Inhabitation of Low-Rank Intersection Types[⋆]

Paweł Urzyczyn

Institute of Informatics, University of Warsaw
urzy@mimuw.edu.pl

Abstract. We prove that the inhabitation problem ("Does there exist a closed term of a given type?") is undecidable for intersection types of rank 3 and exponential space complete for intersection types of rank 2.

1 Introduction

Calculi with intersection types have been known for almost 30 years [4,16] and their importance is widely recognized in the literature. Just to mention a few applications: construction of lambda-models [2,4], normalization issues and optimal reductions [16,15], type inference and compilation [6,22].

Another, more foundational, aspect is the logic of intersection types. It has been noticed long ago that this "proof-functional", rather than "truth-functional", logic significantly differs from most other logical systems [1,13,14] and various studies were undertaken to understand the nature of this peculiar logic [5,12,17,21]. In particular, it is known that the decision problem for the logic of intersections, that is, the inhabitation problem, is undecidable [20]. This means that the expressive power of the propositional (!) logic of intersections is enormous. It is thus important to locate the exact borderline between decidable and undecidable cases. We address here the question what is the maximal *rank* of types for which the logic of intersection is still decidable, and what is the complexity? (Roughly speaking, an intersection of simple types is of rank 1, and a rank $n + 1$ type has arguments of rank n.)

In [20] it is shown that the inhabitation problem is undecidable for types of rank 4. For rank 2 the problem is decidable, but EXPTIME-hard [10]. (Incidentally, rank 2 subsystem is interesting in that its typing power is exactly the same as of rank 2 parametric polymorphism [23].)

This paper aims at closing the gap between the results of [10,20]. We show that the logic of rank 3 types is undecidable (Theorem 4). We also improve the lower bound for rank 2, proving EXPSPACE-hardness. Since the obvious algorithm for rank 2 runs in exponential space, we conclude that the problem is EXPSPACE-complete (Theorem 9). The intermediate problem used in the proof of Theorem 9 is the halting problem for a *bus machine*—an alternating device operating on a fixed length word, but expanding its own program (the set of available instructions) during the computation. This relatively simple model capturing the notion of exponential space, without explicitly operating in exponential space, may perhaps be of some interest by itself?

[⋆] Partly supported by Ministry of Science and Higher Education grant N N206 355836.

The system. We consider the basic system of intersection types, with no constants nor subtyping. For simplicity, we assume that the intersection operator is idempotent, commutative and associative. The inference rules are given below:

$$(\mathrm{Ax})\ \Gamma,\, x:\tau \vdash x : \tau$$

$$(\mathrm{I}{\to})\ \frac{\Gamma,\, x:\sigma \vdash M : \tau}{\Gamma \vdash (\lambda x{:}\sigma\, M) : \sigma \to \tau} \qquad (\mathrm{E}{\to})\ \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$$

$$(\mathrm{I}{\cap})\ \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash M : \sigma \cap \tau} \qquad (\mathrm{E}{\cap})\ \frac{\Gamma \vdash M : \sigma \cap \tau}{\Gamma \vdash M : \sigma}$$

After Leivant [11], we set $\mathrm{rank}(\tau) = 0$, when τ is a simple type, and we define: $\mathrm{rank}(\sigma{\to}\tau) = \max\{\mathrm{rank}(\sigma){+}1, \mathrm{rank}(\tau)\}$; $\mathrm{rank}(\sigma{\cap}\tau) = \max\{1, \mathrm{rank}(\sigma), \mathrm{rank}(\tau)\}$.

The algorithm. The natural partial algorithm to find inhabitants of a given intersection type [3,10,20] is similar to the Wajsberg/Ben-Yelles algorithm for simple types [18]. But instead of processing constraints of the form $\Gamma \vdash\ ? : \tau$ one at a time, one asks for a single solution X of a system of constraints:

$$\Gamma_1 \vdash X : \tau_1,\ \ldots,\ \Gamma_n \vdash X : \tau_n,$$

where the domains of the environments Γ_i are the same for all $i = 1, \ldots, n$.

The nondeterministic procedure guesses step by step the shape of a normal solution X and verifies the correctness by transforming the constraints until they are trivially satisfied. There are esssentially three possible transformations.

1. If one of the types τ_i is an intersection, say $\tau_i = \sigma \cap \rho$, then the constraint $\Gamma_i \vdash X : \tau_i$ is replaced by two: $\Gamma_i \vdash X : \sigma$ and $\Gamma_i \vdash X : \rho$.
2. If every τ_i is an implication, say $\tau_i = \sigma_i \to \rho_i$, then a possible guess is that $X = \lambda x.X'$, and therefore we can set the following new constraints:
$$\Gamma_1, x : \sigma_1 \vdash X' : \rho_1,\ \ldots,\ \Gamma_n, x : \sigma_n \vdash X' : \rho_n,$$
if there is no variable y assigned type σ_i in every Γ_i; otherwise we can identify variables x and y and keep the Γ_i unchanged.
3. Another possibility is that there is a variable x and a number k such that we have $\Gamma_i \vdash \lambda z_1 \ldots z_k.x z_1 \ldots z_k : \rho_i^1 \to \cdots \to \rho_i^k \to \tau_i$, for each i, where z_i are fresh. Then we may guess that $X = xZ^1 \ldots Z^k$ and consider k systems:
$$\Gamma_1 \vdash Z^j : \rho_1^j\ \ldots,\ \Gamma_n, \vdash Z^j : \rho_n^j,$$
for $j = 1, \ldots, k$. These k systems must now be independently solved in parallel. (If $k = 0$ then x is a solution and the algorithm accepts.)

Ranks 1 and 2. It should be clear that the above algorithm yields a solution whenever one exists. To obtain a decision procedure we must be able to give an upper bound for the number of steps in which a solution should be found. The crucial issue is if we can limit the number of constraints. This is possible for ranks 1 and 2. Indeed, if an initial problem $\emptyset \vdash X : \tau$, with $\mathrm{rank}(\tau) = n$, leads to a system $\Gamma_1 \vdash X : \tau_1, \ldots, \Gamma_n \vdash X : \tau_n$, then the ranks of all types in Γ_i are at most $n-1$. For $n = 1$ this means that all types in Γ_i are simple, and after a linear number of steps we have no intersections left in the constraints. Since all types in the constraints are subformulas of the original input, the number of transformations is at most polynomial, quite like in the simply-typed case. Inhabitation for rank 1 is thus in PSPACE, the same as for propositional intuitionistic logic.

For rank 2 we still can give a linear bound on the number of constraints. Intersections can occur in Γ_i, but only at the top level, and therefore no intersection can be moved (in case 3) to the right-hand side of the $\vdash$ sign. There are no more parallel constraints than top-level intersections in the input type.

An input τ of size n has at most n subformulas. With at most n constraints this makes n^2 choices of types at the rhs of $\vdash$. After every n^2 steps we must add a new variable at the lhs, in order not to get into a loop. A variable can be assigned any of n types in each of the n environments, and this is n^n different type assignments. Therefore the number of steps we can reasonably make is at most $n^n \cdot n^2 \leq 2^{n^2}$ a.e. This is alternating exponential time, so the upper bound we obtained is exponential space, cf. [8].

We show that the exponential space upper bound is tight: the problem is EXPSPACE-complete. This improves the result of [10], where it was shown that the problem is EXPTIME-hard, via simulation of linear bounded alternating machines. We use a stronger model, a *bus machine*, which is like a linear bounded machine, capable of expanding its own program by creating new instructions on the run. Each of the new instructions is of severely limited applicability (it can be used only when the tape contents is equal to a specific word), but there may be plenty of them, and this is where the exponential storage is available. Bus machines and rank 2 types are discussed in Section 3.

Rank 3 and up. For types of rank 3 the number of constraints can grow. Here is an example from [20]. Let $\alpha = ((a \to a) \to 1) \cap ((b \to a) \to 2) \to 1$, $\beta = ((a \to b) \to 1) \cap ((b \to b) \to 2) \to 2$, $\gamma = ((c \to a) \to a) \to 1$, and $\delta = ((c \to b) \to a) \to 2$. Then the type $\alpha \cap \beta \to \gamma \cap \delta \to c \to 1$ has infinitely many inhabitants, for example $\lambda xyz.x(\lambda u_1.x(\lambda u_2.x(\lambda u_3.y(\lambda v.u_1(u_2(u_3 v))))))$. There are arbitrarily long runs of our algorithm without repeating the steadily growing set of constraints.

Although types of rank 3 could create a potentially infinite search space, the "vertical" information-passing used in [20] required the fourth rank, and the problem of rank 3 remained open. It turns out that it is also undecidable. We treat this case in Section 2 using an encoding of a restricted form of emptiness problem for a simple class of semi-Thue systems.

2 Expanding Tape

Let $\mathcal{A}$ be a finite alphabet, with a designated subset $\mathcal{E} \subseteq \mathcal{A}$ of *end symbols*. A *simple switch* over $\mathcal{A}$ is a pair of elements of $\mathcal{A}$, written $a \leftarrow b$. A *labeled switch* is a quadruple, written $a \leftarrow b(c \leftarrow d)$, where the simple switch $c \leftarrow d$ is the *label.* Finally, a *splitting switch* or *split*, is a triple $a \leftarrow b \cdot c$. We require that:

– In a simple switch $a \leftarrow b$, if $b \in \mathcal{E}$ then $a \in \mathcal{E}$;
– In a labeled switch $a \leftarrow b(c \leftarrow d)$, if $b \in \mathcal{E}$ then $a \in \mathcal{E}$ and $c, d \notin \mathcal{E}$;
– In a split $a \leftarrow b \cdot c$, always $a \in \mathcal{E}$ and $b \notin \mathcal{E}$.
We draw switches (resp. simple, labeled, and splitting) as follows:

$$\begin{tikzcd}
a & a & a & \\
b \arrow[u] & b \arrow[u, "c\leftarrow d"'] & b \arrow[u] & c \arrow[ul]
\end{tikzcd}$$

The three forms of switches represent types of shape $b \to a$, $((d \to c) \to b) \to a$, and $b \cap c \to a$, respectively, and this is reflected by the direction of the arrows. One can also read the arrow as an assignment: for instance the switch $a \leftarrow b$ (resp. $a \leftarrow b \cdot c$) indicates an option to replace a by b (resp. bc).

An *expanding tape machine (etm)* is a device operating on words with help of *instructions*, each instruction consisting of a number of switches. Executing an instruction I on a word amounts to replacing every symbol a of the word by another symbol b, using a switch of the form $a \leftarrow b$ or $a \leftarrow b(c \leftarrow d)$ provided by I. An exception is the last symbol; if it is an end symbol then it can be replaced by two, using a split occurring in I. This is illustrated by the drawing below: a computation of an etm (starting with a single symbol) may be represented as a tree (expanding only to the right); symbols at each level of the tree, read from left to right, show the contents of the tape in the consecutive steps.

$$\begin{tikzcd}
0 & & \\
b \arrow[u] & c \arrow[ul] & \\
d \arrow[u, "a\leftarrow e"] & h \arrow[u, "f\leftarrow g"'] & \\
a \arrow[u] & f \arrow[u] & f \arrow[ul] \\
e \arrow[u] & g \arrow[u] & g \arrow[u]
\end{tikzcd}$$

In our example, the computation begins with 0 and the first level of the tree is obtained by applying the split $0 \leftarrow b \cdot c$. The next step uses two switches $b \leftarrow d(a \leftarrow e)$, and $c \leftarrow h(f \leftarrow g)$, etc. (Note that labels $a \leftarrow e$ and $f \leftarrow g$ introduced in the second step are later used as switches.) The contents of the tape is initially 0 and then bc, dh, aff, and egg. End symbols are 0, c and h.

In order to precisely explain how all this happens, we define an etm as a tuple of the form $\mathcal{M} = \langle \mathcal{A}, 0, 1, \mathcal{I} \rangle$, where $\mathcal{A}$ is a finite alphabet, $0, 1 \in \mathcal{A}$ are the *initial* and *final* symbols, and $\mathcal{I}$ is the set of *global instructions*. Every global instruction is a finite set of switches, and we assume that either all switches in an instruction $I \in \mathcal{I}$ are labeled, or no labeled switch belongs to I.

A sequence $\langle a_1 \leftarrow b_1, \ldots, a_k \leftarrow b_k \rangle$ of simple switches is called a *local instruction* of length k. A *configuration* of an etm is a pair of the form $C = \langle w, \mathcal{J} \rangle$, where w is a word over $\mathcal{A}$ (in which all symbols, except possibly the last one, are not active) and $\mathcal{J}$ is a set of local instructions of lengths not exceeding the length of w. The *initial configuration* is $C_0 = \langle 0, \emptyset \rangle$, and configurations of the form $\langle 1^k, \mathcal{J} \rangle$ are *final*.

A single move of a machine is a result of applying either a global instruction (a *global step*) or a local one (a *local step*). The latter is easier to explain. Suppose that $C = \langle w, \mathcal{J} \rangle$ and $I = \langle a_1 \leftarrow b_1, \ldots, a_k \leftarrow b_k \rangle \in \mathcal{J}$, where $k \leq |w|$. The instruction is applicable provided $w = a_1 \ldots a_{k-1} a_k \ldots a_k$, and the resulting ID is $C' = \langle b_1 \ldots b_{k-1} b_k \ldots b_k, \mathcal{J} \rangle$. We then write $C \Rightarrow^I_{\mathcal{M}} C'$, omitting the indices when possible. There may be more than one applicable instruction (the machine is nondeterministic). Now we describe the action of global instructions $I \in \mathcal{I}$:

– Let $w = a_1 \ldots a_n$ and $w' = b_1 \ldots b_n$. If for every $i \leq n$ the simple switch $a_i \leftarrow b_i$ belongs to I then $\langle w, \mathcal{J} \rangle \Rightarrow^I_{\mathcal{M}} \langle w', \mathcal{J} \rangle$ holds for any $\mathcal{J}$.
– Let $w = a_1 \ldots a_n$ and $w' = b_1 \ldots b_n b_{n+1}$. If all the switches
$$a_1 \leftarrow b_1,\ \ldots,\ a_{n-1} \leftarrow b_{n-1},\ a_n \leftarrow b_n \cdot b_{n+1},$$
belong to I then $\langle w, \mathcal{J} \rangle \Rightarrow^I_{\mathcal{M}} \langle w', \mathcal{J} \rangle$, for every $\mathcal{J}$.
– Let $w = a_1 \ldots a_n$ and $w' = b_1 \ldots b_n$. If $a_i \leftarrow b_i(c_i \leftarrow d_i)$ is in I, for all $i \leq n$, then $\langle w, \mathcal{J} \rangle \Rightarrow^I_{\mathcal{M}} \langle w', \mathcal{J}' \rangle$, where $\mathcal{J}' = \mathcal{J} \cup \{\langle c_1 \leftarrow d_1, \ldots, c_n \leftarrow d_n \rangle\}$.

For instance, suppose that $\mathcal{I}$ contains the global instructions $I_1 = \{0 \leftarrow b{\cdot}c, \ldots\}$, $I_2 = \{b \leftarrow d(a \leftarrow e), c \leftarrow e(f \leftarrow g), \ldots\}$, $I_3 = \{d \leftarrow a, e \leftarrow f \cdot f, \ldots\}$. Our example tree illustrates a computation which begins with an application of I_1, followed by I_2, and I_3, and then followed by an application of the local instruction $I' = \langle a \leftarrow e, f \leftarrow g \rangle$, created by the earlier use of I_1.

The notation $C \Rightarrow C'$ means that $C \Rightarrow^I_{\mathcal{M}} C'$, for some I, and the symbol $\Rrightarrow$ denotes as usual the transitive and reflexive closure of $\Rightarrow$.

Observe an essential difference between global and local instructions. Global instructions provide for additional nondeterminism: one can pick any of the switches and apply them in any order, with or without repetitions. A local instruction, created by an earlier global step, is deterministic: it applies in exactly one way, in a strictly defined context.

A configuration C is *accepting* iff $C \Rrightarrow C_1$, for some final configuration C_1. We say that an etm $\mathcal{M}$ *halts* when the initial configuration C_0 is accepting. The *halting problem* for etm is to determine if a given machine halts.

Encoding an etm with intersection types. Given an etm $\mathcal{M}$, we construct an environment Γ, of types of rank at most 2, such that $\Gamma \vdash X : 0$ for some X (where 0 is a designated type variable) if and only if $\mathcal{M}$ halts. The assumptions in Γ represent global instructions of $\mathcal{M}$ as follows:

– An instruction of the form $I = \{a_t \leftarrow b_t(c_t \leftarrow d_t) \mid t \in T\}$ is represented by a variable x_I of type $\bigcap_{t \in T}(((d_t \to c_t) \to b_t) \to a_t)$.
– An instruction $I = \{a_t \leftarrow b_t \mid t \in T\} \cup \{a_s \leftarrow b_s, c_s \mid s \in S\}$ is represented by a variable x_I of type $\bigcap_{t \in T}(b_t \to a_t) \cap \bigcap_{s \in S}(b_s \cap c_s \to a_s)$.

In addition Γ contains the declaration $y : 1$.

A system of constraints $\Gamma_1 \vdash X : a_1, \ldots, \Gamma_n \vdash X : a_n$, where a_i are type variables, *represents* a configuration $C = \langle w, \mathcal{J} \rangle$ when $w = a_1 \cdots a_n$ and:

– $\Gamma_j = \Gamma \cup \Delta_j$, where $\mathrm{Dom}(\Delta_j) = \{x_J \mid J \in \mathcal{J}\}$, for each $j = 1, \ldots, n$;
– If $J = \langle a_1 \leftarrow b_1, \ldots, a_k \leftarrow b_k \rangle$ belongs to $\mathcal{J}$ then $\Delta_j(x_J) = b_j \to a_j$ for $j \leq k$, and $\Delta_j(x_J) = b_k \to a_k$, otherwise.

Lemma 1. *The system representing C has a solution iff C is accepting.*

Proof. The proof from left to right is by induction with respect to the size of a normal solution X. First suppose that the solution is a variable. That can only happen when $a_i = 1$, for all i, and $X = y$. In this case C itself is a final configuration.

Otherwise the term X must be an application of the form $x_I X'$, where I is either a global or a local instruction. First suppose that for every $j \leq n$ we have $\Gamma_j \vdash x_I : b_j \to a_j$, for some b_j, and X' is a solution of the constraints $\Gamma_1 \vdash X' : b_1, \ldots, \Gamma_n \vdash X' : b_n$. This system represents a certain configuration C' such that $C \Rightarrow C'$. Since X' is shorter than X, we can apply induction and conclude that C' (and thus also C) is accepting.

If $\Gamma_j \vdash x_I : b_j \to a_j$, for all $j < n$, and $\Gamma_n \vdash x_I : b_n \cap c_n \to a_n$, then we have $\Gamma_1 \vdash X' : b_1, \ldots, \Gamma_n \vdash X' : b_n, \Gamma_n \vdash X' : c_n$ and the induction applies as well.

The remaining case is when all switches of I are labeled (and I is global) and $\Gamma_j \vdash X' : (d_j \to c_j) \to b_j$, for all $j \leq n$. Now X' must necessarily be an abstraction, $X' = \lambda z\, X''$, and X'' is a solution of the system

$$\Gamma_1, z : d_1 \to c_1 \vdash X'' : b_1, \ldots\ldots, \Gamma_n, z : d_n \to c_n \vdash X'' : b_n.$$

The sequence $\langle\, c_1 \leftarrow d_1, \ldots, c_n \leftarrow d_n \,\rangle$ is nothing else but the local instruction obtained from C by applying I. Therefore, the new system represents a configuration obtained from C in one step, and we can again use induction.

This completes the left-to-right part. The converse is shown by induction with respect to the number of steps from C to a final configuration. □

Undecidability of the halting problem. Lemma 1 reduces the halting problem for etm to the inhabitation problem for types of rank 3. We now show that the halting problem is undecidable using a restricted class of semi-Thue systems. A semi-Thue system over an alphabet $\mathcal{B}$, where each rule has the form $st \Rightarrow uv$, for some $s, t, u, v \in \mathcal{B}$, is called a *simple semi-Thue system (ssts).*

Lemma 2. *For an ssts it is undecidable if there exists $w \in \mathcal{B}^*$ with $w \twoheadrightarrow 1^{|w|}$.*

Proof. Easy reduction from emptiness for linear bounded automata [7]. □

In the remainder of this section we assume a fixed ssts over an alphabet $\mathcal{B}$ with $1 \in \mathcal{B}$, and with m rewriting rules of the form $s_i t_i \Rightarrow u_i v_i$ $(i = 1, \ldots, m)$. We construct an etm $\mathcal{M}$ such that $\mathcal{M}$ halts iff there exists a word w that can be rewritten to a string of 1's. The alphabet $\mathcal{A}$ of the machine contains $\mathcal{B}$, and a number of additional symbols. More precisely,

$$\mathcal{A} = \mathcal{B} \cup \{0, 0', 0'', *, \$, \circ, \bullet\} \cup \bigcup\{\{\ell_i, r_i, \$_i, ?i, i?, !i, i!\} \mid i = 0, \ldots, m\}.$$

The initial symbol is 0, and the global instructions are as follows. First there are:

$$I_0 = \{0 \leftarrow 0' \cdot 0''\} \text{ and } I_1 = \{0' \leftarrow \ell_0,\ 0'' \leftarrow r_0 \cdot \$_0\}.$$

Then, for every $i = 1, \ldots m$, we have a global instruction

$$D_i = \{* \leftarrow *\, (\circ \leftarrow \bullet),\ \ell_{i-1} \leftarrow \ell_i (i? \leftarrow i!),\ r_{i-1} \leftarrow r_i (?i \leftarrow !i),\ \$_{i-1} \leftarrow \$_i (\circ \leftarrow \bullet)\},$$

and there are two more:

$$D_0 = \{* \leftarrow *,\ \ell_m \leftarrow *,\ r_m \leftarrow \ell_0,\ \$_m \leftarrow r_0, \$_0\}$$
$$D_\infty = \bigcup\{\{* \leftarrow x, \ell_m \leftarrow x, r_m \leftarrow x\} \mid x \in \mathcal{B}\} \cup \{\$_0 \leftarrow \$\}.$$

Finally, for $i = 1, \ldots, m$, there is an instruction:

$$E_i = \{x \leftarrow \circ(\bullet \leftarrow x) \mid x \in \mathcal{B} \cup \{\$\}\} \cup \{s_i \leftarrow i?(i! \leftarrow u_i),\ t_i \leftarrow ?i(!i \leftarrow v_i)\}.$$

The machine $\mathcal{M}$ has no more global instructions. We now explain their meaning and use. The only instructions using $0, 0', 0''$ are I_0 and I_1, so the computation begins with I_0 followed by I_1 and we obtain the word $\ell_0 r_0 \$_0$.

Instructions D_i, for $i = 0, \ldots, m$ are to be used in an initial phase of the computation and their intended role is twofold: to expand the tape to a sufficient length and to "declare" local instructions to be later used to simulate rewriting rules. Here is a graphic representation of the switches of D_i $(i > 0)$ and D_0:

$$\begin{tikzcd}
* & \ell_{i-1} & r_{i-1} & \$_{i-1} & & * & \ell_m & r_m & \$_m & \\
* \arrow[u, "\circ\leftarrow\bullet"'] & \ell_i \arrow[u, "i?\leftarrow i!"'] & r_i \arrow[u, "?i\leftarrow !i"'] & \$_i \arrow[u, "\circ\leftarrow\bullet"'] & & * \arrow[u] & * \arrow[u] & \ell_0 \arrow[u] & r_0 \arrow[u] & \$_0 \arrow[ul]
\end{tikzcd}$$

Suppose for a moment that the tape contents of our machine has the form $** \cdots * \,\ell_0 r_0 \$_0$ with $j \geq 0$ stars. The only applicable instruction is D_1, and the next configuration must contain the word $** \cdots * \,\ell_1 r_1 \$_1$. Then we have $** \cdots * \,\ell_i r_i \$_i$, for all $i \leq n$, ending up with $** \cdots * \,\ell_m r_m \$_m$. If we now apply the instruction D_0 then the resulting word is $** \cdots ** \ell_0 r_0 \$_0$, with $j+1$ stars. This can be iterated a number of times. The iteration can terminate only by an application of D_∞, which yields an arbitrary word of the form $a_1 \ldots a_n \$$, where $n \in \mathbb{N}$, and $a_j \in \mathcal{B}$, for all j. No further tape expansion is possible. Each of the local instructions generated during the initial phase has the form $\langle \circ \leftarrow \bullet, \ldots, \circ \leftarrow \bullet, i? \leftarrow i!,\ ?i \leftarrow !i,\ \circ \leftarrow \bullet \rangle$, and can be used to replace $\circ \circ \cdots \circ\ \ i?\ ?i\ \circ \cdots \circ$ by $\bullet \bullet \cdots \bullet\ \ i!\ !i\ \bullet \cdots \bullet$. The two switches $i? \leftarrow i!$, $?i \leftarrow !i$ occur next to each other at an arbitrary position (because the i-th rewrite rule can be used anywhere in the word).

The next phase of computation is the simulation of the rewriting. Our intention is that the machine should halt iff the word $a_1 \ldots a_n$ can be rewritten into 1^n. The global instructions E_i serve this purpose, and each E_i simulates the rule $s_i t_i \Rightarrow u_i v_i$. The switches of E_i can act on a word of the form $b_1 \ldots b_n \$$ in a quite arbitrary way, but consider the next step after E_i. The only instructions that may be applicable are the local ones defined during the initial phase. So the only way to avoid a deadlock is that E_i uses each of the symbols $i?$ and $?i$ exactly once and places them next to each other. All other symbols written by E_i must be $\circ$. Then a local instruction is applied, and then another local instruction: one generated by an application of E_i. Figure 1 illustrates the situation occurring when this was the last application of E_i (assuming that $b_{j+1} = s_i$, $b_{j+2} = t_i$). The bottom line of Figure 1 is exactly what one obtains by applying rule $s_i t_i \Rightarrow u_i v_i$ to the word $b_1 \ldots b_n$. But it may happen that another application of E_i (at the same position in the word) took place earlier in the computation, a trace of this being left in memory in the form of the appropriate local instruction. This local instruction can be applied as well, yielding a word $c_1 \ldots c_j u_i v_i c_{j+3} \ldots c_n \$$ rather than the intended $b_1 \ldots b_j u_i v_i b_{j+3} \ldots b_n \$$. Note that in this case the word $c_1 \ldots c_j s_i t_i c_{j+3} \ldots c_n \$$ must have earlier occurred in the computation, and thus $c_1 \ldots c_j u_i v_i c_{j+3} \ldots c_n$ represents a result of a legitimate rewriting sequence from $a_1 \ldots a_n$. Therefore we can state:

$$\begin{tikzcd}
b_1 & \cdots & b_j & s_i & t_i & b_{j+3} & \cdots & b_n & \$ \\
\circ \arrow[u, "\bullet\leftarrow b_1"'] & \cdots & \circ \arrow[u, "\bullet\leftarrow b_j"'] & i? \arrow[u, "i!\leftarrow u_i"'] & ?i \arrow[u, "!i\leftarrow v_i"'] & \circ \arrow[u, "\bullet\leftarrow b_{j+3}"'] & \cdots & \circ \arrow[u, "\bullet\leftarrow b_n"'] & \circ \arrow[u, "\bullet\leftarrow \$"'] \\
\bullet \arrow[u] & \cdots & \bullet \arrow[u] & i! \arrow[u] & !i \arrow[u] & \bullet \arrow[u] & \cdots & \bullet \arrow[u] & \bullet \arrow[u] \\
b_1 \arrow[u] & \cdots & b_j \arrow[u] & u_i \arrow[u] & v_i \arrow[u] & b_{j+3} \arrow[u] & \cdots & b_n \arrow[u] & \$ \arrow[u]
\end{tikzcd}$$

Fig. 1. Hiding and restoring a word

Lemma 3. *Assuming definitions as above, the machine $\mathcal{M}$ halts if and only if there exists a non-empty word $a_1 \dots a_n$ which rewrites to 1^n.*

Proof. The hard part is "only if". As we have seen, an arbitrary computation of $\mathcal{M}$ must begin with an initial iteration yielding a word of the form $a_1 \dots a_n\$$. By induction with respect to the number of steps we then show that every third configuration must have the form $\langle w, \mathcal{J} \rangle$ where $a_1 \dots a_n \twoheadrightarrow w$. If $\mathcal{M}$ halts then $a_1 \dots a_n \twoheadrightarrow 1^n$. The "if" direction goes by induction with respect to the number of rewriting steps. □

Theorem 4. *The inhabitation problem is undecidable for rank 3.*

Proof. Lemma 3 implies that the halting problem for etm is undecidable. This, together with Lemma 1, yields the undecidability of inhabitation in rank 3. □

3 Bus Machines

To investigate the complexity of inhabitation in rank 2 we consider another machine model, the *bus machine*. Unlike an etm, a bus machine operates on a word (bus) of a fixed length. However, the instructions are more complex, and a bus machine is an alternating, rather than just nondeterministic, device. Alternation is introduced by an additional form of a switch: a *universal switch* is a triple, written $a \leftarrow b \times c$, which corresponds to a type of the form $c \to b \to a$, i.e. to an implication with two premises. (N.B. universality has nothing to do with the intersection operator.) Needless to say, splitting switches are not permitted.

Formally, we define a bus machine as a tuple $\mathcal{M} = \langle \mathcal{A}, m, w_0, w_1, \mathcal{I} \rangle$, where $\mathcal{A}$ is a finite alphabet, $m > 0$ is the *bus length* of $\mathcal{M}$ (the length of the words processed), w_0 and w_1 are words of length m over $\mathcal{A}$, called the *initial* and *final word*, respectively, and $\mathcal{I}$ is again called the set of *global instructions*.

Every global instruction is an m-tuple $\mathbb{I} = \langle I_1, \dots, I_m \rangle$ of sets of switches. Switches in I_i are meant to act on the i-th symbol of the bus. It is required that all switches in a given instruction $\mathbb{I}$ are of the same kind: either all are simple, or all are labeled, or all are universal. Therefore we classify instructions as simple, labeled, and universal. A *local instruction* is an m-tuple of simple switches.

A *configuration* of $\mathcal{M}$ is a pair $\langle w, \mathcal{J} \rangle$, where w is a word over $\mathcal{A}$ of length m, and $\mathcal{J}$ is a set of local instructions. The *initial* configuration is of course $\langle w_0, \emptyset \rangle$, and any configuration of the form $\langle w_1, \mathcal{J} \rangle$ is called *final*. Local steps of computation are defined exactly as in the case of an etm. Global steps differ in that the i-th component of a global instruction applies to the i-th symbol of the tape (bus). To make it more precise, suppose that $\mathbb{I} = \langle I_1, \ldots, I_m \rangle$, and let $w = a_1 \ldots a_m$ and $w' = b_1 \ldots b_m$, $w'' = c_1 \ldots c_m$.

- If $\mathbb{I}$ is a simple instruction, and for every $i \leq m$ the simple switch $a_i \leftarrow b_i$ belongs to I_i, then $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{\mathcal{M}} \langle w', \mathcal{J} \rangle$;
- If for every $i \leq m$ there is $a_i \leftarrow b_i(c_i \leftarrow d_i)$ in I_i, then $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{\mathcal{M}} \langle w', \mathcal{J}' \rangle$, where $\mathcal{J}' = \mathcal{J} \cup \{\langle c_1 \leftarrow d_1, \ldots, c_m \leftarrow d_m \rangle\}$;
- If $\mathbb{I}$ is universal and $a_i \leftarrow b_i \times c_i$ is in I_i, for $i \leq m$, then $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{M} \langle w', \mathcal{J} \rangle$, and also $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{M} \langle w'', \mathcal{J} \rangle$.

A configuration $\langle w, \mathcal{J} \rangle$ is *accepting* iff it is either a final configuration, or

- There exists a non-universal instruction $\mathbb{I}$, such that $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{\mathcal{M}} \langle w', \mathcal{J}' \rangle$ and $\langle w', \mathcal{J}' \rangle$ is accepting, or
- There is a universal instruction $\mathbb{I}$ such that we have $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{\mathcal{M}} \langle w', \mathcal{J} \rangle$ and $\langle w, \mathcal{J} \rangle \Rightarrow^{\mathbb{I}}_{\mathcal{M}} \langle w'', \mathcal{J} \rangle$, where both $\langle w', \mathcal{J} \rangle$ and $\langle w'', \mathcal{J} \rangle$ are accepting.

The machine $\mathcal{M}$ *halts* iff the initial configuration is accepting.

Example. This example, based on [10], should give some hint about bus programming. Let $I = \{0 \leftarrow 0, 1 \leftarrow 1\}$, $I^+ = \{0 \leftarrow 1\}$, and $I^- = \{1 \leftarrow 0\}$. Take $\mathcal{M} = \langle \mathcal{A}, 4, 0000, 1111, \mathcal{I} \rangle$, where $\mathcal{I}$ consists of the following tuples:

$$(I, I, I, I^+),\ (I, I, I^+, I^-),\ (I, I^+, I^-, I^-),\ (I^+, I^-, I^-, I^-).$$

The machine $\mathcal{M}$ makes $2^4 - 1$ steps and halts after it has seen all binary strings of length 4. (It uses neither alternation nor local instructions.)

Inhabitation in rank 2. We associate simple types with switches:

- $a \leftarrow b$ translates to $b \to a$;
- $a \leftarrow b(c \leftarrow d)$ translates to $((d \to c) \to b) \to a$;
- $a \leftarrow b \times c$ translates to $b \to c \to a$.

A set of switches translates to an intersection of the corresponding types.

Suppose that $\mathcal{M} = \langle \mathcal{A}, m, w_0, w_1, \mathcal{I} \rangle$ is a bus machine with $\mathcal{I} = \{\mathbb{I}_1, \ldots, \mathbb{I}_n\}$, $w_0 = a_1 \ldots a_m$, and $w_1 = b_1 \ldots b_m$. Define $\Gamma_i = \{x_0 : b_i, x_1 : \tau^i_1, \ldots, x_n : \tau^i_n\}$, where τ^i_j is the translation of the i-th coordinate of $\mathbb{I}_j$.

Lemma 5. *The machine $\mathcal{M}$ halts if and only if there exists a closed term M such that M has type a_i in every Γ_i.*

Proof. Let $J = \langle c^J_1 \leftarrow d^J_1, \ldots, c^J_m \leftarrow d^J_m \rangle$, for each $J \in \mathcal{J}$. Prove that the configuration $\langle e_1 \ldots e_m, \mathcal{J} \rangle$ is accepting if and only if there is a term M satisfying all the judgements $\Gamma^{\mathcal{J}}_i \vdash M : e_i$, where $\Gamma^{\mathcal{J}}_i = \Gamma_i \cup \{y_J : d^J_i \to c^J_i \mid J \in \mathcal{J}\}$. Proceed by induction with respect to the definition of acceptance ("only if" part) and with respect to M ("if" part). □

It follows that the halting problem for $\mathcal{M}$ reduces to the existence of an inhabitant of a rank 2 type of the form $\bigcap_{i=1}^{m}(b_i \to \tau^i_1 \to \cdots \to \tau^i_n \to a_i)$.

The exponential space hardness. Our goal is to simulate exponential space Turing Machines by means of bus machines. To this end, we exploit the identity EXPSPACE=AEXPTIME, that is, we actually simulate alternating exponential time. To begin with, assume that an alternating Turing Machine $\mathcal{T}$ is given, together with an input word $v = a_1 \dots a_n$. For simplicity we assume the following:

- $\mathcal{T}$ is a single-tape machine, working in time 2^{n^k}.
- The tape expands to the right and it is initially filled with blanks, except that the first n cells hold the input word. (Let $a_k =$ blank, when $k > n$.)
- The set of states of $\mathcal{T}$ is partitioned into final states, universal states, and existential states; the universal and existential states are called *choice states*.
- Every choice state q of $\mathcal{T}$ has exactly two successors; in a choice state the machine does not write nor move its head.
- In a deterministic state the machine always moves its head.

We construct a bus machine $\mathcal{M}$ of size polynomial in n such that $\mathcal{M}$ halts if and only if $\mathcal{T}$ accepts v. The bus of $\mathcal{M}$ should be seen as composed of 7 segments:

$$| \; d \; | \; \boldsymbol{i} \; | \; \boldsymbol{x} \; | \; \boldsymbol{y} \; | \; \boldsymbol{z} \; | \; q \; | \; Q \; |$$

to be interpreted as follows:

1. The presently scanned tape symbol d of $\mathcal{T}$ (the *symbol segment*);
2. The position $\boldsymbol{i}$ of the head of $\mathcal{T}$ (the *head segment*);
3. The *left time stamp* $\boldsymbol{x}$, for the $i - 1$-st tape cell;
4. The present time $\boldsymbol{y}$ (the *clock segment*);
5. The *right time stamp* $\boldsymbol{z}$, for the $i + 1$-st tape cell;
6. The present state q of $\mathcal{T}$ (the *state segment*);
7. The state Q of $\mathcal{M}$'s own control (the *control segment*).

Segments 1, 6, and 7 are single symbols. Segments 2–5 hold numbers up to 2^{n^k} in binary, i.e. use n^k space. (This is indicated by the bold symbols used.) Note that the bus length of our machine is $m = 4n^k + 3$.

The bus of $\mathcal{M}$ is too short to store a complete ID of $\mathcal{T}$. Therefore only partial information is kept on the bus, namely the internal state and the contents and number of the presently scanned tape cell. In addition we have the present time, and the two additional time stamps. Their role is to remember the time when the machine last scanned its $i-1$-st (resp. $i+1$-st) tape cell. The left time stamp is $\boldsymbol{\bot}$ when $\boldsymbol{i} = \boldsymbol{1}$, i.e., when the machine head visits the left end of tape. Also the right time stamp can be $\boldsymbol{\bot}$ (if the right neighbour cell has not been seen yet).

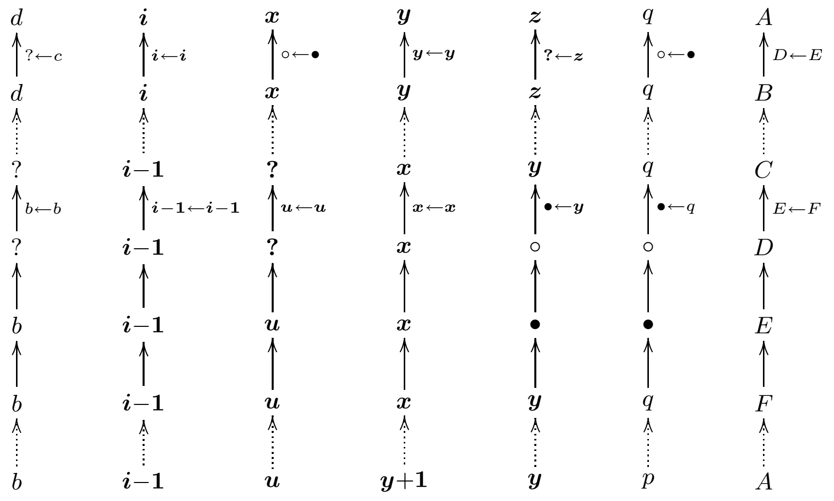
The initial word of $\mathcal{M}$ is $| \; a_1 \; | \; \boldsymbol{1} \; | \; \boldsymbol{\bot} \; | \; \boldsymbol{0} \; | \; \boldsymbol{\bot} \; | \; q_0 \; | \; A \; |$, where $\boldsymbol{1}$ represents the string $0 \dots 01$ (i.e., the binary code of the number 1). Similarly, $\boldsymbol{\bot}$ is the string of $\bot$'s, and $\boldsymbol{0}$ is the string of 0's. The final word has $*$'s everywhere.

The simulation. A configuration $\langle \, w, \mathcal{J} \, \rangle$ of $\mathcal{M}$ is called *proper* when the word w has the form $| \; d \; | \; \boldsymbol{i} \; | \; \boldsymbol{x} \; | \; \boldsymbol{y} \; | \; \boldsymbol{z} \; | \; q \; | \; A \; |$. We now show how a single step of $\mathcal{T}$ is simulated by one or more steps of $\mathcal{M}$. The case when q is a choice state is easy: no writing, no moving, just change the internal state of $\mathcal{T}$, using a global step. In deterministic states, the machine $\mathcal{T}$ moves its head to the left or to the right. To simulate this, the symbol in the first segment should be replaced by the contents of the tape cell to the left or to the right of the present head position. The machine $\mathcal{M}$ must know what symbol it is. Thus, whenever $\mathcal{T}$ leaves a particular tape cell, and the current symbol must be deleted from the bus, $\mathcal{M}$ creates a local instruction containing the information about the present time and symbol. This local instruction is left in the memory of $\mathcal{M}$ to be used at the time of return to the same tape cell. To make sure that we always use the appropriate local instruction we need the time stamps.

If $\mathcal{T}$ makes a left move from tape cell i to $i-1$, then positions $i+1, i, i-1, i-2$ of tape are called *back*, *present*, *target*, and *forward* positions, respectively. For a right move, we use these words in the opposite direction, e.g., $i-1$ is the back position. Suppose the machine leaves position $i-1$ at time x and is expected to return there at time y. Then the back, present, and target position at time x are, respectively, the forward, target, and present position at time y, because the moves made at x and at y are in the opposite directions.

Consider the case when $\mathcal{T}$, scanning d in state q, writes c, moves its head to the left and enters state p. The simulation of such a move begins with a global labeled step, which does not affect the bus, except that the control segment is set to B. But a local instruction is created (a *message* is sent) to be used at the time of return. It contains the information about the current time $\boldsymbol{y}$, the symbol c to be written, and the right (back) time stamp $\boldsymbol{z}$. These parts of the bus can now be safely erased. This is illustrated by the top two rows in Figure 2, where the arrows between many-symbol segments abbreviate multiple switches, and the same applies to their labels. (For instance the switches used in the third segment are actually $0 \leftarrow 0(\circ \leftarrow \bullet)$ and $1 \leftarrow 1(\circ \leftarrow \bullet)$.)

The third row in the figure is obtained after several steps of execution, involving a number of simple global instructions. The machine decreases the head segment $\boldsymbol{i}$ by one (if this is impossible, it gets stuck) and shifts $\boldsymbol{x}$ and $\boldsymbol{y}$ one



$$\begin{tikzcd}
d & \boldsymbol{i} & \boldsymbol{x} & \boldsymbol{y} & \boldsymbol{z} & q & A \\
d \arrow[u, "?\leftarrow c"'] & \boldsymbol{i} \arrow[u, "\boldsymbol{i}\leftarrow \boldsymbol{i}"'] & \boldsymbol{x} \arrow[u, "\circ\leftarrow\bullet"'] & \boldsymbol{y} \arrow[u, "\boldsymbol{y}\leftarrow\boldsymbol{y}"'] & \boldsymbol{z} \arrow[u, "?\leftarrow\boldsymbol{z}"'] & q \arrow[u, "\circ\leftarrow\bullet"'] & B \arrow[u, "D\leftarrow E"'] \\
? \arrow[u, dotted] & \boldsymbol{i-1} \arrow[u, dotted] & \boldsymbol{?} \arrow[u, dotted] & \boldsymbol{x} \arrow[u, dotted] & \boldsymbol{y} \arrow[u, dotted] & q \arrow[u, dotted] & C \arrow[u, dotted] \\
? \arrow[u, "b\leftarrow b"'] & \boldsymbol{i-1} \arrow[u, "\boldsymbol{i-1}\leftarrow\boldsymbol{i-1}"'] & \boldsymbol{?} \arrow[u, "\boldsymbol{u}\leftarrow\boldsymbol{u}"'] & \boldsymbol{x} \arrow[u, "\boldsymbol{x}\leftarrow\boldsymbol{x}"'] & \circ \arrow[u, "\bullet\leftarrow\boldsymbol{y}"'] & \circ \arrow[u, "\bullet\leftarrow q"'] & D \arrow[u, "E\leftarrow F"'] \\
b \arrow[u] & \boldsymbol{i-1} \arrow[u] & \boldsymbol{u} \arrow[u] & \boldsymbol{x} \arrow[u] & \bullet \arrow[u] & \bullet \arrow[u] & E \arrow[u] \\
b \arrow[u] & \boldsymbol{i-1} \arrow[u] & \boldsymbol{u} \arrow[u] & \boldsymbol{x} \arrow[u] & \boldsymbol{y} \arrow[u] & q \arrow[u] & F \arrow[u] \\
b \arrow[u, dotted] & \boldsymbol{i-1} \arrow[u, dotted] & \boldsymbol{u} \arrow[u, dotted] & \boldsymbol{y+1} \arrow[u, dotted] & \boldsymbol{y} \arrow[u, dotted] & p \arrow[u, dotted] & A \arrow[u, dotted]
\end{tikzcd}$$

Fig. 2. A deterministic left move

segment to the right. Then question marks are put in place of the symbol and left time stamp. The details of the construction are left to the reader; cf. [10].

The purpose of the next step is to prepare the bus to "receive" the message sent at time x, when the machine last scanned tape cell $i-1$ and moved right to cell i. In fact, the machine must guess the essential contents of the message, namely the symbol b and the time stamp $\boldsymbol{u}$ for tape cell $i-2$. (At time y this is seen as the forward time stamp, but it was the back time stamp when the message was sent.) Since the values $\boldsymbol{y}$ and q must not be lost, we "hide" them inside an auxiliary local instruction J, using a trick similar to that in Figure 1.

Now we can use the message sent at time x, i.e., the local instruction created at time x. The result is the third last row in Figure 1. Then we restore $\boldsymbol{y}$ and q using J (second last row) and enter the last phase of the simulation: writing $\boldsymbol{y}{+}\mathbf{1}$ in the clock segment, changing state q to p and resetting control to A.

At some later time t, the machine $\mathcal{T}$ may return to the i-th tape cell from the left. Then the simulation begins with the bus $\mid e \mid \boldsymbol{i}{-}\mathbf{1} \mid \boldsymbol{v} \mid \boldsymbol{t} \mid \boldsymbol{y} \mid r \mid A \mid$. After a number of steps this is replaced by $\mid ? \mid \boldsymbol{i} \mid \boldsymbol{t} \mid \boldsymbol{y} \mid \boldsymbol{?} \mid r \mid C \mid$, and then by the bus $\mid ? \mid \boldsymbol{i} \mid \circ \mid \boldsymbol{y} \mid \boldsymbol{?} \mid \circ \mid D \mid$. Now the message sent at time y is applicable. Note that we send only one message at a time, so that there is no danger of reading a wrong message: the proper one is identified by the target time stamp. The same applies to the auxiliary local instructions.

A computation step of $\mathcal{T}$ involving a right head shift is simulated by $\mathcal{M}$ in a dual way, provided that the right time stamp is not $\bot$. Otherwise (when $\mathcal{T}$ first time enters a tape cell) we only use global instructions, as shown below:

$$\begin{tikzcd}
d & \boldsymbol{i} & \boldsymbol{x} & \boldsymbol{y} & \bot & q & A \\
d \arrow[u, "?\leftarrow c"'] & \boldsymbol{i} \arrow[u, "\boldsymbol{i}\leftarrow\boldsymbol{i}"'] & \boldsymbol{x} \arrow[u, "?\leftarrow\boldsymbol{x}"'] & \boldsymbol{y} \arrow[u, "\boldsymbol{y}\leftarrow\boldsymbol{y}"'] & \bot \arrow[u, "\circ\leftarrow\bullet"'] & q \arrow[u, "\circ\leftarrow\bullet"'] & B_i \arrow[u, "D\leftarrow E"'] \\
a_{i+1} \arrow[u, dotted] & \boldsymbol{i}{+}\mathbf{1} \arrow[u, dotted] & \boldsymbol{y} \arrow[u, dotted] & \boldsymbol{y}{+}\mathbf{1} \arrow[u, dotted] & \bot \arrow[u, dotted] & p \arrow[u, dotted] & A \arrow[u, dotted]
\end{tikzcd}$$

Details of the construction are left to the reader. It is also left to the reader to define the appropriate global instruction that can reset the whole bus to a string of stars, when an accepting state of $\mathcal{T}$ appears in the 6-th segment.

Correctness. A computation of the alternating machine $\mathcal{T}$ is a tree of height 2^{n^k}. An initial segment of a branch in the tree, i.e., a sequence $C_0, C_1, \ldots, C_y$ of IDs of $\mathcal{T}$, is called a *play*. We describe conditions under which a proper configuration $\langle w, \mathcal{J} \rangle$ of $\mathcal{M}$ *represents* a play $P = C_0, C_1, \ldots, C_y$ of $\mathcal{T}$. First, we want $w = \mid d \mid \boldsymbol{i} \mid \boldsymbol{x} \mid \boldsymbol{y} \mid \boldsymbol{z} \mid q \mid A \mid$, where the meaning of the components is as in the previous subsection. For instance, q is the state in C_y, the binary word $\boldsymbol{x}$ stands for the largest $x < y$ such that the machine head in C_x is at $i-1$, etc.

Then we require that the instructions in $\mathcal{J}$ encode the machine moves made so far, as follows. For every $y' < y$, such that $\mathcal{T}$ makes a left (right) move in $C_{y'}$, there must be one local instruction in $\mathcal{J}$ of one of the forms below (cf. Figure 2):

$$\begin{tikzcd} ? & \boldsymbol{i} & \circ & \boldsymbol{y'} & \boldsymbol{?} & \circ & D \\ c \arrow[u] & \boldsymbol{i} \arrow[u] & \bullet \arrow[u] & \boldsymbol{y'} \arrow[u] & \boldsymbol{z} \arrow[u] & \bullet \arrow[u] & E \arrow[u] \end{tikzcd} \qquad \begin{tikzcd} ? & \boldsymbol{i} & \boldsymbol{?} & \boldsymbol{y'} & \circ & \circ & D \\ c \arrow[u] & \boldsymbol{i} \arrow[u] & \boldsymbol{z} \arrow[u] & \boldsymbol{y'} \arrow[u] & \bullet \arrow[u] & \bullet \arrow[u] & E \arrow[u] \end{tikzcd}$$

where the values of $\boldsymbol{i}$, c, and $\boldsymbol{z}$ are adequate for $C_{y'}$. If tape cell j was visited at time $x < y'$ and then at time $y' + 1$, but not in between, then we may also have *auxiliary* local instructions in $\mathcal{J}$, of one of two possible shapes:

$$\begin{tikzcd} b & \boldsymbol{j} & \boldsymbol{u} & \boldsymbol{x} & \bullet & \bullet & E \\ b \arrow[u] & \boldsymbol{j} \arrow[u] & \boldsymbol{u} \arrow[u] & \boldsymbol{x} \arrow[u] & \boldsymbol{y'} \arrow[u] & q \arrow[u] & F \arrow[u] \end{tikzcd} \qquad \begin{tikzcd} b & \boldsymbol{j} & \bullet & \boldsymbol{x} & \boldsymbol{u} & \bullet & E \\ b \arrow[u] & \boldsymbol{j} \arrow[u] & \boldsymbol{y'} \arrow[u] & \boldsymbol{x} \arrow[u] & \boldsymbol{u} \arrow[u] & q \arrow[u] & F \arrow[u] \end{tikzcd}$$

Although there may be more than one $\langle w, \mathcal{J} \rangle$ satisfying the above conditions, note that the local instructions representing deterministic steps are uniquely determined by the values $\boldsymbol{y'}$ of the clock.

Lemma 6. *Let a configuration $\langle w, \mathcal{J} \rangle$ represent a play $C_0, C_1, \ldots, C_y$ of $\mathcal{T}$.*

1. *If $C_0, C_1, \ldots, C_y, C_{y+1}$ is a play then $\langle w, \mathcal{J} \rangle \twoheadrightarrow \langle w', \mathcal{J}' \rangle$, where $\langle w', \mathcal{J}' \rangle$ represents $C_0, C_1, \ldots, C_y, C_{y+1}$.*
2. *If $\langle w, \mathcal{J} \rangle \twoheadrightarrow \langle w', \mathcal{J}' \rangle$, with $\langle w', \mathcal{J}' \rangle$ proper and with no proper configuration as an intermediate step, then $\langle w', \mathcal{J}' \rangle$ represents a play of the form $C_0, C_1, \ldots, C_y, C_{y+1}$.*

Proof. (1) The nontrivial case is when C_y is deterministic. The message sent at time y (to be received and executed at some time t) contains two kinds of data.

1. Known both at time y and time t:
   - the present position (at head segment), to become target at time t;
   - the present time (at clock segment), to be target time stamp at time t.
2. Known at time y but not at time t:
   - the present symbol (at symbol segment), to be target symbol at time t;
   - the back time stamp (at back segment), to be forward time stamp at t.

Data of the first kind identify the instruction uniquely. Indeed, only one message is sent at time y; when the machine returns to the same position next time, the target time stamp is y and it will never be y again, so there is no chance of confusing messages. Data of the second kind constitute the actual contents of the message. It is used to restore the information about the back/forward position that was deleted from the bus at time y.

Now if $\langle w, \mathcal{J} \rangle$ represents $C_0, C_1, \ldots, C_y$, we can see that $\mathcal{M}$ can move from $\langle w, \mathcal{J} \rangle$ to a configuration representing $C_0, C_1, \ldots, C_y, C_{y+1}$ by properly restoring the missing information. This requires creating and executing an auxiliary local instruction, and again there is no way to confuse it, because no other auxilary instruction available at time $\boldsymbol{y}$ can refer to the clock $\boldsymbol{x}$. The one created at time $\boldsymbol{y}$ uses the clock $\boldsymbol{x}$, and after step $\boldsymbol{y{+}1}$ it becomes obsolete too.

(2) We only consider the case of deterministic C_y. The machine $\mathcal{M}$ cannot really behave in a different way than described in part (1). The only nondeterminism available at time y is the guessing of a part of the message to be received. If $\mathcal{M}$ makes a wrong guess, it must get stuck, because there is no other

instruction that could be used. Any other behaviour of $\mathcal{M}$ is fully determined by the configuration. In particular the message to be received must introduce itself using the proper position and time stamp. $\square$

Lemma 7. *The Turing machine $\mathcal{T}$ accepts $a_1 \ldots a_n$ if and only if $\mathcal{M}$ halts.*

Proof. First observe that the initial configuration of $\mathcal{M}$ represents the initial play of $\mathcal{T}$. So it is enough to show for $\langle w, \mathcal{J} \rangle$ representing $C_0, C_1, \ldots, C_y$, that:

$$C_y \textit{ is an accepting ID of } \mathcal{T} \quad \Leftrightarrow \quad \langle w, \mathcal{J} \rangle \textit{ is an accepting configuration of } \mathcal{M}.$$

The proof of each direction is by induction with respect to the size of the accepting computation. That is, in the base case we deal with a final ID of $\mathcal{T}$, and we must observe that the final bus $** \cdots *$ can be directly obtained from $\langle w, \mathcal{J} \rangle$ only if w contains a final state of $\mathcal{T}$.

The induction step in the deterministic case follows directly from Lemma 6. In the existential and universal cases we must note that $\langle w, \mathcal{J} \rangle$ has exactly two proper successors with respect to $\Rrightarrow$. $\square$

The following is now immediate:

Proposition 8. *The halting problem for bus machines is* Aexptime-*complete, and therefore* Expspace-*complete.*

Theorem 9. *The inhabitation problem for rank 2 types is* Expspace-*complete.*

Proof. Follows from Lemma 5 and Proposition 8. $\square$

4 Related Results and Open Problems

There are very few positive results on decidability of inhabitation for subsystems of intersection types. One specific case is the system without rule $(I\cap)$ shown decidable in [9], by observing that the number of constraints is bounded. Some other cases are treated in [3]; unfortunately the proof given there for the system without rule $(E\cap)$ contains a gap.[1] The question of inhabitation without rule $(E\cap)$ seems to be harder than expected. We still have rule $(I\cap)$ and types of unbounded rank, so there is no immediate limit for the number of constraints, take e.g., $((\alpha \to p) \cap (\beta \to p) \to p) \to p$. On the other hand, with no elimination rule, nondeterminism is severely restricted. The conjecture that the problem is decidable seems plausible, yet at present we do not know how to prove it.

A different aspect of inhabitation emerges from the semantics-motivated systems built over a finite number of constants [2,19, Problem #13]. The results of the present paper do not apply to these questions directly, however we believe that the techniques developed here may be useful in solving these cases too.

[1] The claim $m \leq r$ on page 14 is not supported.

References

1. Alessi, F., Barbanera, F.: Strong conjunction and intersection types. In: Tarlecki, A. (ed.) MFCS 1991. LNCS, vol. 520, pp. 64–73. Springer, Heidelberg (1991)
2. Alessi, F., Barbanera, F., Dezani-Ciancaglini, M.: Intersection types and lambda models. Theoretical Computer Science 355(2), 108–126 (2006)
3. Bunder, M.W.: The inhabitation problem for intersection types. In: Harland, J., Manyem, P. (eds.) Computing: The Australasian Theory Symposium. Conferences in Research and Practice in Information Technology, vol. 77, pp. 7–14. Australian Computer Society (2008)
4. Coppo, M., Dezani-Ciancaglini, M.: An extension of basic functionality theory for lambda-calculus. Notre Dame Journal of Formal Logic 21, 685–693 (1980)
5. Dezani-Ciancaglini, M., Ghilezan, S., Venneri, B.: The "relevance" of intersection and union types. Notre Dame Journal of Formal Logic 38(2), 246–269 (1997)
6. Kfoury, A.J., Wells, J.B.: Principality and type inference for intersection types using expansion variables. Theoretical Computer Science 311(1–3), 1–70 (2004)
7. Kozen, D.: Automata and Computability. Undergraduate Texts in Computer Science. Springer, Heidelberg (1997)
8. Kozen, D.: Theory of Computation. Springer, Heidelberg (2006)
9. 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)
10. Kuśmierek, D.: The inhabitation problem for rank two intersection types. In: Ronchi Della Rocca, S. (ed.) TLCA 2007. LNCS, vol. 4583, pp. 240–254. Springer, Heidelberg (2007)
11. Leivant, D.: Polymorphic type inference. In: PoPL, pp. 88–98. ACM Press, New York (1983)
12. Liquori, L., Ronchi Della Rocca, S.: Intersection-types à la Church. Information and Computation 205(9), 1371–1386 (2007)
13. Lopez-Escobar, E.G.K.: Proof functional connectives. In: Methods in Mathematical Logic. LNMath, vol. 1130, pp. 208–221. Springer, Heidelberg (1993)
14. Mints, G.E.: The completeness of provable realizability. Notre Dame Journal of Formal Logic 30, 420–441 (1989)
15. Møller Neergaard, P., Mairson, H.G.: Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In: ICFP, pp. 138–149. ACM Press, New York (2004)
16. Pottinger, G.: A type assingment for the strongly normalizable λ-terms. In: Seldin, J.P., Hindley, J.R. (eds.) To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pp. 561–577. Academic Press, London (1980)
17. Ronchi Della Rocca, S., Roversi, L.: Intersection logic. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 414–428. Springer, Heidelberg (2001)
18. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism. Elsevier, Amsterdam (2006)
19. TLCA List of Open Problems, `http://tlca.di.unito.it/opltlca/`
20. Urzyczyn, P.: The emptiness problem for intersection types. Journal of Symbolic Logic 64(3), 1195–1215 (1999)
21. Venneri, B.: Intersection types as logical formulae. Journal of Logic and Computation 4(2), 109–124 (1994)
22. Wells, J.B.: Christian Haack. Branching types. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 115–132. Springer, Heidelberg (2002)
23. Yokouchi, H.: Embedding a second-order type system into an intersection type system. Information and Computation 117(2), 206–220 (1995)