THE JOURNAL OF SYMBOLIC LOGIC
Volume 64, Number 3, Sept. 1999

THE EMPTINESS PROBLEM FOR INTERSECTION TYPES

PAWEŁ URZYCZYN

Abstract. We study the intersection type assignment system as defined by Barendregt, Coppo and Dezani. For the four essential variants of the system (with and without a universal type and with and without subtyping) we show that the emptiness (inhabitation) problem is recursively unsolvable. That is, there is no effective algorithm to decide if there is a closed term of a given type. It follows that provability in the logic of "strong conjunction" of Mints and Lopez-Escobar is also undecidable.

§1. Introduction. There are essentially two ways to talk about types in the lambda calculus. One of them, called the "Church style" approach, is to extend the syntax of lambda terms to include type information. In the other approach, the "Curry style", types are predicates assigned to pure lambda terms. This usually takes the form of a system of type assignment rules used to derive judgements of the form "$M : \tau$", meaning that the type τ can be assigned to the pure term M. Such judgements are normally dependent on an *environment* E consisting of type assumptions for (at least) the free variables of M, thus the general form of a judgement can be written as "$E \vdash M : \tau$".

There are some decision problems immediately created by every such type assignment system. One of them, the *type reconstruction problem,* has been recently extensively studied in the literature. The type reconstruction problem is to determine, for a given lambda term M, whether it can be assigned a type, i.e., whether a judgement "$E \vdash M : \tau$" is derivable for some E and τ. Type reconstruction is decidable for simple types and ML-style polymorphism ([10], [18]), but it has been recently proved undecidable for the polymorphic systems $\mathbf{F}$ ([27]) and $\mathbf{F}_\omega$ ([24]). A related problem is *type checking*, where we ask about typability of a given term in a given environment. This is usually equivalent to type reconstruction, but not always (see [7]).

A dual question is as follows: given a type, we ask whether it may be assigned to a term. Here, we cannot ignore the environment. Indeed, any type τ can always be assigned to a variable, if the variable was so declared: the judgement "$x : \tau \vdash x : \tau$" is correct in all reasonable type systems. Thus, the *type inhabitation* or *type emptiness* problem is usually formulated as follows: given a type τ, decide whether there exists a *closed* term M, such that $M : \tau$ holds (in the empty environment).

Received 1995; revised December 22, 1997.

This work is partly supported by NSF grant CCR–9113196, by Polish KBN Grant 2 1192 91 01 and by ESPRIT Basic Research Action grant No. 7232, "Gentzen". A preliminary version [22] of this paper was presented at the 9-th IEEE Symposum *Logic in Computer Science*, Paris, France, 1994.

0022-4812/99/0014/$3.10

Under the *formulae as types* principle, known also as the "Curry-Howard isomorphism" (see [9, 11]), a non-empty type is a provable formula in the corresponding logic. Thus, type inhabitation problem is equivalent to the validity problem. For instance, type inhabitation is decidable for the simply typed lambda calculus, but not for the second-order lambda calculus, because validity is decidable for the ordinary propositional intuitionistic logic, but not for the second-order propositional intuitionistic calculus (see [16, 8]).

It is quite different however, if our type assignment system does not correspond directly to a logical system with well-known properties. This is exactly the case of the intersection types, first introduced by Coppo and Dezani in [4] and by Coppo, Dezani and Venneri in [5]. (The reader is referred to [3] or [25] for a survey.) The idea is that a term M has type $\tau \cap \sigma$ if and only if it has both types τ and σ. This creates a very general form of polymorphism — it has been first shown by Pottinger [20] that typability within the basic system of intersection types is equivalent to strong normalizability. This implies that type reconstruction is undecidable (see [15]).

Logical interpretations for intersection types via the Curry-Howard isomorphism are much less obvious than for many other type systems. This is related to the fact that intersection type systems are strictly "Curry-style" and have no immediate "Church-style" counterpart. The analysis of the system from a logical point of view can be done in two possible ways. One leads to the "strong conjunction" logic of Lopez-Escobar and Mints ([17, 19, 1]). Another idea is to modify the system so that an appropriate "Church-style" calculus can be designed; this is the approach of Venneri [26], followed by Dezani *et al* [6]. Another idea to design a Church-style system can be found also in [28]. The undecidability of provability in these logics follows from our main result.

We show the undecidability of the emptiness problem via reduction from the halting problems for queue automata. The reduction is performed in two stages: first we show that a process of constructing an inhabitant of a given type can express a behaviour of a machine. We define a class of auxiliary machines, called tree-makers, and we prove that the tree-maker halting problem can be reduced to type inhabitation. Then we show how to program tree-makers to simulate queue automata.

Some restrictions of the intersection type discipline for which the problem becomes decidable are discussed in [14]. However the problem is undecidable for a very small fragment of the system — for types of rank 4 in the Leivant hierarchy [15]. It is an open problem whether type emptiness is decidable for types of rank 3.

The paper is organized as follows: Section 2 introduces details of our intersection type assignment (we consider the basic system and its extensions with the constant ω and the subtyping relation $\leq$). In Section 3 we define tree-makers and we show the relationships between their behaviour and the existence of terms of certain types. The reduction from queue machines to tree-makers is worked out in Section 4. We discuss the rank 3 open problem in Section 5.

§2. Intersection type assignment. Our most general intersection type assignment system is essentially that of [2], and includes both the constant ω and the relation $\leq$. Types are defined by the following induction:

- Type variables and the constant ω are types;
- If σ and τ are types then $\sigma \to \tau$ and $\sigma \cap \tau$ are types;

with the additional assumptions that the operator $\cap$ is associative, commutative and idempotent. That is, an intersection type of the form $\tau_1 \cap \cdots \cap \tau_n$ does not depend on the order of components nor on the number of their occurrences. (A notational convention: intersection has a higher priority than arrow, e.g. "$\alpha \cap \beta \to \gamma$" means "$(\alpha \cap \beta) \to \gamma$".) The subtyping relation $\leq$ is the least transitive and reflexive relation satisfying the following clauses:

(1) $\sigma \leq \omega$;
(2) $\omega \leq \omega \to \omega$;
(3) $\sigma \cap \tau \leq \sigma$;
(4) $(\sigma \to \rho) \cap (\sigma \to \tau) \leq \sigma \to \rho \cap \tau$;
(5) $\sigma \leq \sigma'$ and $\tau \leq \tau'$ implies $\sigma \cap \tau \leq \sigma' \cap \tau'$;
(6) $\sigma \leq \sigma'$ and $\tau \leq \tau'$ implies $\sigma' \to \tau \leq \sigma \to \tau'$.

A *type environment* is a set E of pairs of the form $(x : \sigma)$, where x is a variable and σ is a type, such that if $(x : \sigma), (x : \sigma') \in E$ then $\sigma = \sigma'$. Thus an environment is a finite partial function from variables into types. If E is an environment, then $E(x : \sigma)$ is an environment such that

$$E(x : \sigma)(y) = \begin{cases} \sigma, & \text{if } x \equiv y; \\ E(y), & \text{if } x \not\equiv y. \end{cases}$$

The most general system consists of the following rules

(VAR) $E \vdash x : \sigma$ if $(x : \sigma)$ is in E

(Iω) $E \vdash M : \omega$

(E$\to$) $$\frac{E \vdash M : \tau \to \sigma,\ E \vdash N : \tau}{E \vdash (MN) : \sigma}$$

(I$\to$) $$\frac{E(x : \tau) \vdash M : \sigma}{E \vdash (\lambda x.M) : \tau \to \sigma}$$

(E$\cap$) $$\frac{E \vdash M : \sigma \cap \tau}{E \vdash M : \sigma}$$

(I$\cap$) $$\frac{E \vdash M : \tau,\ E \vdash M : \sigma}{E \vdash M : \tau \cap \sigma}$$

($\leq$) $$\frac{E \vdash M : \sigma}{E \vdash M : \sigma'} \quad \text{if } \sigma \leq \sigma'.$$

Note that rule (Iω) implies that every term is typable. In order to remain within the class of strongly normalizable terms, one may consider a restriction of the above system to ω-free types. Another common restriction (that does not change the class

of typable terms) is to eliminate rule $(\leq)$. This way we obtain four variants of the system:

(1) The full system;
(2) The ω-free system with rule $(\leq)$;
(3) The system without $(\leq)$ but with ω;
(4) The basic system of ω-free types without $(\leq)$.

The notation $\vdash_{\cap\omega\leq}$ will be used for judgements in the full system, while the symbols $\vdash_{\cap\leq}$, $\vdash_{\cap\omega}$ and $\vdash_{\cap}$ will stand for derivability in the appropriate subsystems.

Here is our main result:

THEOREM 2.1. *For each of the four systems defined above, it is undecidable whether for a given type τ there exists a term M such that $\vdash M : \tau$ can be derived in that system.*

Theorem 2.1 will be shown for the ω-free systems (2) and (4). This however implies the analogous result for (1) and (3), as can be easily deduced from the following fact:

LEMMA 2.2. *Let "$\vdash$" stand for either of "$\vdash_{\cap\leq}$" and "$\vdash_{\cap}$", and let "$\vdash_{\omega}$" stand for "$\vdash_{\cap\omega\leq}$" and "$\vdash_{\cap\omega}$", respectively. If M is in normal form and $E \vdash_{\omega} M : \sigma$, where E and σ are ω-free, then $E \vdash M : \sigma$.*

PROOF. For the system with "$\leq$", the above is exactly Lemma 7.2.1. in [25]. Without "$\leq$", the proof is even simpler: First note that a type derived for a term of the form $xN_1 \ldots N_k$, where x is a variable, must either be equal to ω or be ω-free. Then proceed by induction with respect to the length of derivations. ⊣

It follows that the presence of ω does not matter for the emptiness of omega-free types: if type emptiness is decidable for system (1) or system (3) then it is also decidable for (2) or (4). This allows us to forget about ω altogether. It is not so easy to forget about "$\leq$": there are types inhabited in system (2) but empty in system (4), take for example $\delta \cap (\alpha \to \beta) \cap (\alpha \to \gamma) \to \delta \cap (\alpha \to \beta \cap \gamma)$. The author at present does not know about any immediate reduction from type emptiness in system (4) to the same problem for system (2). The undecidability proof works however equally well for both. In order to unify the exposition for the two cases, in what follows we use the following notational conventions:

- the generic notation "$\vdash$" means either "$\vdash_{\cap}$" or "$\vdash_{\cap\leq}$";
- whenever "$\vdash$" means "$\vdash_{\cap\leq}$", the symbol "$\preceq$" means "$\leq$"; however, if "$\vdash$" stands for "$\vdash_{\cap}$", then "$\preceq$" denotes the relation "$\subseteq$", defined as follows:

$$\sigma \subseteq \tau \quad \text{iff} \quad \sigma = \tau \cap \tau', \text{ for some } \tau'.$$

The remaining part of this section is devoted to some technical preparation. The first lemma allows us to partially reconstruct a derivation of a given judgement.

LEMMA 2.3.

(1) *If $E \vdash MN : \tau$, and τ is not an intersection, then $E \vdash M : \sigma \to \tau'$ and $E \vdash N : \sigma$, for some σ and τ', such that $\tau' \preceq \tau$.*

(2) *If $E \vdash \lambda x.M : \rho$, and ρ is not an intersection, then $\rho = \sigma \to \tau$ and $E(x : \sigma) \vdash M : \tau$.*

(3) *If $E \vdash x : \tau$, then $E(x) \preceq \tau$.*

PROOF. (1) By an easy induction w.r.t. the length of derivations we prove the following statement: if $E \vdash MN : \tau$, then $\tau = \tau_1 \cap \cdots \cap \tau_n$, and the condition (1) is satisfied with τ replaced by any τ_i. The only nontrivial case is when $E \vdash MN : \sigma \to \rho_1 \cap \rho_2$ is obtained from $E \vdash MN : (\sigma \to \rho_1) \cap (\sigma \to \rho_2)$ via rule $(\leq)$, and this requires observing that $(\sigma_1 \to \sigma \to \rho_1) \cap (\sigma_2 \to \sigma \to \rho_2) \leq (\sigma_1 \cap \sigma_2) \to \sigma \to (\rho_1 \cap \rho_2)$.

(2) For $\vdash_{\cap\leq}$, this is a consequence of Lemma 2.8 in [2]. For $\vdash_{\cap}$, it can be easily proved by induction.

(3) Again, an easy proof by induction. ⊣

In the construction to follow we will need only types of the following forms:

(A) $\alpha_1 \cap \cdots \cap \alpha_n$, where $\alpha_1, \ldots, \alpha_n$ are type variables;

(B) $(\alpha_1 \to \beta_1) \cap \cdots \cap (\alpha_n \to \beta_n)$, where $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$ are type variables;

($\frac{1}{2}$C) $(\tau_1 \to \alpha_1) \cap (\tau_2 \to \alpha_2) \to \beta$, where $\alpha_1, \alpha_2, \beta$ are type variables, and τ_1, τ_2 are of the form (B);

(C) $\sigma_1 \cap \cdots \cap \sigma_n$, where $\sigma_1, \ldots, \sigma_n$ are of the form ($\frac{1}{2}$C).

Types of the forms (A), (B) and (C) are called *good*. A type environment E is *good* iff all types in E are good. Note that the good types are of rank at most 3 in the sense of [15]. We now state some technical properties of type judgements involving good types.

LEMMA 2.4. *Let E be a good environment. Then*:

(1) *If $E \vdash xM : \tau$ then τ is an intersection of type variables (in particular it follows that no application of the form xMN is typable in E).*

(2) *If $E \vdash xM : \alpha$, where $E(x) = (\sigma_1 \to \beta_1) \cap \cdots \cap (\sigma_n \to \beta_n)$, then $\alpha = \beta_i$ and $E \vdash M : \sigma_i$, for some i.*

PROOF. We begin with the following observation: if $(\sigma_1 \to \beta_1) \cap \cdots \cap (\sigma_n \to \beta_n) \leq \rho$, then $\rho = (\sigma'_1 \to \tau_1) \cap \cdots \cap (\sigma'_m \to \tau_m)$, and for each i, there are $j^i_1, \ldots, j^i_{p_i}$ such that $\tau_i = \beta_{j^i_1} \cap \cdots \cap \beta_{j^i_{p_i}}$ and $\sigma'_i \leq \sigma_\ell$, for all $\ell = j^i_1, \ldots, j^i_{p_i}$. Both parts of the lemma now follow from Lemma 2.3, parts (1) and (3). ⊣

LEMMA 2.5. *Let E be a good environment, and let M be in normal form. Then*:

(1) *If $E \vdash M : \alpha$, where α is a type variable, then there are two possibilities*:

– *either M is a variable z and $E(z)$ is of the form (A)*;

– *or $M \equiv xP$, where $E(x) = (\sigma_1 \to \beta_1) \cap \cdots \cap (\sigma_n \to \beta_n)$, and for some i, we have $\alpha = \beta_i$ and $E \vdash P : \sigma_i$.*

(2) *If $E \vdash M : (\tau_1 \to \alpha_1) \cap (\tau_2 \to \alpha_2)$, where τ_1, τ_2 are of the form (B), then $M \equiv \lambda u.Q$, and $E(u : \tau_1) \vdash Q : \alpha_1$ and $E(u : \tau_2) \vdash Q : \alpha_2$.*

PROOF. Part (1) follows from Lemma 2.4, since no abstraction can be of type α. For part (2) note first that by Lemma 2.4(1), the term M cannot be an application. To show that it cannot be a variable, let the *arrow depth* of a type τ, denoted $ad(\tau)$, be as follows: $ad(\alpha) = 0$, $ad(\sigma \to \tau) = 1 + \max(ad(\sigma), ad(\tau))$, and $ad(\sigma \cap \tau) = ad(\sigma)$, provided $ad(\sigma) = ad(\tau)$. Otherwise $ad(\sigma \cap \tau)$ is undefined. Call a type τ *uniform* iff $ad(\tau)$ is defined. Now it is easy to see that, for uniform types, $\sigma \preceq \tau$ implies $ad(\sigma) = ad(\tau)$. Since good types are of arrow depth 0, 1 or 3, our term M cannot be

a variable by Lemma 2.3(3). It must be an abstraction, and the hypothesis follows from Lemma 2.3(2). ⊣

The above Lemma suggests how to perform a search for an inhabitant of a given type in a good environment. The following example demonstrates a possible (nondeterministic) search behaviour. The tree-makers, to be defined in Section 3, are machines constructed to exhibit that kind of behaviour.

EXAMPLE 2.6. Let E be an environment such that each $E(x)$ is either of the form (A) or (C), and assume we look for a term M such that $E \vdash M : \alpha$, for a type variable α. With no loss of generality we can assume that M is in normal form. If there is a variable z, with $E(z) = \alpha \cap \beta_1 \cap \cdots \cap \beta_n$, then we can take $M \equiv z$. Otherwise, by part (1) of Lemma 2.5, we have $M \equiv xM'$, where $E(x)$ is of the form (C). Now, $E(x)$ is in general an intersection, and one component of that intersection must be $(\tau_1 \to \alpha_1) \cap (\tau_2 \to \alpha_2) \to \alpha$, with $E \vdash M' : (\tau_1 \to \alpha_1) \cap (\tau_2 \to \alpha_2)$. We can apply part (2) of Lemma 2.5, to obtain $M' \equiv \lambda u.N$ with $E(u : \tau_1) \vdash N : \alpha_1$ and $E(u : \tau_2) \vdash N : \alpha_2$. This means that our previous task — to find a single term of a given type in a given environment — has been transformed into a new one — to find a single term that has two given types in two given environments. One possible way to satisfy this requirement is to find a variable z such that $E(z) = \alpha_1 \cap \alpha_2 \cap \cdots$. Otherwise, N must again be an application. Suppose, for instance, that we can find a variable y with $E(y) = ((\sigma_1 \to \beta_1) \cap (\sigma_2 \to \beta_2) \to \alpha_1) \cap ((\sigma_3 \to \beta_3) \cap (\sigma_4 \to \beta_4) \to \alpha_2) \cap \cdots$. Then a possible shape of N is $y(\lambda v.P)$, where P must satisfy *four* possibly different conditions: $E(u : \tau_1, v : \sigma_1) \vdash P : \beta_1$ and $E(u : \tau_1, v : \sigma_2) \vdash P : \beta_2$ and $E(u : \tau_2, v : \sigma_3) \vdash P : \beta_3$ and $E(u : \tau_2, v : \sigma_4) \vdash P : \beta_4$.

We now look for a candidate for P. Again it could be a variable or an application zQ, where z is declared in E. (In the latter case we would obtain eight new conditions.) But it could also be an application of the form uQ or vQ. Assume for example that $P \equiv uQ$. In order to satisfy the four conditions we must have $\tau_1 = (\gamma_1 \to \beta_1) \cap (\gamma_2 \to \beta_2) \cap \cdots$ and $\tau_2 = (\gamma_3 \to \beta_3) \cap (\gamma_4 \to \beta_4) \cap \cdots$, for some variables $\gamma_1, \gamma_2, \gamma_3, \gamma_4$. The new conditions to be satisfied by Q are as follows: $E(u : \tau_1, v : \sigma_1) \vdash Q : \gamma_1$ and $E(u : \tau_1, v : \sigma_2) \vdash Q : \gamma_2$ and $E(u : \tau_2, v : \sigma_3) \vdash Q : \gamma_3$ and $E(u : \tau_2, v : \sigma_4) \vdash Q : \gamma_4$.

This process can be seen as creating a tree of conditions. The conditions on any given level are of the form "$E_i \vdash ? : \xi_i$", where ξ_i are variables, and are to be satisfied by one lambda term. The environments E_i are determined by the choices made in the previous phases of the construction, and it is quite easy to see that the whole story can be represented by the labelled tree in Figure 1. The nodes of the tree are labelled by type variables, and the labels of edges represent types of variables added to the current environment at the corresponding moment. The tree construction ends when we can find a variable z, such that $E(z)$ is of the form (A) and contains the labels of all the leaves.

§3. Tree-makers into types. A tree-maker, as the reader can easily guess, is a machine that makes trees. Starting with a single root node, it adds (level after level) new vertices of increasing depth, and assigns labels to these vertices, and to

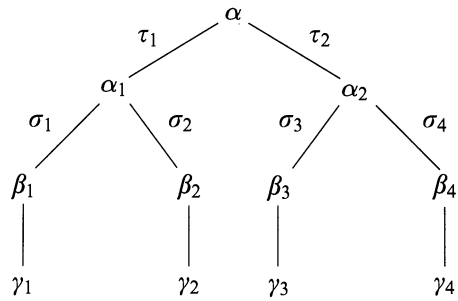


FIGURE 1

some of the edges. At each stage, a tree-maker determines a rule to be applied, and creates a next level of vertices according to the chosen rule. This process is nondeterministic and continues until all leaf nodes are labelled in a certain way. (Of course the intuition here should be that each rule corresponds to a variable available in an environment.)

In order to provide a formal definition of a tree-maker, let us fix a finite alphabet A for node labels. We begin with the notion of a "link", which represents a local labelling of a father node and its son(s). Each tree can be seen as built up from links. A *unary link* is just a pair of symbols in A, written $\alpha \leftarrow \beta$, and represents a possible labelling, that assigns α to a father node and β to the (only) son. (We prefer to draw arrows from sons to fathers rather than in the opposite direction.) A *binary link* is more complicated: it is a quintuple, written $\alpha \leftarrow (\beta, X), (\gamma, Y)$, and it is best presented on a picture:

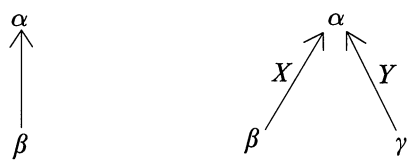


FIGURE 2. A unary link and a binary link

Here, α, β, and γ are elements of A, and X, Y are finite, nonempty sets of unary links. The sets X and Y are used as labels for the two edges. The meaning of α, β, and γ should be clear: a father node labelled α and two sons labelled β and γ. The meaning of X and Y is as follows: the unary links of X and Y are available from now on for the further construction of the tree, but not everywhere. The links of X may occur only in the left subtree (rooted at the left son), while the links of Y may be used below the right son. We say that the links in X and Y are *declared* by the binary link $\alpha \leftarrow (\beta, X), (\gamma, Y)$.

A *global rule* is a finite set of binary links. A *tree-maker* (over A) is a triple $\langle A, \alpha_0, F, \mathscr{R} \rangle$ consisting of the alphabet A, a distinguished *initial label* $\alpha_0 \in A$, a set of *final labels* $F \subseteq A$ and a finite set $\mathscr{R}$ of global rules.

A tree construction begins with a single root node labelled α_0, which is the simplest *run* of a tree-maker (of depth 0). Runs of arbitrary depth are finite trees defined by induction as follows. A run of depth n (with all leaves at depth n) can

be extended into a run of depth $n+1$ in two ways. The first possibility is to apply a global rule $R \in \mathscr{R}$, provided each leaf node is labelled with a symbol α such that a link of the form $\alpha \leftarrow (\beta, X), (\gamma, Y)$ is in R, for some β, X, γ, and Y. This means that the following steps are executed for each leaf v:

– choose an appropriate link $\alpha \leftarrow (\beta, X), (\gamma, Y)$, where α is the label of v;

– create two sons of v, the left one labelled β and the right one labelled γ;

– assign the labels X and Y to the edges leading to v from the left and right son of v, respectively.

The assignment of the sets of links to the edges leading from level $n+1$ to n is called the $n+1$-st *local rule*. Local rules differ from global rules in that the links "declared" at an edge attached at a node u (i.e., leading from u to its father) can be applied only to descendants of u. (Also, local rules involve unary links and these do not declare other links. This means the n-th local rule does not have to exist for every n.)

The second possibility of extending a run of depth n is to apply one of the local rules introduced before, say the i-th local rule ($i \leq n$). This time, for each leaf v, we perform the following steps:

– determine the ancestor of v at level i, say u, and find the set X assigned to the edge attached at u;

– choose a link $\alpha \leftarrow \beta$ from X, such that α is the label of v;

– create a single son of v labelled with β.

This time there are no declared links, in consequence the $n+1$-st local rule does not exist.

The goal is to obtain a run such that all leaf nodes are labelled by final labels (elements of F). Such a run is called *closed*. The problem we address is: given a tree-maker, does it have a closed run? The main result of this section is to show that the closed run problem effectively reduces to the emptiness problem.

LEMMA 3.1. *For a given tree-maker T, one can effectively construct a type τ such that T has a closed run iff there exists M such that $\vdash M : \tau$.*

The proof of the above lemma covers the remainder of this section. We begin with an easy observation. Let E be an environment consisting of type assignments $(x_i : \tau_i)$, for $i = 1, \ldots, n$. Then the following conditions are equivalent:

(1) there exists a term M such that $E \vdash M : \tau$;

(2) there exists a term N such that $\vdash N : \tau_1 \to \cdots \to \tau_n \to \tau$.

Indeed, if (1) holds, then we can define N as $\lambda x_1 \ldots x_n . M$. If (2) holds, one can take $M := N x_1 \cdots x_n$. It follows that it suffices to define E and τ such that T has a closed run iff $E \vdash M : \tau$, for some M.

Let $T = \langle A, \alpha_0, F, \mathscr{R} \rangle$ be a tree-maker. For simplicity, we assume that elements of the alphabet A can be used as type variables. Let $F = \{\alpha_1, \ldots, \alpha_r\}$ and let $\mathscr{R} = \{R_1, \ldots, R_m\}$. We prove that T has a closed run iff there exists M such that $E \vdash M : \alpha_0$, where E is an environment, that assigns types to $m+1$ variables: one variable y representing F, and m variables $x_1, \ldots, x_m$ representing $R_1, \ldots, R_m$.

The type assigned to y is easy to define: we take $E(y) := \alpha_1 \cap \cdots \cap \alpha_r$. To define the types $E(x_i)$, we first identify each unary link of the form $\alpha \leftarrow \beta$ with the type $\beta \to \alpha$. Then a set X of unary links is identified with an intersection of all its elements, denoted $\hat{X}$, and finally a binary link $\alpha \leftarrow (\beta, X), (\gamma, Y)$ is represented by

the type $(\tilde{X} \to \beta) \cap (\tilde{Y} \to \gamma) \to \alpha$. We define $E(x_i)$ as the intersection of all such types obtained for all the elements of the global rule R_i. Of course, the obtained environment E is good.

As it was already mentioned, the goal is to represent a closed run by a successful type assignment $E \vdash M : \alpha_0$. We think of such an assignment (with unknown M) as an initial question asked for the initial one-element run. As the run developes into a larger tree, we ask more questions — one for each leaf. More precisely, consider an arbitrary run of depth n, with k leaves $v_1, \ldots, v_k$, labelled $\beta_1, \ldots \beta_k$. We define environments E_i such that our run can be extended to a closed one iff there is a term N satisfying all the conditions $E \cup E_i \vdash N : \beta_i$.

The domain of all the E_i's is the same and consists of new variables z_j, for all such $j \leq n$ that the j-th local rule is defined. To define $E_i(z_j)$ one has to find the label X of the edge attached at the ancestor of v_i at level j. The type $E_i(z_j)$ is the intersection of all unary link types in X, i.e., all the link types available in v_i for the j-th local rule.

LEMMA 3.2. *The following conditions are equivalent for each run t of T:*
(1) *There is a term M such that $E \cup E_i \vdash M : \beta_i$, for each leaf v_i of t;*
(2) *The run t can be extended (in zero or more steps) to a closed run of T.*

PROOF. ($\Rightarrow$) Induction w.r.t. M, assumed to be in normal form. By Lemma 2.5, if M is a variable then it must be y. Thus, all the β_i's are among the final labels, and the run t is closed. Otherwise, M is an application. The first possibility is that $M \equiv z_j P$, for some j and P. Using Lemma 2.5 we obtain that for each i, the type $E_i(z_j)$ contains a component of the form $\gamma_i \to \beta_i$, where $E_i \vdash P : \gamma_i$. We extend t to a new run t' with help of the j-th local rule, and we apply the induction hypothesis to P and t'.

Another possibility is when $M \equiv x_j M'$, for some j, M'. An argument similar to that in Example 2.6 implies that $M \equiv x_j(\lambda u.P)$, for some P. In addition, for each i, there must be a component

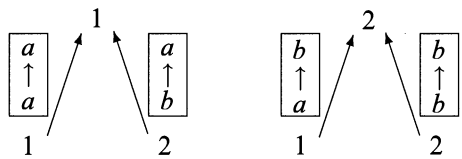
$$(*) \qquad (\tau_1^i \to \alpha_1^i) \cap (\tau_2^i \to \alpha_2^i) \to \beta_i$$

in $E(x_j)$, such that $(E \cup E_i)(u : \tau_1^i) \vdash P : \alpha_1^i$ and $(E \cup E_i)(u : \tau_2^i) \vdash P : \alpha_2^i$. Since $E(x_j)$ corresponds to the global rule R_j, we may define an extension t' of t applying at each leaf node v_i the binary link corresponding to $(*)$. If the depth of t was n, then the depth of t' is $n+1$ and the $n+1$-st local rule is now defined (with $z_{n+1} = u$). This local rule is as follows: the left edge outgoing from v_i is assigned the unary links corresponding to components of τ_1^i. Respectively, the components of τ_2^i are attached to the right edge. It remains to apply the induction hypothesis to P and t'.

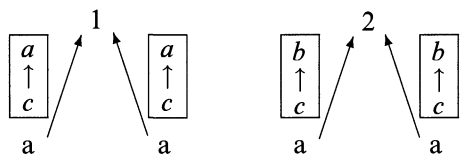
($\Leftarrow$) Induction w.r.t. the number of steps needed to complete the closed run. If no steps are needed, i.e., t is closed, then all the β_i's are final labels, and we can take $M \equiv y$. Suppose now that (1) holds for a run t' obtained from t in one step, and let M' be the appropriate term. There are two cases, let us first assume that t' has been obtained by an application of the j-th local rule. Then for each i, a link $\beta_i \leftarrow \gamma_i$ was used (which means that the corresponding type is a component of $E_i(z_j)$) and γ_i is the label of the only son of v_i. We take $M \equiv z_j M'$.

The second case is when a global rule R_j is applied. At node v_i we must have a binary link of R_j corresponding to a type of the form $(\tau_1^i \to \alpha_1^i) \cap (\tau_2^i \to \alpha_2^i) \to \beta_i$. Labels of the two sons of v_i are α_1^i and α_2^i, and the labels of the corresponding edges are sets of links representing types τ_1^i and τ_2^i. The reader can easily check that it suffices to define $M \equiv x_j(\lambda z_{n+1}.M')$. ⊣

§4. Queue automata into tree-makers. In the present section we show that the closed run problem for tree-makers is undecidable. The construction to follow is quite complex. In order to help the reader understand it, we begin with a simple exercise in tree-maker programming. Let us consider a tree-maker T_1 over the alphabet $\{1, 2, a, b, c\}$, with the initial label 1, a final label c, and with two global rules B and EB, defined as follows. Rule B consists of the following two links:



and rule EB of the following two:



The boxes in the above pictures represent one-element sets of links (edge labels). The only final label is c. Now, every run of T_1 begins with a number of applications of rule B (possibly zero), as shown below:

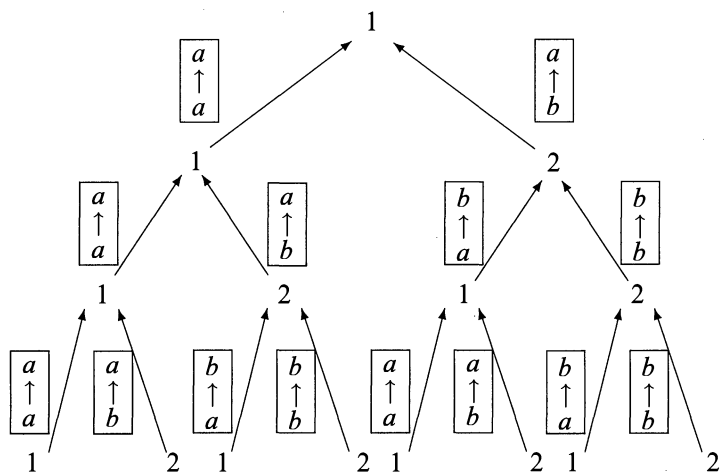


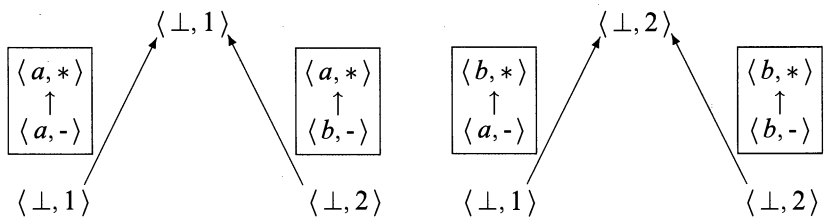
FIGURE 3. Initial phase

It should be easily seen that, in the further tree-making, the first local rule is applicable if and only if all leaf nodes carry the label a. The second local rule requires a for the leaves of the left subtree and b for the right subtree. Each subsequent local rule requires a finer distribution of the a's and b's. In particular, there will never be a choice between two different local rules.

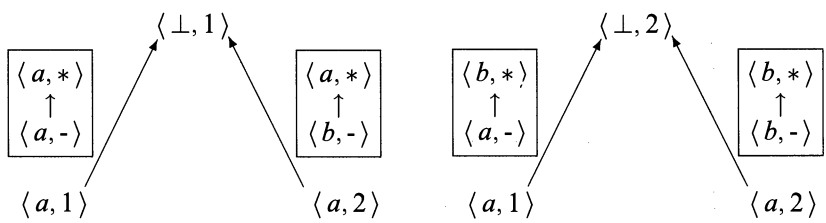
The initial phase ends when the tree-maker nondeterministically decides to execute rule EB. This assigns label a to all nodes on the currently constructed level n, and now the first local rule may be applied, followed by the second, third, and all the other local rules. A closed run is of height $2n$.

The crucial property of the above tree-maker is that the i-th local rule can be used only in the $(n+i)$-th step of the construction (i.e., applied to nodes at level $n+i-1$). We refine this idea in our next example. This time we want a run consisting of a possibly infinite number of phases, such that a local rule declared at step i of phase m is executed at step i of phase $m+1$. This can be achieved if we agree that the construction of two consecutive levels may be sometimes treated as one "double step".

The alphabet of our second tree-maker T_2 will be a product of the form $\{a, b, \bot\} \times \{1, 2\}$, so that each node may carry two labels. As long as it does not lead to a confusion we use expressions like, e.g., "a-node" or "2-node" to refer to nodes whose labels have a (resp. 2) as their first (resp. second) components. Such labels are of course called "a-labels" or "2-labels". Similar conventions will be also used later. There will be 3 global rules, denoted B, EB and G. Rule B, used in the initial phase has again two links:

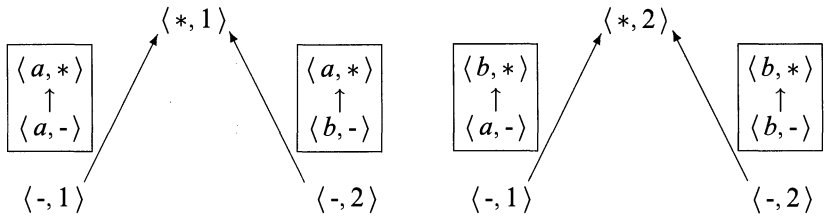


A box used as an edge label represents the set of all links that match the pattern inside the box, according to the following convention: An asterisk matches everything, and a hyphen means that the appropriate component of the son label is the same as the corresponding component of the father label. Rule EB, which ends the initial phase, also has two links:



It is easy to see that the initial phase ends with all nodes assigned a-labels, and that the applicability of local rules is determined by the distribution of a-labels and b-labels very much like in the previous example. But now we also have another

global rule G, that may be used after the initial phase ends. Rule G has four links, presented here on two pictures with help of the asterisk and hyphen convention:



A possible run of T_2 may be as follows: after the initial phase of n steps, the local and global rules begin to alternate. Each "double step" consists of an application of a local rule, determined by the first components of node labels, and an application of rule G. The latter declares a new local rule, and this local rule is used after a delay of exactly n "double steps". This naturally divides the whole run into phases: the initial phase of n "single steps", and all other phases of n "double steps". The local rules establish a natural connection between corresponding steps of the consecutive phases. This machinery is a basis of our main construction below, where we use local rules to pass information between different phases.

As it is defined now, T_2 does not have to obey the alternation pattern, it may, e.g., execute G twice in a row. In order to force the behaviour we want, we can modify the label alphabet, by adding a third component to carry the necessary information.

After this preparation, we turn to our main task: simulation of queue automata by tree-makers. A (deterministic) *queue automaton* is understood here as a numbered list of instructions taking one of the following forms:

i : $insert(0)$;
i : $insert(1)$;
i : $remove(x)$; $case\ x\ of\ 0 : j,\ 1 : k$;
i : $basta$.

The symbols i, j and k are instruction numbers. A *computation* of a queue automaton is a sequence of configurations, each consisting of an instruction number and a queue, i.e., a sequence of 0's and 1's. The queue is initially empty, and the initial instruction number is 1. An *insert* instruction numbered i places an appropriate bit at the end of queue and transfers control to the $i + 1$-st instruction. A *remove* instruction as above deletes a bit from the front of queue and passes control to j or k depending on whether that bit is 0 or 1. An attempt to execute *remove* on an empty queue is understood as a divergent loop (this may be formally defined so that the next configuration is equal to the previous one). The computation stops if a *basta* instruction is reached. Without loss of generality we can assume that there is only one such instruction, and that the initial instruction is an *insert*.

It is well known that the halting problem for queue automata (given an automaton, is the computation finite or not?) is undecidable. We show a reduction of this problem to the closed run problem for tree-makers. That is, we prove the following Lemma.

LEMMA 4.1. *There is an effective procedure to construct a tree-maker $T_{\mathscr{A}}$ from a given queue automaton $\mathscr{A}$, so that $T_{\mathscr{A}}$ has a closed run iff $\mathscr{A}$ terminates.*

All the rest is the proof of the above Lemma. Together with Lemma 3.1, it constitutes a reduction from the queue automaton halting problem to the type emptiness problem, thus proving our main Theorem 2.1.

Let us fix a queue automaton $\mathscr{A}$ with k instructions, and assume that the only *basta* instruction has number k. We will use the notation $[k]$ for the set $\{1, \ldots, k\}$. During the only computation of our automaton, the length of the queue will be changing. For technical reasons this is inconvenient, and we prefer to represent the contents of the queue as a string of constant length. We may think of the automaton as of a machine that first makes a nondeterministic guess of how much space it will need to complete the computation, and then works within that space bound (diverging if the bound is insufficient). Of course a successful computation can be done within a finite space and thus the halting problem remains the same.

We represent a queue, say "011100010", as a string "$\$\cdots\$<011100010>\sharp\cdots\sharp\sharp$" with a certain number of the \$'s and $\sharp$'s. The initial empty queue is just "$<>\sharp\cdots\sharp\sharp$". Now an *insert* instruction means: *replace ">" with a digit and replace the first "$\sharp$" with ">"*, and similarly for a *remove*. Note that the number of \$'s will increase after each *remove*, while the suffix of $\sharp$'s will shrink after each *insert*, so that the queue "moves to the right". Let the symbol Q stand for the alphabet $\{0, 1, \$, \sharp, <, >, \bot\}$ (note the extra symbol "$\bot$").

Let now w_i be the word representing the queue after i steps of the computation. The history of the queue during the computation may now be presented as the concatenation $w_0 \cdot w_0 \cdot w_1 \cdot w_2 \cdots$ (we repeat twice the initial word for technical convenience). The tree-maker we are about to define will simulate the computation so that one or two levels of nodes will be created for each symbol in the sequence. Clearly, a run is naturally divided into phases representing different configurations, i.e., the words $w_0, w_0, w_1, w_2, \ldots$

We begin the definition of our tree-maker with the label alphabet A, which is the following product:

$$A = \{a, b, \bot\} \times \{1, 2\} \times Q \times [k] \times ([k] \cup \{\bot\}) \times \{0, 1\} \times \{L, G\} \times \{t, f\}.$$

Some hints about the meaning of the components: The value of the first one will be used to determine, which of the local rules is to be executed. The assignment of a's and b's to the leaves of a run will always force the execution of a particular local rule, and moreover, each local rule will be applicable only once. The second component identifies the kind of unary links to be declared in case a global rule is executed. (The first two components are like in the example we worked out at the beginning of this section. They constitute the basic control machinery, that allows for passing information from one phase of the run to the next one.)

Unlike the first two, all the other components of a label will depend only on the depth in the tree. That is they are always the same for all nodes on any fixed level. (Recall that all these nodes represent a single symbol in the "queue history" $w_0 \cdot w_0 \cdot w_1 \cdot w_2 \cdots$.)

The third component stands for the current queue symbol, and the fourth one for the currently executed instruction. The fifth one is the next instruction (which may not be determined yet — that's why we need the "$\bot$"). Thus the third, fourth and fifth component are used to represent information about the queue automaton.

The remaining components take care of some technical details. The "1" at the sixth place means that the execution of current instruction has begun but has not been completed yet. The "L" or "G" tells if the next rule to be used is local or global, respectively. Finally, a "t" at the last component signals the end of a phase.

The initial label is the tuple "$\langle \perp, 1, <, 1, 1, 0, G, f \rangle$". The final labels are all labels that match the pattern "$\langle *, *, \perp, k, \perp, 0, L, t \rangle$" where an asterisk matches everything.

To complete the definition of our tree-maker we describe the global rules with help of the pictures below. The boxes represent edge labels, i.e., sets of all links that match the patterns inside the boxes. As before, an asterisk matches everything, and a hyphen means that the appropriate component of the son label is the same as the corresponding component of the father label. Each of the rules will consist of two kinds of links, which differ very little, and are dual to each other. In order to avoid repeating almost identical pictures, we use one more convention: the notation "$\frac{x}{y}$" should read "x" for the first kind of links, and "y" for the second.

The first global rule, denoted B, consists of all links matching the pattern on Fig. 4, where S and S' are such that S' follows S in the initial queue $<>\sharp\cdots\sharp\sharp$. This rule will be used only during the first phase of a run. Links of the first kind can be applied only to 1-nodes, and those of the second kind — only to 2-nodes.

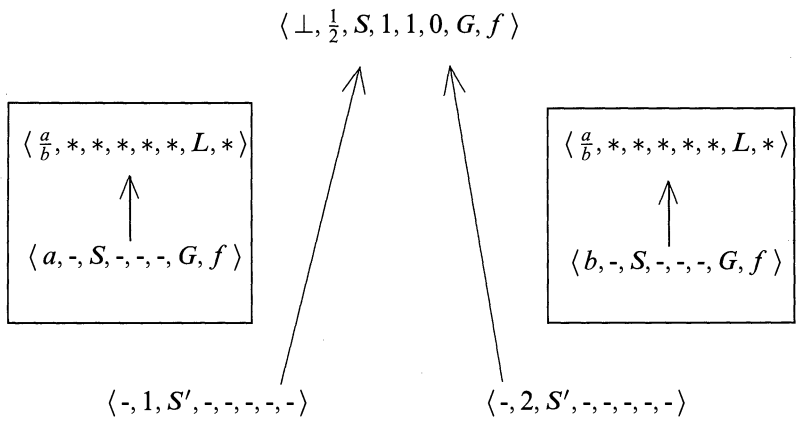


FIGURE 4. Rule B

Before we define the remaining global rules, let us observe that the initial part of a run, obtained with the rules already defined, may be schematically presented as on Fig. 3, if we mark only the relevant components of labels. As before, the first local rule is applicable only if all leaf nodes carry a-labels. The second local rule requires a-labels at the left and b-labels at the right. Further local rules need more alternations of a and b, and no two local rules are applicable at the same moment. This machinery will be present also in all the rules to be defined in the sequel.

If we write down the third components of the labels at levels $0, 1, 2, \ldots$ (all these labels are constant within any level) we obtain the sequence "$<>\sharp\sharp\sharp\ldots$" The first phase ends when the tree-maker nondeterministically guesses that the length of this

sequence is sufficient (in fact, this is the only place when nondeterminism occurs). For this, we have the global rule EB, which has only two links, shown on Fig. 5.

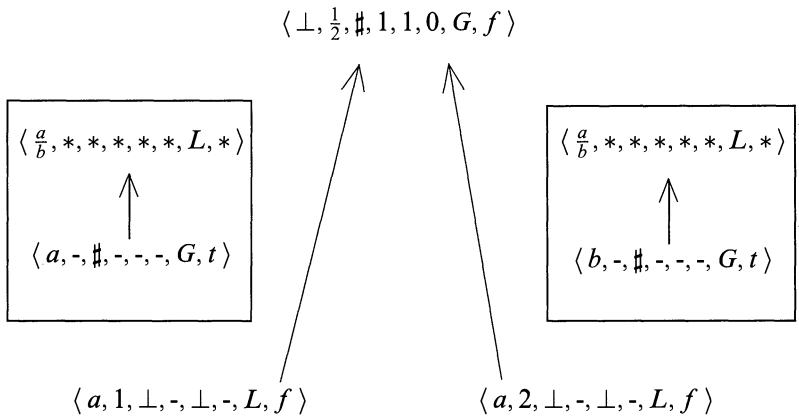


FIGURE 5. Rule EB

An execution of EB results in a run ending with a-labels assigned to all the leaves. Also, these labels are also L-labels, and this means the next rule used is the first local rule. Starting from this point, applications of local and global rules will alternate. This is because the seventh component of labels will always be set to "G" (respectively "L") by an execution of a local (respectively global) rule. On the other hand only a global rule can match a G-label, and only a local one can match an L-label. This principle is followed by all the rules defined below.

We now describe the global rules G, G' and EG, used after the first phase ends. We begin with rule G, with links of the form described on Fig. 6.

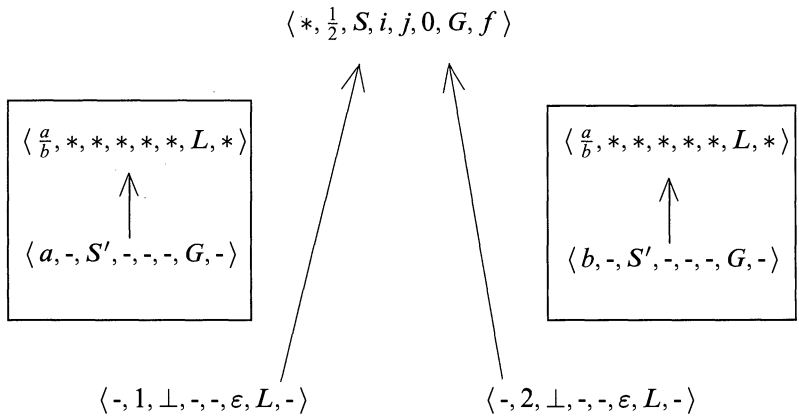


FIGURE 6. Rule G

There, S is a queue symbol and i is an instruction number. Then j is an arbitrary element of $[k] \cup \{\perp\}$, except that the case $i = j = 1$ is excluded. The number ε and the symbol S' depends on the instruction i: the symbol S' is the symbol to replace S in the next step. We take $\varepsilon = 0$ and $S' = S$, except of the following cases:

1. If i is a number of a *remove* instruction, and S is $<$, then $\varepsilon = 1$ and $S' = \$$;
2. If i is a number of an $insert(0)$ instruction, and S is $>$, then $\varepsilon = 1$ and $S' = 0$;
3. If i is a number of an $insert(1)$ instruction, and S is $>$, then $\varepsilon = 1$ and $S' = 1$.

Note that the next symbol S' is built into the local rule just declared, so that it is brought again to a node label as a result of a later execution of the local rule. The analogy that comes to mind is that a local rule is like a locally declared procedure which, when called, brings back the environment of its static parent. The first two components of labels are used to guarantee that the local rule is executed at a proper moment. The value 1 assigned to ε is a signal that the next global rule must complete the queue modification. This is the role of another rule, denoted G', that is applied to labels with the sixth component equal to 1. Links are again of two kinds and we show both on Fig. 7.

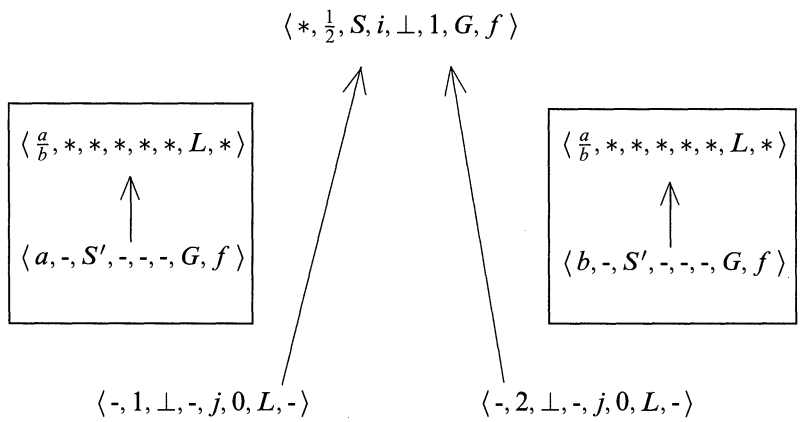


FIGURE 7. Rule G'

Now we know that the rule will be used only when the fifth component is $\perp$ (unlike rule G used both before and after the next instruction is determined). Also we know that S is not arbitrary and we have only links satisfying one of the following cases:

1. The number i refers to an instruction of the form $remove(x)$; *case* x *of* $0 : j_0,\ 1 : j_1$; and S is either 0 or 1. Then S' is $<$ and $j = j_0$ if $S = 0$, and $j = j_1$ otherwise.
2. The number i denotes a number of an *insert* instruction, and $S = \sharp$. Then S' is $>$ and $j = i + 1$.

There is one more global rule, to be used at the end of each phase. We denote this rule by EG. It consists again of two types of links, described on Figure 8. This completes the definition of our tree-maker, and we are now facing the task of proving the correctness of our construction.

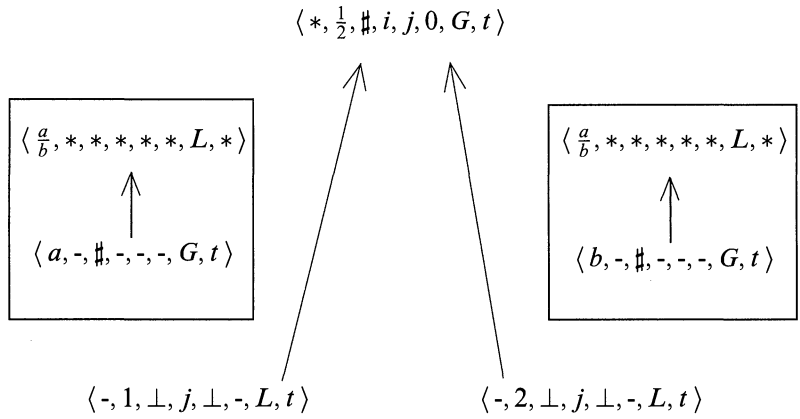


FIGURE 8. Rule *EG*

PROOF OF LEMMA 4.1. It should be obvious that the construction just described is effective, so we have only to show that $T_{\mathscr{A}}$ has a closed run iff $\mathscr{A}$ terminates.

($\Rightarrow$) We begin with collecting some basic information about the possible behaviour of $T_{\mathscr{A}}$. First note that the only rule applicable at the initial node (labelled $\langle \bot, 1, <, 1, 1, 0, G, f \rangle$) is rule B. This is because rule EB requires a "♯" at the third place, and no other global rule permits two 1's at the fourth and fifth place simultaneously. After the second step there are two rules applicable: B and EB, so the closed run must begin with a series of B's and then EB must be applied, say, at the m-th step. Since then, every second level is added by a local rule, and this is controlled by the second last label component switching from L to G and back all the time. Note that rules B and EB cannot be applied anymore, as the first component may no longer be equal to $\bot$. It must be either a or b, and the labels are distributed at every level in groups of alternating a-labels and b-labels. If there is 2^p groups, we say that the *degree of distribution* is p.

As in the examples we considered before, the second components of labels at each level are alternating 1's and 2's. Of course, we have 2^i nodes at each level $i \leq m$ and 2^{m+p} nodes at each level $m + 2p$ and $m + 2p + 1$, for all $p \geq 0$. The i-th local rule (i.e., the local rule declared at step i) is applicable only at a level with as many alternating groups of a's and b's as there are alternating 1's and 2's at level $i - 1$. That is,

(a) The i-th local rule, for $i \leq m$, requires degree of distribution $i - 1$;
(b) The $(m+2p)$-th local rule, for $p \geq 0$, requires degree of distribution $m+p-1$.

Of course, the degree of distribution at level m is 0, as there is only one group. Now observe that an application of a global rule does not change the degree of distribution, while an application of a local rule increases it by 1 (duplicates the number of groups). Since local and global rules alternate, we conclude that:

(c) For $p \geq 0$, the degree of distribution at level $m + 2p$ is p.

It follows from (a) and (c) above that:

(e) For $i \leq m$, the i-th local rule is applied at level $m + 2(i - 1)$;

(f) For $p \geq 0$, the $(m+2p)$-th local rule is applied at level $m+2(m+p-1)$ (i.e., after $2m-2$ steps, or $m-1$ "double steps", each consisting of a "local" step, followed by a "global" step).

The unary links declared by EB have t as the last components of the sons, and the reader can easily see that this forces rule EG to be applied repeatedly after each group of $2m$ steps. Since the final labels are of the form "$\langle *, *, \perp, k, \perp, 0, L, t \rangle$", a closed run must end after an application of EG (only this rule can bring labels carrying L and t as their last two components). Thus it must consist of a number of complete phases, the initial phase of m steps, all the other phases of $2m$ steps.

Now assume that there exists a closed run of $T_{\mathscr{A}}$, and let m be as above. Assume that the depth of the whole run is $m+2mr$. Let w_0 be the string $<>\sharp^{m-2}$. For $n = 1, \ldots, r-1$, we define w_n by induction as follows. Let $w_{n-1} = \$^s<\varepsilon u>\sharp^t$ be of length m, and assume $t > 0$. Then:

- If the instruction executed at step n is $insert(\rho)$, then $w_n = \$^s<\varepsilon u \rho>\sharp^{t-1}$;
- If the instruction executed at step n is $remove(x)$; $case\ x\ of\ 0 : j,\ 1 : k$ then $w_n = \$^{s+1}<u>\sharp^t$.

Thus, the sequence $w_0 \cdot w_0 \cdot w_1 \cdot w_2 \cdots w_{r-1}$ represents the "queue history", as defined before, provided all the words w_n are well-defined, i.e., the queue does not move too far to the right.

It follows straight from the definitions that we have the folowing property, for all $i = 1, \ldots, m$:

(g) The i-th symbol of w_0 occurs as the third component of node labels at level $i-1$.

We show the following claims by a simultaneous induction, for all $n = 0, \ldots, r-1$.

(1) The word w_n is well-defined, and has at least one $\sharp$ at the end.
(2) The fourth component of node labels at level $m+2mn+j$, for all $j = 0, \ldots, 2m-1$, is the number of the instruction executed at step n.
(3) The fourth component of node labels at level $m+2mn+2m$ is the number of the instruction executed at step $n+1$.
(4) For $i = 1, \ldots, m$, the i-th symbol of w_n occurs as the third component of node labels at level $m+2mn+2i-1$.

Let us start with $n = 0$. Of course, w_0 is well-defined and must end with $\sharp$, because otherwise rule EB could not be applied. Part (2), for $n = 0$, follows from the fact that no rule, except EG, can change the fourth component, which is initially set to 1. Part (4) follows from the statements (e) and (g) above, because the third component at level $m+2i-1$ is brought by the execution of the i-th local rule, i.e., it is copied from the third component at level $i-1$.

The proof of part (3) is essentially the same for all $n \geq 0$. Since w_n contains exactly one pair of $<$ and $>$ (here we use part (4)), rule G' must be exactly once executed during the corresponding phase of our run (between levels $m+2mn$ and $m+2mn+2m-1$). This sets the fifth component of labels to the next instruction, and this component is shifted to the fourth position by the execution of EG at the end of the phase, i.e., at level $m+2mn+2m$. It remains to check that the number of the next instruction is correctly determined, but this can be seen from the definitions of rules G and G'.

For $n > 0$, first note that w_n is well-defined, because w_{n-1} ends with a $\sharp$. Part (2) follows from part (3) of the induction hypothesis, again because no rule, except EG, can change the fourth component of labels. Part (4) requires a little calculation to see that the third component at level $m + 2mn + 2i - 1$ is introduced by the $(m+2m(n-1)+2i)$-th local rule. Thus, it is either the same as the third component at level $m+2m(n-1)+2i-1$ (i.e., the i-th symbol of w_{n-1}) or is computed according to the definitions of G and G'. From these definitions it can be seen that what we get is the i-th symbol of w_n, provided the fourth component correctly represents the machine instruction. But this is guaranteed by part (2) of the induction hypothesis.

It remains to complete the proof of part (1) for $n > 0$. It can go wrong only when $w_{n-1} = \$\cdots\$<\cdots>\sharp$, and an *insert* instruction is executed. This corresponds to rule G applied at level $m+2m(n-1)+2m-3$ and reading a label with $>$ at the third position. This sets the sixth component to 1, which only permits rule G' as the next global one. But the last component of the labels at level $m + 2m(n-1) + 2m - 1$ is t and this only permits rule EG. This means that our run could not be extended any more, and this is of course a contradiction.

From part (3) of the above claim we conclude that the machine state after completing r steps of computation must be k, because the labels at level $m + 2rm$ are final. The conclusion is that $T_{\mathscr{A}}$ can complete a closed run only if the computation of $\mathscr{A}$ reaches the final state k.

($\Leftarrow$) For the opposite direction, let us assume that $\mathscr{A}$ terminates in r steps. This means that the queue can be represented as a word over Q of length not exceeding $r + 3$, in such a way that the last "$\sharp$" is never replaced by a "$>$". Consider a run of $T_{\mathscr{A}}$, beginning with an initial phase of depth $m = r + 3$. By induction on n, one can show that such a run can be extended to depth $m + 2mn$, so that the third and fourth components of the labels correctly represent the machine instructions and queue symbols. The proof is similar to the previous one and is left to the reader. Since $\mathscr{A}$ reaches the final state after r steps, the last application of rule EG assigns the final labels to all nodes at level $m + 2mr$, and we obtain a closed run. ⊣

§5. Open problem. Following Leivant [15], we define the *rank* of an intersection type (without ω) as follows: $rank(\tau) = 0$, whenever τ is a simple type (without occurrences of $\cap$), $rank(\tau \cap \sigma) = \max(1, rank(\tau), rank(\sigma))$, and $rank(\tau \to \sigma) = \max(1 + rank(\tau), rank(\sigma))$, if $\cap$ occurs in $\tau \to \sigma$. That is, types of rank 1 are intersections of simple types, and types of rank $n + 1$, for $n > 0$, are function types applicable to arguments of rank n.

The good types we used in our undecidability proof are all of rank at most three, thus we have actually shown that the *entailment* problem

Given E and τ, is there M such that $E \vdash M : \tau$?

is undecidable already for E and τ of rank at most three. However, a transformation of the above problem into the original one (where the environment is empty) increases the rank by one, because each type of E becomes now an argument (cf. the proof of Lemma 3.1). Thus we have shown that emptiness is undecidable for rank four types.

The reader is invited to check that a suitable generalization of the algorithm of [21] (explained also in [23]) can be applied to entailment problem for intersection types of rank 1, thus proving emptiness decidable in PSPACE for rank 2. The problem remains open for rank 3 (with entailment open for rank 2).

Let us note that the tree-maker T_1 constructed at the beginning of Section 4 can be represented by types of rank 2. It follows that a search for an inhabitant of a rank 2 type (in presence of rank 2 assumptions) may lead to an infinite sequence of different environments. This observation excludes a naive approach to prove decidability. However, our further construction corresponds to types of rank 3, because the edge labels may contain more than one unary link. Types corresponding to our boxes are intersections of types of the form $\alpha \to \beta$, where α and β are type variables. The question is whether a similar effect can be obtained if we use complex simple types instead.

It is perhaps worth noting here that the dual problem of type reconstruction is decidable for intersection types of rank at most 2, see [12, 13], and undecidable in general. But the undecidability proof of [15] uses (implicitely) types of unbounded rank, and does not imply the result for any finite rank. Thus, type reconstruction for intersection types of any fixed rank above rank 2 also remains an open problem.

Acknowledgment. The author thanks Mariangiola Dezani-Ciancaglini for her patience in providing all kinds of useful explanations (including finding bugs in an early attempt to prove decidability) and Igor Walukiewicz for reading an initial version of the proof.

REFERENCES

[1] F. ALESSI and F. BARBANERA, *Strong conjunction and intersection types*, ***Proceedings of the MFCS 1991*** (A. Tarlecki, editor), LNCS 520, Springer, Berlin, 1991, pp. 64–73.

[2] H. BARENDREGT, M. COPPO, and M. DEZANI-CIANCAGLINI, *A filter lambda model and the completeness of type assignment*, this JOURNAL, vol. 48 (1983), pp. 931–940.

[3] F. CARDONE and M. COPPO, *Two extensions of Curry's type inference system*, ***Logic and Computer Science*** (P. Odifreddi, editor), Academic Press, 1990, pp. 19–75.

[4] M. COPPO and M. DEZANI-CIANCAGLINI, *An extension of basic functionality theory for lambda-calculus*, ***Notre Dame Journal of Formal Logic***, vol. 21 (1980), pp. 685–693.

[5] M. COPPO, M. DEZANI-CIANCAGLINI, and B. VENNERI, *Functional character of solvable terms*, ***Zeitschrift für Mathematische Logik und Grundlagen der Mathematik***, vol. 27 (1981), pp. 45–58.

[6] M. DEZANI-CIANCAGLINI, S. GHILEZAN, and B. VENNERI, *The "relevance" of intersection and union types*, ***Notre Dame Journal of Formal Logic*** (1997), to appear.

[7] G. DOWEK, *The undecidability of type assignment in the λΠ-calculus*, ***Proceedings of Typed Lambda Calculi and Applications*** (M. Bezem and J.F. Groote, editors), LNCS 664, Springer-Verlag, Berlin, 1993, pp. 139–145.

[8] D.M. GABBAY, ***Semantical Investigations in Heyting's Intuitionistic Logic***, D. Reidel Publ. Co., 1981.

[9] J.-Y. GIRARD, Y. LAFONT, and P. TAYLOR, ***Proofs and Types***, Cambridge University Press, 1989.

[10] J.R. HINDLEY, *The principal type-scheme of an object in combinatory logic*, ***Transactions of the American Mathematical Society***, vol. 146 (1969), pp. 29–60.

[11] W.A. HOWARD, *The formulae-as-types notion of construction*, in ***To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*** (J.P. Seldin and J.R. Hindley, editors), Academic Press, 1980, pp. 479–490.

[12] A.J. KFOURY and J. TIURYN, *Type reconstruction in finite-rank fragments of the second-order λ-calculus*, ***Information and Computation***, vol. 98 (1989), no. 2, pp. 228–257.

[13] A.J. KFOURY and J.B. WELLS, *A direct algorithm for type inference in the rank-2 fragment of the*

second-order λ-calculus, ***Proceedings of LISP and Functional Programming*** ACM (1994), pp. 196–207.

[14] T. KURATA and M. TAKAHASHI, *Decidable properties of intersection type systems*, ***Proceedings of Typed Lambda Calculi and Applications*** (M. Dezani-Ciancaglini and G. Plotkin, editors), LNCS 902, Springer-Verlag, Berlin, 1995, pp. 297–311.

[15] D. LEIVANT, *Polymorphic type inference*, ***Proceedings of the 10th ACM Symposium on Principles of Programming Languages***, Austin, Texas, 1983, pp. 88–98.

[16] M. H. LÖB, *Embedding first order predicate logic in fragments of intuitionistic logic*, this JOURNAL, vol. 41 (1976), no. 4, pp. 705–718.

[17] E.G.K. LOPEZ-ESCOBAR, *Proof Functional Connectives*, ***Proceedings of Methods in Mathematical Logic, 1993***, LNMath 1130, Springer-Verlag, Berlin, 1985, pp. 208–221.

[18] R. MILNER, *A theory of type polymorphism in programming*, ***Journal of Computer and System Sciences***, vol. 17 (1978), pp. 348–375.

[19] G.E. MINTS, *The completeness of provable realizability*, ***Notre Dame Journal of Formal Logic***, vol. 30 (1989), pp. 420–441.

[20] G. POTTINGER, *A type assignment for the strongly normalizable λ-terms*, ***To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*** (J.P. Seldin and J.R. Hindley, editors), Academic Press, London, 1980, pp. 561–577.

[21] R. STATMAN, *Intuitionistic propositional logic is polynomial-space complete*, ***Theoretical Computer Science***, vol. 9 (1979), pp. 67–72.

[22] P. URZYCZYN, *The emptiness problem for intersection types*, ***Proceedings of the 9th IEEE Symposium on Logic in Computer Science***, Paris, France, 1994, pp. 300–309.

[23] ———, *Type inhabitation in typed lambda calculi (a syntactic approach)*, ***Proceedings of Typed Lambda Calculi and Applications*** (P. de Groote and J.R. Hindley, editors), LNCS 1210, Springer-Verlag, Berlin, 1997, pp. 373–389.

[24] ———, *Type reconstruction in* $\mathbf{F}_\omega$, ***Mathematical Structures in Computer Science***, vol. 7 (1997), pp. 329–358, (Preliminary version in ***Proceedings of TLCA***, 1993.).

[25] S. VAN BAKEL, ***Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems***, Mathematisch Centrum, Amsterdam, 1993.

[26] B. VENNERI, *Intersection types as logical formulae*, ***Journal of Logic and Computation***, vol. 4 (1994), no. 2, pp. 109–124.

[27] J.B. WELLS, *Typability and type checking in the second-order λ-calculus calculus are equivalent and undecidable*, ***Proceedings of the 9th IEEE Symposium on Logic in Computer Science***, Paris, France, 1994, (To appear in ***Annals of Pure and Applied Logic.***), pp. 176–185.

[28] J.B. WELLS, A. DIMOCK, R. MULLER, and F. TURBAK, *A typed intermediate language for flow-directed compilation*, ***Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Development, Lille, France***, LNCS 1214, Springer-Verlag, Berlin, 1997, pp. 757–771.

INSTITUTE OF INFORMATICS
UNIVERSITY OF WARSAW
UL. BANACHA 2, 02-097 WARSZAWA, POLAND
E-mail: urzy@mimuw.edu.pl