tau\}$ if τ is either a type variable or $\tau = \tau_1 \to \tau_2$;
$Rem(\Gamma \vdash X\colon \tau_1 \cap \tau_2) = Rem(\Gamma \vdash X\colon \tau_1) \cup Rem(\Gamma \vdash X\colon \tau_2)$.

The purpose of the *Rem* operation is to eliminate "$\cap$" and to convert a judgement $\Gamma \vdash X\colon \tau$ with τ being possibly an intersection type into a set of judgements where the types on the right side are not intersections.

For a given type τ the first task is:

$$Z_0 = Rem(\emptyset \vdash X\colon \tau)$$

Recall that the types on the right-hand sides of tasks are not intersections. Let the current task be:

$$Z = [\Gamma_1 \vdash X\colon \tau_1, \ldots, \Gamma_n \vdash X\colon \tau_n]$$

1. If each type τ_i is of the form $\alpha_i \to \beta_i$, then the next task processed recursively by the algorithm will be:

$$Z' = Rem(\Gamma_1 \cup \{x\colon \alpha_1\} \vdash X'\colon \beta_1) \cup \ldots \cup Rem(\Gamma_n \cup \{x\colon \alpha_n\} \vdash X'\colon \beta_n),$$

   where x is a fresh variable not used in any of the Γ_i.
   If the recursive call for Z' returns M', then $M = \lambda x.M'$, if on the other hand the recursive call gives an answer "empty type", we shall give the same answer.
2. If at least one of the τ_i is a type variable, then the solution cannot be an abstraction, but must be an application or a variable. Note that all environments $\Gamma_1, \ldots, \Gamma_n$ have the same domain of variables. Suppose that there exists a number k and a variable x which is k-ary in each of the environments, and for all $j = 1 \ldots n$ we have that:

$$\Gamma_j \vdash x\colon \beta_{j1} \to \cdots \to \beta_{jk} \to \tau_j$$

   (if there is more than one such a pair, we pick nondeterministically one of them). Then:

- If $k = 0$, then $M = x$,
- If $k > 0$, then $M = xM_1 \dots M_k$, where M_i are solutions of the k independent tasks:

$$Z_1 = Rem(\Gamma_1 \vdash X_1 \colon \beta_{11}) \cup \dots \cup Rem(\Gamma_n \vdash X_1 \colon \beta_{n1}),$$
$$\dots$$
$$Z_k = Rem(\Gamma_1 \vdash X_k \colon \beta_{1k}) \cup \dots \cup Rem(\Gamma_n \vdash X_k \colon \beta_{nk}).$$

  If any of the k recursive calls gives the answer "empty type", we shall give the same answer.

If there is no such a number and a variable we give the answer "empty type".

2.2 Soundness and Completeness

Lemma 7. *If the above algorithm finds a term M for an input type τ, then $\vdash M \colon \tau$.*

Proof. By induction on M.

The algorithm proposed in Definition 6 is not capable of finding all the terms of a given type. Hence, to prove the correctness of the proposed procedure, we first define the notion of a *long* solution, then we show that every task, which has a solution, has also a long solution. Finally we complete the proof of the soundness of the algorithm by proving that every long solution can be found by the given procedure.

Definition 8. M is a *long* solution of the task $Z = [\Gamma_1 \vdash X \colon \tau_1, \dots, \Gamma_n \vdash X \colon \tau_n]$, when one of the following holds:

- All types τ_i are of the form $\alpha_i \to \beta_i$ and $M = \lambda x.M'$, where M' is a long solution of the task
  $Z' = Rem(\Gamma_1 \cup \{x \colon \alpha_1\} \vdash X' \colon \beta_1) \cup \dots \cup Rem(\Gamma_n \cup \{x \colon \alpha_n\} \vdash X' \colon \beta_n)$, or
- Some τ_i is a type variable and $M = xM_1 \dots M_k$ (possibly with $k = 0$), where for $i = 1 \dots n$ we have $\Gamma_i \vdash x \colon \alpha_{i1} \to \dots \to \alpha_{ik} \to \tau_i$ and $M_1, \dots, M_k$ are long solutions of tasks $Z_1, \dots, Z_k$, where $Z_j = [\Gamma_1 \vdash X_j \colon \alpha_{1j}, \dots, \Gamma_n \vdash X_j \colon \alpha_{nj}]$ for $j = 1 \dots k$.

Lemma 9. *If there exists a solution of a task Z, then there exists a long one.*

Proof. Assume that M is a solution of $Z = [\Gamma_1 \vdash X \colon \tau_1, \dots, \Gamma_n \vdash X \colon \tau_n]$ and that M is in a normal form. We construct a long solution $A(M, Z)$ in the following way:

- If there is a τ_i which is a type variable, then M is not an abstraction and:
  - If $M = x$, then $A(M, Z) = M$, because in this case M is a long solution,
  - If $M = xM_1 \dots M_k$, then it must hold that $\Gamma_i \vdash x \colon \alpha_{i1} \to \dots \to \alpha_{ik} \to \tau_i$ for $i = 1 \dots n$, so $A(M, Z) = xA(M_1, Z_1) \dots A(M_k, Z_k)$, where $Z_j = [\Gamma_1 \vdash X_j \colon \alpha_{1j}, \dots, \Gamma_n \vdash X_j \colon \alpha_{nj}]$ for $j = 1 \dots k$.

– Otherwise (if all τ_i have the form of $\alpha_i \to \beta_i$):
  - If $M = xM_1 \ldots M_k$ (possibly for $k = 0$) then again it must hold that $\Gamma_i \vdash x{:}\, \alpha_{i1} \to \cdots \to \alpha_{ik} \to \tau_i$ for $i = 1 \ldots n$, and also for $i = 1 \ldots n$, $j = 1 \ldots k$ it must hold that $\Gamma_i \vdash M_j{:}\, \alpha_{ij}$. Let r be the largest number, such that all the types τ_i are of the form $\beta_{1i} \to \cdots \to \beta_{ri} \to \gamma_i$. Then $A(M, Z) = \lambda z_1 \ldots z_r.xA(M_1, Z_1) \ldots A(M_k, Z_k)A(z_1, Z'_1) \ldots A(z_r, Z'_r)$, where

$$\begin{aligned}Z_j &= [\Gamma_1, (z_1{:}\, \beta_{1j}), \ldots, (z_r{:}\, \beta_{rj}) \vdash X_j{:}\, \alpha_{1j}, \ldots,\\ &\qquad\qquad \Gamma_n, (z_1{:}\, \beta_{nj}), \ldots, (z_r{:}\, \beta_{nj}) \vdash X_j{:}\, \alpha_{nj}],\\ Z'_j &= [\Gamma_1, (z_1{:}\, \beta_{1j}), \ldots, (z_r{:}\, \beta_{rj}) \vdash X'_j{:}\, \beta_{1j}, \ldots,\\ &\qquad\qquad \Gamma_n, (z_1{:}\, \beta_{nj}), \ldots, (z_r{:}\, \beta_{nj}) \vdash X'_j{:}\, \beta_{nj}].\end{aligned}$$

  - If $M = \lambda x.M'$, then $A(M, Z) = \lambda x.A(M', Z')$, where $Z' = \mathit{Rem}([\Gamma_1, (x{:}\, \alpha_1) \vdash X'{:}\, \beta_1, \ldots, \Gamma_n, (x{:}\, \alpha_n) \vdash X'{:}\, \beta_n])$.

Lemma 10. *Every long solution of the task $Z = [\Gamma_1 \vdash X{:}\, \tau_1, \ldots, \Gamma_n \vdash X{:}\, \tau_n]$ can be found by the nondeterministic procedure described in Definition 6 in one of its possible runs.*

Proof. By induction on the structure of the long solution M.

– $M = x$. Since M is long, at least one of the τ_i must be a type variable. Hence the algorithm working on the task Z will search in the environments Γ_i for a variable of the right type (case 2 of the algorithm). One of these variables is x.

– $M = xM_1 \ldots M_k$. Like before we can reason that the algorithm shall choose the case 2, and in one of its possible runs the algorithm will choose the variable x. After x is chosen, the procedure shall search for solutions of the tasks $Z_1, \ldots, Z_k$. By the definition of a long solution, we have that $M_1, \ldots, M_k$ are long solutions of the tasks $Z_1, \ldots, Z_k$, and by the induction hypothesis, these solutions can be found by the recursive runs of our procedure. It follows that also M can be found.

– $M = \lambda x.M'$. Then of course all the types τ_i have to be of the form $\alpha_i \to \beta_i$. Hence for the task Z the procedure will choose case 1, and will search for a solution of the task $Z' = \mathit{Rem}([\Gamma_1, (x{:}\, \alpha_1) \vdash X'{:}\, \beta_1, \ldots, \Gamma_n, (x{:}\, \alpha_n) \vdash X' : \beta_n\,])$. By the induction hypothesis, the long solution M' for Z' can be found by the algorithm. Hence also M can be found.

Corollary 11. *Our algorithm finds an inhabitant for every non-empty type, for which it terminates.*

Proof. A direct conclusion of Lemmas 9 and 10.

2.3 The Termination of the Algorithm

Let us consider the work of the algorithm for a type τ of rank two.

Fact 12. *Types of variables put in the environments during the work of the algorithm are of rank at most one.*

Proof. The environments are modified only in case 1. Since the type τ is of a rank at most two, the variables put in the environments have rank at most one.

Fact 13. *In every recursive run each task has at most $|\tau|$ constraints to solve.*

Proof. The number of the constraints increases only when the *Rem* operation is used. Let us denote the number of "$\cap$" operators in the type τ by $C(\tau)$. It is easy to notice that if $Rem(\Gamma \vdash X{:}\,\tau) = \{\Gamma \vdash X{:}\,\tau_1, \ldots, \Gamma \vdash X{:}\,\tau_k\}$, then the following holds:

$$C(\tau) = C(\tau_1) + \ldots + C(\tau_k) + k.$$

In other words creating new constraint always removes one "$\cap$". The recursive calls in case 2 do not increase the number of constraints. In these problems the procedure searches for terms which can serve as arguments for a variable taken from the environment. As stated in Fact 12, in environments there are only variables with types of rank zero and one, and such variables can only be given arguments with types of rank zero. And these types are simple (without intersections), so the *Rem* operation applied to them will not increase the number of constraints in a task.

The Decidability

Theorem 14 *The inhabitation problem for the types of rank two is decidable.*

Proof. The algorithm proposed in Definition 6 can be easily modified in a way which will prevent the environments from growing bigger infinitely during the work of the algorithm. Variables are added to the environments only when all currently examined types τ_i are of the form $\alpha_i \to \beta_i$. Then every environment Γ_i is expanded by a new variable of the type α_i. Note that there are only $O(|\tau|)$ types that can be assigned to a variable in one environment. Since we do not need to keep several variables of the same type (meaning of the same type in each of the environments) it follows that there is a bounded number of possible distinct environments that may occur during the work of the algorithm. Also the number of the types that may occur on the right hand side of each $\vdash$ is $O(|\tau|)$. Hence each branch of the procedure must finish or repeat a configuration after a bounded (although possibly exponential) number of steps.

3 The Lower Bound

3.1 Terms of Exponential Size

First we shall consider an instructive example. We propose a schema for creating instances of the inhabitation problem for which the above algorithm has to perform an exponential number of steps before finding the only inhabitant. The size of the inhabitant will also be exponential in the size of the type. Our example demonstrates a technique used in the construction to follow. Let $T(n) = \tau_1 \cap \ldots \cap \tau_n$, where

$$\tau_i = \alpha \to \underbrace{\Psi \to \cdots \to \Psi}_{i-1} \to (\alpha \to \beta) \to \underbrace{(\beta \to \alpha) \cdots \to (\beta \to \alpha)}_{n-i} \to \beta, \text{ and}$$
$$\Psi = (\alpha \to \alpha) \cap (\beta \to \beta).$$

For instance $T(3) =$

$$\begin{array}{l}(\alpha \to (\alpha \to \beta) \to (\beta \to \alpha) \to (\beta \to \alpha) \to \beta) \cap \\ (\alpha \to \quad \Psi \quad \to (\alpha \to \beta) \to (\beta \to \alpha) \to \beta) \cap \\ (\alpha \to \quad \Psi \quad \to \quad \Psi \quad \to (\alpha \to \beta) \to \beta)\end{array}$$

One can notice that a construction of an inhabitant for this type is similar to the rewriting from the word $\beta\beta\beta$ to the word $\alpha\alpha\alpha$ in the following way:

$$\beta\beta\beta \to \alpha\beta\beta \to \beta\alpha\beta \to \alpha\alpha\beta \to \beta\beta\alpha \to \alpha\beta\alpha \to \beta\alpha\alpha \to \alpha\alpha\alpha.$$

For $|T(n)| = O(n^2)$, there is only one (modulo α-equivalence) term t of type $T(n)$, and $|t| = O(2^n)$. For instance, the only term of type $T(3)$ is:

$$\lambda x_1 x_2 x_3 x_4 . x_2(x_3(x_2(x_4(x_2(x_3(x_2 x_1)))))),$$

while for $T(4)$ it is:

$$\lambda x_1 x_2 x_3 x_4 x_5 . x_2(x_3(x_2(x_4(x_2(x_3(x_2(x_5(x_2(x_3(x_2(x_4(x_2(x_3(x_2 x_1)))))))))))))).$$

In what follows, while proving EXPTIME-hardness of the inhabitation problem, we shall generate types of a similar form to $T(n)$. For this reason it is convenient to use $T(n)$ for introducing notions and notations, which we shall use later on. Because of the different role played by the "$\cap$" and "$\to$" it is convenient to consider the type as an object composed of columns and rows. The rows are connected with "$\cap$", and columns with "$\to$". According to this terminology $T(3)$ has three rows and five columns. We can think of α and β as of states of a certain object (e.g. a tape cell). Each row represents operations available for a given object and the initial and final state of the object. Each column represents a certain operation (that is a step of a certain automaton) which can modify the state of all objects. In type $T(3)$ there are three available operations. The i-th operation changes the i-th sign from β to α, and all earlier signs from α to β. More precisely each $(\alpha \to \beta)$ in the type $T(n)$ represents the change from β to α, and an occurence of Ψ represents no change of sign (changes α and β to themselves).

3.2 EXPTIME-Hardness

We shall show the lower bound for the complexity of the inhabitation problem by a reduction from the EXPTIME-complete problem of the in-place acceptance for alternating Turing machines.

We consider the *Alternating Linear Bound Automata* (ALBA) which are just alternating Turing machines which never leave the input word.

Definition 15. *An Alternating Linear Bound Automaton* is a sixtuple:

$$M = (Q, \Gamma, \delta, q_0, q_{acc}, g), \text{ where}$$

- Q is a non-empty, finite set of states;
- Γ is a non-empty, finite set of symbols. We shall assume that $\Gamma = \{0, 1\}$;
- $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$ - is a non-empty, finite transition relation;
- $q_0 \in Q$ is the initial state;
- $q_{acc} \in Q$ is the final state;
- $g : Q \to \{\wedge, \vee\}$ is a function which assigns a kind to every state.

Definition 16. A *configuration* of an ALBA is a triple:

$$C = (q, t, n), \text{ where}$$

- $q \in Q$ is a state;
- $t \in \Gamma^*$ is a tape content;
- $n \in N$ is a position of the head.

Definition 17. We shall say that a *transition* $p = ((q_1, s_1), (q_2, s_2, k)) \in \delta$ is *available* in a configuration $C_1 = (q_1, t, n)$, when the following conditions hold:

- $t(n) = s_1$;
- $(k = L$ and $n > 1)$ or $(k = R$ and $n < |t|)$.

In this case, we shall say that p *transforms* the configuration C_1 into the configuration $C_2 = (q_2, t_2, n_2)$, such that

- $t_2(m) = \begin{cases} s_2 & \text{if } m = n, \\ t(m) & \text{otherwise.} \end{cases}$
- $n_2 = \begin{cases} n + 1 \text{ if } k = R, \\ n - 1 \text{ otherwise.} \end{cases}$

It is worth noting that, in configurations in which the head scans the first symbol of the tape, the only available transitions are these which move the head to the right, and when the head reaches the end of the word, the only active transitions will move it to the left.

Definition 18. An ALBA *accepts* a configuration $C = (q, t, n)$, if

- $q = q_{acc}$ and $|t| = n$, or
- $g(q) = \vee$ and there exists a transition available in C, which transforms C into a configuration which the automaton accepts, or
- $g(q) = \wedge$ and every transition available in C, transforms C into a configuration which the automaton accepts.

Definition 19. *The problem of in-place acceptance* is defined as follows: does a given ALBA accept a given word t (meaning it accepts the configuration $C_0 = (q_0, t, 1)$).

It is known that APSPACE = EXPTIME (see Corollary 2 to Theorem 16.5 and Corollary 3 to Theorem 20.2 in [6]).

Lemma 20. *The problem of in-place acceptance for ALBA is* EXPTIME-complete (APSPACE-complete).

Proof. A simple modification of the proof of Theorem 19.9 in [6]. First we note that the in-place acceptance is in APSPACE. Consider a machine $M = (Q, \Gamma, \delta, q_0, q_{acc}, g)$. Keeping the counter of steps, we simulate the run of M on the input word t. We reject if M rejects, or if the machine makes $|t||Q||\Gamma|^{|t|} + 1$ steps, because after so many steps the machine has to repeat a configuration.

Let L be a language in APSPACE accepted in space n^k by a machine M. It means that M does not use in any of its parallel computations more than n^k cells of the tape (where n is the length of the input word). Let $\perp$ denote the blank symbol. Let us consider a modified machine M', which during its work performs the same moves as M, but when M reaches a final state, the head of M' makes $n^k - n$ steps to the right and also enters a final state. It is clear that M accepts t if and only if the machine M' accepts $t\perp^{n^k-n}$ without ever leaving this word (note that according to the definition, the machine M accepts in the final state only with the head at the rightmost symbol of the input word). Blank symbols $\perp$ at the very end of the word do not change the behaviour of the machine, and M does not use more than n^k cells of the tape. So t belongs to L if and only if M' accepts $t\perp^{n^k-n}$ in-place.

Definition 21. *The construction of the type.*

Let us consider the input word $t = t_1 t_2 \dots t_{n-1} t_n$. We construct a type with $n+2$ rows and some number of columns (according to the terminology introduced in Section 3.1). The first n rows shall represent the contents of n tape cells. The second last row shall represent the position of the head (values $1 \dots n$). The last row shall stand for the state of the machine.

Initial and Final State: We shall begin our construction with these two columns:

$$\begin{array}{l}
(\dots \to 0 \cap 1 \to t_1\ )\cap \\
\qquad\quad \dots \\
(\dots \to 0 \cap 1 \to t_n\ )\cap \\
(\dots \to \quad n \quad \to\ 1\ )\cap \\
(\dots \to \ q_{acc} \ \to q_0\ ).
\end{array}$$

The last column in the type represents the initial configuration: the variables $t_1 \dots t_n$ represent the symbols of the input word, the head is at first position, and the machine is in state q_0. The second last column represents the final configuration: the head of the machine is on the last cell of the tape. The content of the tape is irrelevant. Each column of the type (except the last one) will be assigned to one variable in a term. The components of a column are different types that are assigned to the same variable in $n+2$ different environments.

In the further construction we shall add new columns on the left side.

States of Kind $\vee$: Let $Id(p) = (\overbrace{0 \to \dots \to 0}^{p+1}) \cap (\overbrace{1 \to \dots \to 1}^{p+1})$. For each element $((q_1, s_1), (q_2, s_2, k))$ of δ, such that $g(q_1) = \vee$, we add $n-1$ columns — one column for each position of the head.

If $k = L$, then the i-th added column is of the form:

$$\begin{array}{l} \left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} i \\ (s_2 \rightarrow s_1) \rightarrow \\ \left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} n-i-1 \\ (i-1 \rightarrow i) \rightarrow \\ (q_2 \rightarrow q_1) \rightarrow \end{array}$$

And if $k = R$, then the i-th added column is:

$$\begin{array}{l} \left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} i-1 \\ (s_2 \rightarrow s_1) \rightarrow \\ \left.\begin{array}{l} Id(1) \rightarrow \\ \dots \\ Id(1) \rightarrow \end{array}\right\} n-i \\ (i+1 \rightarrow i) \rightarrow \\ (q_2 \rightarrow q_1) \rightarrow \end{array}$$

States of Kind $\wedge$: For each state q, such that $g(q) = \wedge$, and for each sign $s \in \Gamma$ we add n columns (one for each position of the head). The i-th column is generated this way: let $((q, s), (q_1, s_1, k_1)), \dots, ((q, s), (q_p, s_p, k_p))$ be all transitions available in q, when head is at i-th position, which holds sign s. In this case, the i-th column has the form of:

$$\begin{array}{l} \left.\begin{array}{l} Id(p) \rightarrow \\ \dots \\ Id(p) \rightarrow \end{array}\right\} i-1 \\ (s_1 \rightarrow \dots \rightarrow s_p \rightarrow s) \rightarrow \\ \left.\begin{array}{l} Id(p) \rightarrow \\ \dots \\ Id(p) \rightarrow \end{array}\right\} n-i \\ ((i + r(k_1)) \rightarrow \dots \rightarrow (i + r(k_p)) \rightarrow i) \rightarrow \\ (q_1 \rightarrow \dots \rightarrow q_p \rightarrow q) \rightarrow \end{array}$$

where

$$r(k) = \begin{cases} 1 & \text{if } k = R \\ -1 & \text{otherwise} \end{cases}$$

The above construction corresponds to the definition of the acceptance in a state of kind $\wedge$, when the automaton needs to accept in all the reachable configurations. The variable corresponding to the added column can be used in a term (inhabitant) only when it is possible to find inhabitants for each of the arguments. Each such inhabitant represents a computation in one of the possible next configurations.

Note that, if there is no reachable configuration from a state of the kind $\wedge$, then the added column will not be of a functional type (it will not have any arrows except for the one on the right), and so it will not require any further searching for inhabitants. The computation will terminate successfully, which corresponds to the acceptation of a word in states of kind $\wedge$, from which the machine has nowhere to go.

3.3 Correctness of the Reduction

We shall consider the instances of the type inhabitation problem generated by the above construction. Notice that, for such types, the construction of the inhabitant according to the algorithm proposed in Definition 6 will go as follows: first the task shall be split into $n+2$ constraints by use of the *Rem* operator, then the algorithm will use case 1 several times, after which the current task will be

$$Z_0 = [\Gamma_1 \vdash X{:}\, t_1, \ldots, \Gamma_n \vdash X{:}\, t_n, \Gamma_{n+1} \ \vdash X{:}\, 1, \Gamma_{n+2} \vdash X{:}\, q_0]$$

From this moment the algorithm will use only the application case (case 2), since the types under consideration will always be type variables. In the following steps the only thing that will change will be $s_1, \ldots, s_n, k, q$, but the environments $\Gamma_1, \ldots, \Gamma_n$ will stay the same.

Lemma 22. *Let $s_1, \ldots, s_n \in \Gamma$, $1 \leq k \leq n$ and $q \in Q$.*
The task $Z = [\Gamma_1 \vdash X{:}\, s_1, \ldots, \Gamma_n \vdash X{:}\, s_n, \Gamma_{n+1} \vdash X{:}\, k, \Gamma_{n+2} \vdash X{:}\, q]$ has a solution if and only if M accepts the configuration $C = (q, s_1 \ldots s_n, k)$.

Proof
($\Rightarrow$) Induction with respect to the structure of the solution T of the task Z.

- T is a variable x. Then $\Gamma_{n+2} \vdash x{:}\, q$, and either q is the final state, and $k = n$ (see 3.2) or q is of the kind "$\wedge$", and in the configuration C there are no available transitions (because only in this case there was a variable of a type q added to the environment Γ_{n+2} (see 3.2)). In both cases M accepts the configuration C.
- T is an abstraction. Impossible, because the types, for which we seek an inhabitant in Z are type variables.
- T is an application. There are two possibilities.
  - $T = xT_1$ and $g(q) = \vee$. According to the type construction (see 3.2) it holds that:

$$\begin{gathered}\Gamma_1 \vdash x{:}\, s_1 \to s_1,\\ \ldots\\ \Gamma_k \vdash x{:}\, s'_k \to s_k,\\ \ldots\\ \Gamma_n \vdash x{:}\, s_n \to s_n,\\ \Gamma_{n+1} \vdash x{:}\, k + r(c) \to k,\\ \Gamma_{n+2} \vdash x{:}\, q' \to q,\end{gathered}$$

and $((q, s_k), (q', s'_k, c)) \in \delta$. Hence T_1 is a solution of the task

$$[\Gamma_1 \vdash X_1 : s_1, \ldots, \Gamma_k \vdash X_1 : s'_k, \ldots, \Gamma_n \vdash X_1 : s_n, \\ \Gamma_{n+1} \vdash X_1 : k + r(c), \Gamma_{n+2} \vdash X_1 : q'].$$

By the induction hypothesis (for T_1) the machine M accepts $C_1 = (q', s_1 \ldots s'_k \ldots s_n, k + r(c))$. However, since q is of the kind $\vee$ and there exists a transition from C to C_1, it follows that M accepts also C.

- $T = xT_1 \ldots T_m$, for some m and $g(q) = \wedge$. According to the type construction (see 3.2) it must hold that:

$$\begin{gathered} \Gamma_1 \vdash x : s_1 \to \cdots \to s_1 \to s_1, \\ \ldots \\ \Gamma_k \vdash x : s_{k1} \to \cdots \to s_{km} \to s_k, \\ \ldots \\ \Gamma_n \vdash x : s_n \to \cdots \to s_n \to s_n, \\ \Gamma_{n+1} \vdash x : k + r(c_1) \to \cdots \to k + r(c_m) \to k, \\ \Gamma_{n+2} \vdash x : q_1 \to \cdots \to q_m \to q, \end{gathered}$$

and the following transitions are all the transitions available in C: $((q, s_k), (q_1, s_{k1}, c_1)), \ldots, ((q, s_k), (q_m, s_{km}, c_m))$. Then of course each T_i is a solution of the task:

$$[\Gamma_1 \vdash X_i : s_1, \ldots, \Gamma_k \vdash X_i : s_{ki}, \ldots, \Gamma_n \vdash X_i : s_n, \\ \Gamma_{n+1} \vdash X_i : k + r(c_i), \Gamma_{n+2} \vdash X_i : q_i].$$

By the induction hypothesis for $T_1, \ldots, T_m$, the machine M accepts all the configurations $C_1, \ldots, C_m$, where $C_i = (q_i, s_1 \ldots s_{ki} \ldots s_n, k + r(c_i))$. It means that M accepts all configurations reachable from C in one step, so it accepts C.

($\Leftarrow$) Induction with respect to the definition of acceptance.

(*Base*) Let $q = q_{acc}$. Then $k = n$, because the machine accepts only with the head at the rightmost position. Then according to the construction for the final state (see 3.2), there exists a variable x, such that:
$\Gamma_1 \vdash x : s_1, \ldots, \Gamma_n \vdash x : s_n, \Gamma_{n+1} \vdash x : n, \Gamma_{n+2} \vdash x : q_{acc}$. So $T = x$ is a solution of Z.

(*Step*) Assume that M accepts $C = (q, s_1 \ldots s_n, k)$, where q is not the final state. There are two possibilities:

– Let $g(q) = \vee$. Since M accepts the configuration C it means that there exists a transition $((q, s_k), (q', s'_k, c))$, such that M accepts configuration $C_1 = (q', s_1 \ldots s'_k \ldots s_n, k + r(c))$. According to the induction hypothesis there exists a solution T_1 of the task

$$[\Gamma_1 \vdash X_1 : s_1, \ldots, \Gamma_k \vdash X_1 : s'_k, \ldots, \Gamma_n \vdash X_1 : s_n, \\ \Gamma_{n+1} \vdash X_1 : k + r(c), \Gamma_{n+2} \vdash X_1 : q'].$$

Since q is of the kind $\vee$, then according to the construction (see 3.2) there exists a variable x, such that

$$\begin{gathered}\Gamma_1 \vdash x\colon s_1 \to s_1,\\ \dots\\ \Gamma_k \vdash x\colon s'_k \to s_k,\\ \dots\\ \Gamma_n \vdash x\colon s_n \to s_n,\\ \Gamma_{n+1} \vdash x\colon k + r(c) \to k,\\ \Gamma_{n+2} \vdash x\colon q' \to q.\end{gathered}$$

So $T = xT_1$ is a solution of the task Z.

– $g(q) = \wedge$. Then for each transition $((q, s_k), (q_i, s_{ki}, c_i))$ available from C, machine M accepts configuration $C_i = (q_i, s_1 \dots s_{ki} \dots s_n, k + r(c_i))$. By the induction hypothesis there exist solutions $T_1, \dots, T_m$ of tasks $Z_1, \dots, Z_m$, where

$$Z_i = [\Gamma_1 \vdash X_i\colon s_1,\ \dots,\ \Gamma_k \vdash X_i\colon s_{ki},\ \dots,\ \Gamma_n \vdash X_i\colon s_n,\\ \Gamma_{n+1} \vdash X_i\colon k + r(c_i),\ \Gamma_{n+2} \ \vdash\ X_i\colon\ q_i].$$

Since q is of the kind $\wedge$, there must (see 3.2) exist a variable x, such that

$$\begin{gathered}\Gamma_1 \vdash x\colon s_1 \to \dots \to s_1 \to s_1,\\ \dots\\ \Gamma_k \vdash x\colon s_{k1} \to \dots \to s_{km} \to s_k,\\ \dots\\ \Gamma_n \vdash x\colon s_n \to \dots \to s_n \to s_n,\\ \Gamma_{n+1} \vdash x\colon k + r(c_1) \to \dots \to k + r(c_m) \to k,\\ \Gamma_{n+2} \vdash x\colon q_1 \to \dots \to q_m \to q.\end{gathered}$$

Then $T = xT_1 \dots T_m$ is a solution of Z.

Corollary 23. *The inhabitation problem for rank two intersection types is* EXPTIME-hard.

References

[1] Alessi, F., Barbanera, F., Dezani-Ciancanglini, M.: Intersection types and lambda models. Theoretical Computer Science 355(2), 108–126 (2006)

[2] Kurata, T., Takahashi, M.: Decidable properties of intersection type systems. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 297–311. Springer, Heidelberg (1995)

[3] Leivant, D.: Polymorphic type inference. Proceedings of the 10th ACM Symposium on Principles of Programing Languages, Austin, Texas, pp. 88–98 (1983)

[4] Lopez-Escobar, E.G.K.: Proof-Functional Connectives. LNCS, vol. 1130, pp. 208–221. Springer-Verlag, Heidelberg (1985)
[5] Mints, G.: The Completeness of Provable Realizability. Notre Dame. Journal of Formal Logic 30, 420–441 (1989)
[6] Papadimitriou, C.H.: Computational Complexity. Addison-Wesley Publishing Company, Reading, MA (1995)
[7] Statman, R.: Intuitionistic propositional logic is polynomial-space complete. TCS 9, 67–72 (1979)
[8] Urzyczyn, P.: The emptiness problem for intersection types. Journal of Symbolic Logic 64(3), 1195–1215 (1999)