

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.

Key words: 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)”. The answer to this question depends on the system and its type inference rules. By the Curry-Howard correspondence, the inhabitation problem for a given type system is equivalent to the decidability of the corresponding logic.

In the simply typed system the inhabitation problem is PSPACE-complete (see [7]).

The intersection types system studied in the current paper allows 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 . This system corresponds to the logic of “strong conjunction” (see [1, 4, 5]).

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 $(I\cap)$.

Leivant in [3] defines the rank of an intersection type. The notion of rank turns out to be very useful, since it provides means for classification and a measure of complexity of the intersection types.

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

[★] Partly supported by KBN grant 3 T11C 002 27

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 $\alpha_1 \cap \dots \cap \alpha_n \cap \alpha_{n+1} \cap \alpha_{n+2}$. The intended meaning of the components $\alpha_1, \dots, \alpha_n$ is that α_i represents the behaviour of the i -th cell of the tape, the α_{n+1} represents changes in the position of the head of machine, and the last part α_{n+2} simulates changes of the machine state. The \cap operator is used here to hold and process information about the whole configuration of the automata.

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

1 Basics

1.1 Intersection Types

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

- Type variables are types
- If α and β are types, then $\alpha \rightarrow \beta$ and $\alpha \cap \beta$ are also types

We assume that the operator \cap is associative, commutative and idempotent. That is, the meaning of the intersection type of the form $\alpha_1 \cap \dots \cap \alpha_m$ does not depend on the order or number of occurrences of each of the types α_i .

The type inference rules for our system are as follows:

$$(VAR) \quad \Gamma \vdash x : \sigma \qquad \text{if } (x : \sigma) \in \Gamma$$

$$(E \rightarrow) \quad \frac{\Gamma \vdash M : \alpha \rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash (MN) : \beta}$$

$$(I \rightarrow) \quad \frac{\Gamma, (x : \alpha) \vdash M : \beta}{\Gamma \vdash \lambda x. M : \alpha \rightarrow \beta}$$

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

$$(I\cap) \quad \frac{\Gamma \vdash M : \alpha \quad \Gamma \vdash M : \beta}{\Gamma \vdash M : \alpha \cap \beta}$$

By the Curry-Howard correspondence the calculus defined above corresponds to the logic of “strong conjunction” of Mints and Lopez-Escobar (see [1, 4, 5]). While considering the rule $(I\cap)$ one can easily notice the specific binding of the

types α and β by the term M . A term has a type $\alpha \cap \beta$ if it has a type α and at the same time it has a type β . Accordingly in the corresponding logic, one can prove a formula $\alpha \cap \beta$ if one can prove α and β and both formulae have at least one common proof (strong conjunction). It follows that checking the correctness of formulae in the logic of “strong conjunction” may prove to be conceptually difficult and usually it is more convenient to regard the equivalent problem of inhabitation for the intersection type systems.

1.2 Intersection Types Classification

Definition 1 Following Leivant ([3]) we define the *rank* of a type τ , denoted by $rank(\tau)$:

$$\begin{aligned} rank(\tau) &= 0, \text{ if } \tau \text{ is a simple type (without “}\cap\text{”)}; \\ rank(\tau \cap \sigma) &= \max(1, rank(\tau), rank(\sigma)); \\ rank(\tau \rightarrow \sigma) &= \max(1 + rank(\tau), rank(\sigma)), \text{ when } rank(\tau) > 0 \text{ or } rank(\sigma) > 0. \end{aligned}$$

2 Decidability of the Inhabitation Problem

2.1 The Algorithm

Definition 2 A variable x is *k-ary* in an environment Γ , if Γ includes a declaration $(x:\alpha)$, such that

$$\begin{aligned} \alpha &= \beta_1 \rightarrow \dots \rightarrow \beta_k \rightarrow \tau \\ &\text{or} \\ \alpha &= \gamma \cap (\beta_1 \rightarrow \dots \rightarrow \beta_k \rightarrow \tau) \end{aligned}$$

The variable is *k-ary*, if one can apply it to some k arguments of certain types.

The Algorithm. The task considered by the algorithm in each step is a set of problems:

$$\begin{aligned} &[\Gamma_1 \vdash M:\tau_1, \dots, \Gamma_n \vdash M:\tau_n] \\ &(n \text{ environments, } n \text{ types, but only one term to be found}) \end{aligned}$$

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

$$\begin{aligned} Rem(\Gamma \vdash M:\tau) &= \{\Gamma \vdash M:\tau\} \text{ if } \tau \text{ is not an intersection} \\ Rem(\Gamma \vdash M:\tau_1 \cap \tau_2) &= Rem(\Gamma \vdash M:\tau_1) \cup Rem(\Gamma \vdash M:\tau_2) \end{aligned}$$

The purpose for the *Rem* operation is to eliminate “ \cap ” and to convert a judgement $\Gamma \vdash M:\tau$ 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:

$$Z_0 = Rem(\emptyset \vdash M:\tau)$$

It is worth noting that in the tasks processed by our procedure types on the right side will never be intersections.

Let the current task be:

$$Z = [\Gamma_1 \vdash M: \tau_1, \dots, \Gamma_n \vdash M: \tau_n]$$

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

$$Z' = \text{Rem}(\Gamma_1 \cup \{x: \alpha_1\} \vdash M': \beta_1) \cup \dots \cup \text{Rem}(\Gamma_n \cup \{x: \alpha_n\} \vdash M': \beta_n),$$

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 term M cannot be an abstraction, but must be an application or a variable. Suppose that there exists a variable x which is k -ary in each of the environments, and for each i it holds that:

$$\Gamma_i \vdash x: \beta_{i1} \rightarrow \dots \rightarrow \beta_{ik} \rightarrow \tau_i$$

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

- If $k = 0$, then $M = x$,
- If $k > 0$, then $M = x M_1 \dots M_k$, where M_i are solutions for the k independent problems:

$$Z_1 = [\Gamma_1 \vdash M_1: \beta_{11}, \dots, \Gamma_n \vdash M_1: \beta_{n1}],$$

...

$$Z_k = [\Gamma_1 \vdash M_k: \beta_{1k}, \dots, \Gamma_n \vdash M_k: \beta_{nk}]$$

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

3. Otherwise we give the answer “empty type”.

Fact 3 *If the above algorithm finds a term M for an input type τ , then $\vdash M: \tau$.*

Proof. Straightforward.

2.2 Soundness

The algorithm described above 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 4 M is a *long* solution of the task $Z = [\Gamma_1 \vdash M : \tau_1, \dots, \Gamma_n \vdash M : \tau_n]$, when one of the following holds:

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

Lemma 5 *If there exists a solution of a task Z , then there exists a long one.*

Proof. Assume that M is a solution of $Z = [\Gamma_1 \vdash M : \tau_1, \dots, \Gamma_n \vdash M : \tau_n]$. 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 $\Gamma_i \vdash x : \alpha_{i1} \rightarrow \dots \rightarrow \alpha_{ik} \rightarrow \tau_i$ for $i = 1 \dots n$, so $A(M, Z) = xA(M_1, Z_1) \dots A(M_k, Z_k)$, where $Z_j = [\Gamma_1 \vdash M_j : \alpha_{1j}, \dots, \Gamma_n \vdash M_j : \alpha_{nj}]$ for $j = 1 \dots k$.
- Otherwise (if all τ_i have the form of $\alpha_i \rightarrow \beta_i$):
 - If $M = xM_1 \dots M_k$ (possibly for $k = 0$), then $A(M, Z) = \lambda z.A(Mz, Z')$, where $Z' = [\Gamma_1, (z : \alpha_1) \vdash Mz : \beta_1, \dots, \Gamma_n, (z : \alpha_n) \vdash Mz : \beta_n]$ (since Mz is a solution of the task Z' , and $\lambda z.Mz$ – of the task Z).
 - If $M = \lambda x.M'$, then $A(M, Z) = \lambda x.A(M', Z')$, where $Z' = \text{Rem}([\Gamma_1, (x : \alpha_1) \vdash M' : \beta_1, \dots, \Gamma_n, (x : \alpha_n) \vdash M' : \beta_n])$.

Lemma 6 *Every long solution M of the task $Z = [\Gamma_1 \vdash M : \tau_1, \dots, \Gamma_n \vdash M : \tau_n]$ can be found by the above alternating procedure.*

Proof. By induction on the structure of 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 the variables that the algorithm chooses from is of course x .
- $M = xM_1 \dots M_k$. 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, \dots, Z_k . By the definition of a long solution, we have that M_1, \dots, M_k are long solutions of the tasks Z_1, \dots, 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 $\alpha_i \rightarrow \beta_i$. Hence for the task Z the procedure shall choose case 1, and search for solution of the task $Z' = \text{Rem}([\Gamma_1, (x:\alpha_1) \vdash M':\beta_1, \dots, \Gamma_n, (x:\alpha_n) \vdash M':\beta_n])$. By the induction hypothesis, a long solution M' for Z' can be found by the algorithm.

Corollary 7 *Our algorithm finds an inhabitant for every non-empty type, for which it terminates.*

Proof. A direct conclusion of Lemmas 5 and 6.

3 The Termination of the Algorithm

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

Fact 8 *Types of variables put in the environments during the work of algorithm are of the rank at most one.*

Proof. The environments are modified only in case 1. If any of the variables put in the environments was of rank two, then the type τ must have been of the rank three.

Fact 9 *In every recursive run there is no task with more than $|\tau|$ simultaneous problems to solve.*

Proof. When a new problem is generated by the *Rem* operation, one “ \cap ” is removed from the type τ and the type is split between the problems. There can never be more than $|\tau|$ problems. The recursive calls in case 2 do not create any new parallel problems, because of the way they are created. Namely, in these problems the procedure searches for terms which can serve as arguments for a variable taken from the environment. As we noticed before, 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 they do not generate new problems by the *Rem* operation.

3.1 The Decidability

Theorem 10 *The inhabitation problem for rank two intersection types is decidable.*

Proof. First notice that the environments cannot grow 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 $\alpha_i \rightarrow \beta_i$. 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 finite 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 alternating procedure must finish or repeat a configuration in a finite (although possibly exponential) number of steps.

4 The Lower Bound

4.1 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 $T(n) = \tau_1 \cap \dots \cap \tau_n$, where

$$\tau_i = \alpha \rightarrow \underbrace{\Psi \rightarrow \dots \rightarrow \Psi}_{i-1} \rightarrow (\alpha \rightarrow \beta) \rightarrow \underbrace{(\beta \rightarrow \alpha) \dots \rightarrow (\beta \rightarrow \alpha)}_{n-i} \rightarrow \beta, \text{ and}$$

$$\Psi = (\alpha \rightarrow \alpha) \cap (\beta \rightarrow \beta).$$

For instance $T(3) =$

$$\begin{aligned} &(\alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha) \rightarrow \beta) \cap \\ &(\alpha \rightarrow \Psi \rightarrow (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \alpha) \rightarrow \beta) \cap \\ &(\alpha \rightarrow \Psi \rightarrow \Psi \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta) \end{aligned}$$

One can notice that a construction of an inhabitant for this type is similar to the rewriting process from the word $\beta\beta\beta$ to the word $\alpha\alpha\alpha$, and it is a letter by letter rewriting. For $|T(n)| = O(n^2)$, there is only one term t of type $T(n)$, and $|t| = O(2^n)$. For instance, the only (modulo α -equivalence) 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 worth 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 “ \rightarrow ” it is convenient to

consider the structure of the type in terms of columns and rows. The rows are connected with “ \cap ”, and columns with “ \rightarrow ” (in the case of $T(n)$ there are n rows: τ_1, \dots, τ_n). According to this terminology $T(3)$ has three rows and five columns. One row represents operations available for a given object and the initial and final state of the object (here states are variables α and β). One column represents a certain operation (that is a step of a certain automaton). 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 $(\alpha \rightarrow \beta)$ in the type $T(n)$ represents the change from β to α , and an occurrence of Ψ represents no change of sign (changes β and α to themselves).

4.2 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.

Definition 11 An *alternating Turing machine* is a quintuple:

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

- 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.
- $g : Q \rightarrow \{\wedge, \vee, \text{accept}\}$ is a function which assigns a kind of every state.

Definition 12 A *configuration* of an alternating Turing machine machine is a triple:

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

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

Definition 13 An *Alternating Linear Bounded Automaton* (ALBA) is an alternating Turing machine whose tape head never leaves the input word.

Definition 14 We shall say that a *transition* $p = ((q_1, s_1), (q_2, s_2, k))$ is *consistent* with the relation δ in a configuration $C_1 = (q_1, t, n)$, when the following conditions hold:

- $p \in \delta$;
- $t(n) = s_1$;
- $(k = L \text{ and } n > 1) \text{ or } (k = R \text{ and } n < |t|)$.

We shall say then that p *transforms* a configuration C_1 to a configuration $C_2 = (q_2, t_2, n_2)$, where

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

Definition 15 An alternating Turing machine *accepts* in configuration $C = (q, t, n)$, if

- $g(q) = \text{accept}$ and $|t| = n$, or
- $g(q) = \vee$ and there exists a transition consistent with δ , which transforms the configuration C to a configuration in which the automaton accepts, or
- $g(q) = \wedge$ and every transition consistent with δ , transforms C to a configuration in which the automaton accepts.

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 16 *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 $\text{APSPACE} = \text{EXPTIME}$ (see Corollary 2 to Theorem 16.5 and Corollary 3 to Theorem 20.2 in [6]).

Lemma 17 *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 $M = (Q, \Gamma, \delta, q_0, g)$. Keeping the counter of steps, we simulate the run of M on the input word t . We reject if M rejects, or if machine makes more than $|t||Q||\Gamma|^{|t|}$ steps, because after so many steps 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 us denote the blank symbol by \perp . Let us consider a modified machine M' , which during its work performs the same moves as M , but when M reaches an accepting state, the head of M' makes $n^k - n$ steps to the right and also enters an accepting 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, machine M accepts with the head at 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.

Theorem 18 *The inhabitation problem for rank two intersection types is EXPTIME-hard.*

Proof. 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 4.1). The first n rows shall represent the state of n tape cells. The next row shall represent the position of the head (values $1 \dots n$). The last row shall stand for the state of the machine. Let q_{acc} be a new type variable. We shall begin our construction with these two columns:

$$\begin{aligned} & (\dots \rightarrow 2 \rightarrow t_1) \cap \\ & \quad \dots \\ & (\dots \rightarrow 2 \rightarrow t_n) \cap \\ & (\dots \rightarrow 0 \rightarrow 1) \cap \\ & (\dots \rightarrow q_{acc} \rightarrow q_0) \end{aligned}$$

where $q_{acc} \notin Q$. The last column in the type represents the initial configuration: in the cells there are symbols from the input word t , the variables $t_1 \dots t_n$ represent the input word, the head is at first position, and the machine is in state q_0 . The second last column represents the final state.

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

Accepting States: If the machine has any accepting states, we add a column responsible for a transition from the accepting states to our additional state q_{acc} . Let the $q_1 \dots q_r$ be all the accepting states of the machine. The additional column will be:

$$\left. \begin{array}{l} S \rightarrow \\ \dots \\ S \rightarrow \end{array} \right\}^n$$

$$\begin{array}{l} K \rightarrow \\ Q \rightarrow \end{array}$$

where

$$\begin{aligned} S &= (2 \rightarrow 0) \cap (2 \rightarrow 1), \\ K &= (0 \rightarrow n), \\ Q &= (q_{acc} \rightarrow q_1) \cap \dots \cap (q_{acc} \rightarrow q_r). \end{aligned}$$

Each column of the type (except the last one) will be assigned to one variable in a term. The components of a column are just different types that are assigned to the same variable in $n + 2$ different environments. The variable, which will have assigned types being parts of this column, is responsible for transition from each accepting state of the machine (for each tape content and for head of machine being at last sign of the word) to the state q_{acc} .

States of Kind \vee : Let $Id(p) = (\overbrace{0 \rightarrow \dots \rightarrow 0}^{p+1}) \cap (\overbrace{1 \rightarrow \dots \rightarrow 1}^{p+1})$. 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:

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

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

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

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:

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

where

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

The above construction corresponds to the definition of an acceptance in a state of the kind \wedge , when the automaton needs to accept in all the reachable configurations. The variable corresponding to the added column can be used in 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 configurations (after executing the appropriate step).

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 acceptance of a word in states of kind \wedge , from which the machine has nowhere to go.

4.3 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 section 2 will go as follows: first the problem shall be split into $n + 2$ subproblems by use of the *Rem* operator, then the algorithm will use the case 1 several times, after which the current task will be

$$Z = [\Gamma_1 \vdash T : s_1, \dots, \Gamma_n \vdash T : s_n, \Gamma_{n+1} \vdash T : k, \Gamma_{n+2} \vdash T : q].$$

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

Lemma 19 *Let $Z = [\Gamma_1 \vdash T : s_1, \dots, \Gamma_n \vdash T : s_n, \Gamma_{n+1} \vdash T : k, \Gamma_{n+2} \vdash T : q]$ be a task for a type generated by the above construction for an alternating machine M . For $q \in Q$ the task Z has a solution if and only if the machine M accepts in place the configuration $C = (q, s_1 \dots s_n, k)$.*

Proof.

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

- T is a variable x . Then $\Gamma_{n+2} \vdash x : q$. Then q is of the kind “ \wedge ”, and from the configuration C there are no transitions consistent with δ (because only in this case there was a variable of a type being a type variable added to the environments (see 4.2)). Hence 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 three possibilities.

- $T = x_1 x_{acc}$, where $\Gamma_{n+2} \vdash x_1 : q_{acc} \rightarrow q$. According to the type construction (see 4.2), the only variables, which in the environment Γ_{n+2} can be supplied with the argument of the type q_{acc} are variables representing accepting states. Hence q is an accepting state of M , so M accepts in C .
- $T = xT_1$ and $g(q) = \vee$. According to the type construction (see 4.2) it holds that:

$$\begin{aligned}
&\Gamma_1 \vdash x : s_1 \rightarrow s_1, \\
&\quad \dots \\
&\Gamma_k \vdash x : s'_k \rightarrow s_k, \\
&\quad \dots \\
&\Gamma_n \vdash x : s_n \rightarrow s_n, \\
&\Gamma_{n+1} \vdash x : k + r(c) \rightarrow k, \\
&\Gamma_{n+2} \vdash x : q' \rightarrow q,
\end{aligned}$$

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

$$\begin{aligned}
&[\Gamma_1 \vdash T_1 : s_1, \dots, \Gamma_k \vdash T_1 : s'_k, \dots, \Gamma_n \vdash T_1 : s_n, \\
&\quad \Gamma_{n+1} \vdash T_1 : k + r(c), \Gamma_{n+2} \vdash T_1 : q'].
\end{aligned}$$

By the induction hypothesis (for T_1) the machine M accepts in $C_1 = (q', s_1 \dots s'_k \dots s_n, k + r(c))$. 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 .

- $T = xT_1 \dots T_m$, for some m and $g(q) = \wedge$. According to the type construction (see 4.2) it must hold that:

$$\begin{aligned}
&\Gamma_1 \vdash x : s_1 \rightarrow \dots \rightarrow s_1 \rightarrow s_1, \\
&\quad \dots \\
&\Gamma_k \vdash x : s_{k1} \rightarrow \dots \rightarrow s_{km} \rightarrow s_k, \\
&\quad \dots \\
&\Gamma_n \vdash x : s_n \rightarrow \dots \rightarrow s_n \rightarrow s_n, \\
&\Gamma_{n+1} \vdash x : k + r(c_1) \rightarrow \dots \rightarrow k + r(c_m) \rightarrow k, \\
&\Gamma_{n+2} \vdash x : q_1 \rightarrow \dots \rightarrow q_m \rightarrow q
\end{aligned}$$

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

$$\begin{aligned}
&[\Gamma_1 \vdash T_i : s_1, \dots, \Gamma_k \vdash T_i : s_{ki}, \dots, \Gamma_n \vdash T_i : s_n, \\
&\quad \Gamma_{n+1} \vdash T_i : k + r(c_i), \Gamma_{n+2} \vdash T_i : q_i].
\end{aligned}$$

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

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

(*Base*) Let $g(q) = \text{accept}$. Then $k = n$, because the machine accepts only with the head in the rightmost position. Then according to the construction for accepting types (see 4.2), there exists a variable x , such that: $\Gamma_1 \vdash x: 2 \rightarrow s_1, \dots, \Gamma_n \vdash x: 2 \rightarrow s_n, \Gamma_{n+1} \vdash x: 0 \rightarrow n, \Gamma_{n+2} \vdash x: q_{acc} \rightarrow q$. So $T = xx_{acc}$ is a solution of Z .

(*Step*) Assume that M accepts in $C = (q, s_1 \dots s_n, k)$, where q is not an accepting state. There are two possibilities:

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

$$\begin{aligned} &[\Gamma_1 \vdash T_1: s_1, \dots, \Gamma_k \vdash T_1: s'_k, \dots, \Gamma_n \vdash T_1: s_n, \\ &\quad \Gamma_{n+1} \vdash T_1: k + r(c), \Gamma_{n+2} \vdash T_1: q']. \end{aligned}$$

Since q is of the kind \vee , then according to the construction (see 4.2) there exists a variable x , such that

$$\begin{aligned} &\Gamma_1 \vdash x: s_1 \rightarrow s_1, \\ &\quad \dots \\ &\Gamma_k \vdash x: s'_k \rightarrow s_k, \\ &\quad \dots \\ &\Gamma_n \vdash x: s_n \rightarrow s_n, \\ &\Gamma_{n+1} \vdash x: k + r(c) \rightarrow k, \\ &\Gamma_{n+2} \vdash x: q' \rightarrow q. \end{aligned}$$

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 in configuration $C_i = (q_i, s_1 \dots s_{ki} \dots s_n, k + r(c_i))$. By the induction hypothesis T_1, \dots, T_m are solutions of the tasks Z_1, \dots, Z_m , where

$$\begin{aligned} &Z_i = [\Gamma_1 \vdash T_i: s_1, \dots, \Gamma_k \vdash T_i: s_{ki}, \dots, \Gamma_n \vdash T_i: s_n, \\ &\quad \Gamma_{n+1} \vdash T_i: k + r(c_i), \Gamma_{n+2} \vdash T_i: q_i]. \end{aligned}$$

Since q is of the kind \wedge , there must (see 4.2) exist a variable x , such that

$$\begin{aligned} &\Gamma_1 \vdash x: s_1 \rightarrow \dots \rightarrow s_1 \rightarrow s_1, \\ &\quad \dots \\ &\Gamma_k \vdash x: s_{k1} \rightarrow \dots \rightarrow s_{km} \rightarrow s_k, \\ &\quad \dots \end{aligned}$$

$$\begin{aligned}
\Gamma_n &\vdash x: s_n \rightarrow \cdots \rightarrow s_n \rightarrow s_n, \\
\Gamma_{n+1} &\vdash x: k + r(c_1) \rightarrow \cdots \rightarrow k + r(c_m) \rightarrow k, \\
\Gamma_{n+2} &\vdash x: q_1 \rightarrow \cdots \rightarrow q_m \rightarrow q.
\end{aligned}$$

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

References

1. Alessi, F., Barbanera, F., Dezani-Ciancanglini, M. Intersection types and lambda models, *Theoretical Computer Science* 355(2), 2006, 108–126.
2. Kurata, T., Takahashi, M. Decidable properties of intersection type systems, *LNCS Vol 902*, Typed Lambda Calculi and Applications, 1995, Dezani, M., Plotkin, G., Eds. 297–311.
3. Leivant, D. Polymorphic type inference, *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, Austin, Texas, 1983, 88–98.
4. Lopez-Escobar, E.G.K. Proof Functional Connectives, *Proceedings of Methods in Logic, 1993, LNMath 1130*, Springer-Verlag, Berlin, 1985, 208–221.
5. Mints, G. The Completeness of Provable Realizability, *Notre Dame Journal of Formal Logic Vol 30*, 1989, 420–441.
6. Papadimitriou, Ch. H. *Computational Complexity*, Adison-Wesley Publishing Company Inc., 1995
7. Statman, R. Intuitionistic propositional logic is polynomial-space complete, *TCS Vol 9*, 1979, 67–72.
8. Urzyczyn, P. The emptiness problem for intersection types, *Journal of Symbolic Logic* 64(3), 1999, 1195–1215.